UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA

Corso di Laurea triennale in
Sicurezza dei sistemi e delle reti informatiche - online

APPROCCI AUTOMATIZZATI PER L'ANALISI DI SMART CONTRACT

Relatore: Prof. Marco Anisettu

Tesi di Laurea di:
Alberto Ameglio
Matr. Nr. 950144

ANNO ACCADEMICO 2022-2023

Ringraziamenti

Desidero ringraziare sinceramente la Schielder S.r.l per avermi dato l'opportunità di svolgere il tirocinio, e in particolare il mio tutor aziendale, il Dott. Davide Silvetti. Vorrei anche esprimere la mia gratitudine Prof. Marco Anisetti per il supporto durante il percorso. Infine, ci tengo a ringraziare la mia famiglia per il grande sostegno che ho avuto durante questi anni. In particolare, dedico il mio percorso a mia madre.

Indice

Ringraziamenti **i**

Indice **ii**

1 Introduzione alla decetralizzazione **1**

1.1 Introduzione 1

1.2 Applicazioni decentralizzate o (DApp) 5

1.2.1 Storia delle Dapp (Decentralized Application) 7

1.3 Differenza tra i sistemi distribuiti e decentralizzati 9

1.4 Introduzione a Web3 11

1.4.1 Web 1.0: solo lettura (1990-2004) 11

1.4.2 Web 2.0: scrittura a lettura (2004 - oggi) 12

1.4.3 Web 3.0: Leggi-Scrivi-Possiedi 12

2 Bitcoin **15**

2.1 Le tecnologie dietro a Bitcoin 15

2.1.1 Le transazioni in Bitcoin 16

2.1.2 Timestamp 17

2.1.3 Lo stato nelle transazioni in Bitcoin 18

2.1.4 Network 20

2.2 Nakamoto Consensus 20

2.2.1 Byzantine Fault Tolerant 21
2.2.2 Proof of Work 23
2.2.3 Selezione del Blocco 24
2.2.4 Incentivo per la creazione dei blocchi 25
2.3 Mining 25
2.4 Gestione dello spazio 27
2.4.1 Merkle Tree 28
2.4.2 Longest Chain 29
2.4.3 Combinare e dividere il valore 31
2.5 Riservatezza 31
2.5.1 Dimostrazione 32
2.6 Applicazioni alternative alla blockchain di Bitcoin 35
2.7 Lo scripting implementato in Bitcoin 36

3 Ethereum 38

3.1 La nascita di Ethereum 38
3.1.1 Una Blockchain General-Purpose 40
3.2 I Componenti di Ethereum 41
3.2.1 Ethereum Turing completo 41
3.2.2 Implicazioni della completezza di Turing 42
3.3 L'implementazione di Ethereum 43
3.3.1 Gli Account 44
3.3.2 Introduzione alle transazioni 45
3.3.3 Introduzione ai messaggi 45
3.4 Le transazioni in Ethereum 46
3.4.1 Il Nonce nella transazione 47
3.4.2 Implicazioni applicate al nonce 48

3.4.3 La concoorenzza nelle transazioni 49
3.4.4 Il destinatario nelle transazioni 50
3.5 Funzione dello stato di transizione di Ethereum 50
3.5.1 Esecuzione della transazione 51
3.5.2 La propagazione delle transazioni 52
3.5.3 Registrazione sulla Blockchain 53
3.5.4 Blockchain e mining 53
3.6 Gas 54
3.6.1 Valuta ed emissione 55
3.7 Proof of Stake (PoS) 56
3.7.1 I vantaggi del Proof of Stake rispetto al Proof of work 57

4 Smart Contract e Solidity **58**
4.1 Smart Contract 58
4.2 Ciclo di vita di uno Smart Contract 59
4.3 Costruire uno Smart contract con solidity 60
4.4 Scrittura di un contratto 63
4.5 Compilazione con il Solidity Compiler (solc) 65
4.6 Application Binary Interface 67
4.7 La macchina virtuale di Ethereum 67
4.8 Il set di istruzioni EVM (operazioni bytecode) 70
4.9 Stato di Ethereum 73
4.10 Disassemblare il Bytecode 77

5 Sicurezza negli Smart contract **82**
5.1 Reentrancy 82
5.2 Over/Underflow aritmetici 86
5.3 Unexpected Ether 91

5.4 Default Visibilities 94
5.5 Entropy Illusion 96
5.6 External Contract Referencing 97
5.7 Short Address/Parameter Attack 103
5.8 Unchecked CALL Return Values 105
5.9 Race Conditions/Front Running 108
5.10 Denial of Service (DoS) 110
5.11 Block Timestamp Manipulation 113
5.12 Constructors with Care 114
5.13 Tx.Origin Authentication 116

6 Analisi automatizzata per gli Smart Contractt 118

6.1 Unit Test Overflow/Underflow 118
6.2 Unit Test Reentrancy 124
6.3 Unit Test Delegate Call Attack Vectors 126
6.4 Analisi Automatica 131
6.5 Analisi Statica - Slither 133
6.6 Analisi Dinamica - Echidna 138
6.7 L'insufficienza dei test Unitari 144
6.8 Property-Based Tester 146
6.9 Conclusioni 159

A Glossario 160

Bibliografia 168

Capitolo 1

Introduzione alla decetralizzazione

1.1 Introduzione

Negli ultimi anni le criptovalute e la tecnologia “blockchain” hanno raggiunto un’enorme popolarità e attrazione da parte del mondo accademico, arrivando, ad ottobre 2021, con una capitalizzazione totale di circa 2,6 triliardi [47] nel mercato delle criptovalute e nella “DeFi” (finanza decentralizzata). Solamente il TVL (Total Value Locked) ovvero il totale degli assets vincolati all’interno di un ecosistema DeFi ha raggiunto nel primo trimestre del 2022 un valore di 89,5 miliardi.[50]

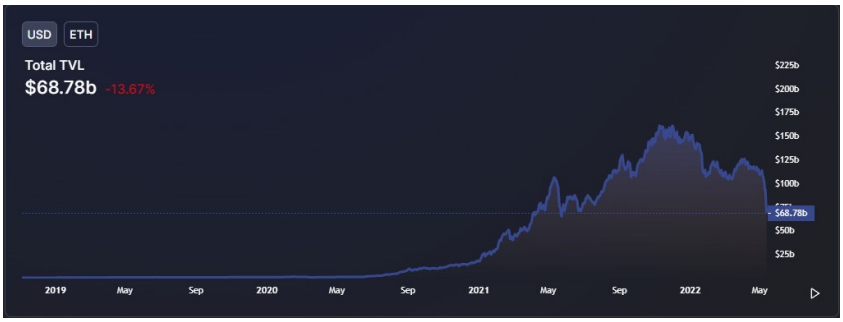

Figura 1: Grafica il TVL nel Q1 2022
[50]

Una criptovaluta utilizza un protocollo di consenso ben progettato, che viene concordato da tutti i nodi che ne fanno parte, e dai nodi di calcolo o di convalida che sono responsabili della registrazione delle transazioni nella rete.
La blockchain è una rete software peer-to-peer distribuita, che utilizza la crittografia per archiviare in modo sicuro applicazioni e dati, e trasferire facilmente strumenti digitali. Tale tecnologia

è stata originariamente proposta, con Bitcoin, come un protocollo per il trasferimento di denaro senza un mutuo accordo tra le parti. Dopo l'arrivo di Bitcoin sono state progettate diverse piattaforme blockchain, tra le quali una delle più rilevanti oggi sul mercato è Ethereum che ha aggiornato il suo protocollo con un linguaggio di programmazione quasi Turing completo, consentendo agli sviluppatori di scrivere "smart contract" e applicazioni decentralizzate.
L'ecosistema di Ethereum è spesso descritto come "the world computer" e negli ultimi anni ha visto una rapida crescita del numero di smart contracts e applicazioni decentralizzate che, solo nel quarto trimestre del 2022, sono stati implementati più di 4.6 milioni di smart contract [46].

Figura 2: Q4'22 Web3 Development Report
[46]

La tecnologia di Ethereum è una piattaforma open source che, a livello globale, esegue programmi chiamati "smart contracts", e che, insieme a una criptovaluta chiamata "ether", usata per misurare e limitare i costi delle risorse in esecuzione, utilizza la rete blockchain per sincronizzare e memorizzare i cambiamenti di stato nel sistema. La blockchain, in relazione con le criptovalute, può essere considerata come un "libro mastro" distribuito e condiviso o "ledger".
Ethereum, dal punto di vista informatico, è infatti una macchina a stati deterministici costituita da un pattern singleton accessibile a livello globale, con capacità praticamente illimitata, ed una macchina virtuale che applica modifiche al suo stato corrente.

Ethereum condivide molte caratteristiche con Bitcoin. In particolare entrambe sono reti peer-to-peer che coordinano i partecipanti con un *"Byzantine fault–tolerant consensus algorithm"* utilizzando, per sincronizzare gli aggiornamenti di stato, una blockchain di tipo proof-of-work, basata sull'algoritmo di consenso di Nakamoto e primitive crittografiche con firme digitali e hash. Ether, o ETH, è la valuta nativa di Ethereum. È una "moneta digitale" che può essere inviata via Internet istantaneamente e a basso costo, ed è parte integrante e necessaria per il funzionamento di Ethereum. Il modo più semplice per ottenere ETH è acquistarli. Ci sono molte piattaforme di scambio per criptovalute su cui è possibile acquistare ETH essendo il suo funzionamento è perfettamente integrato nell'ecosistema. Questa valuta è intesa come una "utility currency" per i pagamenti di piattaforma.
Quando consideriamo questa tecnologia, è possibile usare il termine "Ethereum" per riferirsi a

tre entità distinte: il protocollo Ethereum, la rete Ethereum, creata dai nodi dei computer che utilizzano il protocollo, e il progetto Ethereum che finanzia lo sviluppo dei due.

Il rischio nella sicurezza degli smart contract è un tema di vitale importanza in un mondo in cui la blockchain e i sistemi decentralizzati stanno prendendo sempre più importanza. Gli smart contract, essendo autonomi ed eseguiti su una rete decentralizzata come Ethereum, portano con sé sfide uniche e potenziali vulnerabilità che possono mettere a rischio i fondi e i dati degli utenti. Questi rischi includono vulnerabilità di programmazione, come bug nel codice degli smart contract, che possono essere sfruttati da attaccanti per rubare fondi o compromettere la funzionalità del contratto.
La sicurezza degli smart contract richiede una profonda comprensione della tecnologia blockchain, delle best practice di sviluppo e degli strumenti di analisi delle vulnerabilità. Ignorare il rischio associato alla sicurezza degli smart contract può comportare conseguenze gravi e costose sia per gli sviluppatori che per gli utenti.

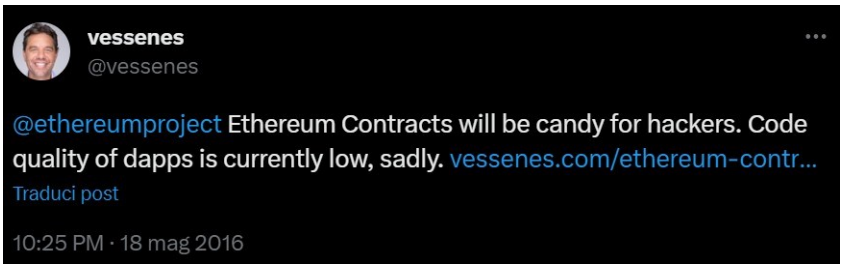

Figura 3: Peter Vessenes è il fondatore e CEO di New Alchemy e ha profonde conoscenze in Bitcoin, tokenizzazione e tecnologia blockchain. Ha anche co-fondato la Bitcoin Foundation e fondato la prima società Bitcoin sostenuta da venture capital
[49]

Il report "The 2023 Crypto Crime Report"[7] è stato prodotto dalla società Chainalysis, una società americana di analisi blockchain con sede a New York e prima start-up dedicata all'attività di tracciamento di Bitcoin. Tra i suoi cui clienti figurano l'FBI (Federal Bureau of Investigation degli Stati Uniti) , la DEA (Drug Enforcement Administration) , IRS-CI(Internal Revenue Service Criminal Investigation) , NCA (National Crime Agency del Regno Unito) e altri.
Tale report descrive come il 2022 sia stato l'anno più importante di sempre per l'hacking di criptovalute, con 3,8 miliardi di dollari rubati, principalmente dai protocolli DeFi. Tali dati non includono le cause per frode avvenute in tale periodo (Celsius, Three Arrows Capital, FTX e altre).
Qui di seguito alcuni dati e grafici ripresi direttamente dal Report.

Citando : *"Il 2022 è stato l'anno più importante di sempre per l'hacking di criptovalute, con 3,8 miliardi di dollari rubati alle aziende che operano nel settore delle criptovalute. L'attività di hacking ha avuto alti e bassi durante tutto l'anno, con picchi enormi a marzo e ottobre, l'ultimo dei quali è diventato il mese più grande di sempre per l'hacking di criptovalute, poiché 775,7 milioni di dollari sono stati rubati in 32 attacchi separati."*

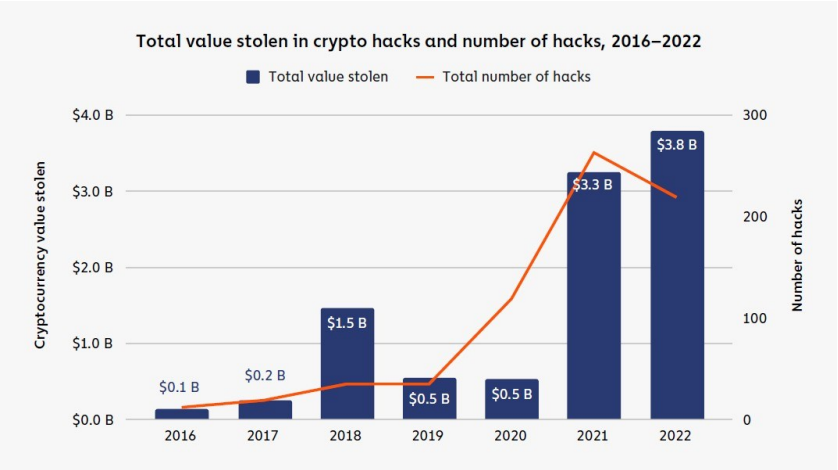

Figura 4: Total Value Stolen in crypto hacks and number of hacks [7]

E' interessante notare come il settore delle cybercrime in ambito Web3 stia prendendo anch'esso una sua rilevanza economica e politica nel mondo. Il report sottolinea anche l'esistenza del famoso gruppo hacker, Lazarus Group della corea del nord, che nel solo 2022 ha infranto il proprio record rubando criptovalute per un valore stimato di 1,7 miliardi di dollari attraverso diversi attacchi. L'intero PIL della corea del nord è stimato attorno ai 20 miliardi e non sorprende che la maggior parte degli esperti concordino sul fatto che il governo nordcoreano stia utilizzando questi fondi rubati per finanziare il proprio arsenale nucleare.

Nel corso di questo elaborato, si seguirà una panoramica completa dei passi chiave che hanno condotto all'evoluzione di Ethereum, con particolare enfasi su Solidity e l'EVM. Per farlo, ho scelto di intraprendere un percorso partendo da Bitcoin come pietra miliare nel panorama delle criptovalute e delle blockchain e da una breve comprensione dei concetti che compongono il Web3.
Una parte significativa del mio elaborato sarà dedicata all'analisi delle vulnerabilità e alla ricerca delle stesse.Questa parte comprende una descrizione dettagliata dei tipi di vulnerabilità più comuni, illustrando come queste minacce possano essere sfruttate dagli attaccanti.
Nel corso di questa ricerca, verranno esplorati una serie di strumenti e tecniche ampiamente riconosciute e utilizzate per l'analisi di smart contract; e per la ricerca di vulnerabilità all'interno dell'ecosistema Ethereum e delle applicazioni decentralizzate.Inoltre per approfondire ulteriormente il mio studio sulla sicurezza degli smart contract e per ottenere una comprensione più approfondita dei processi di analisi e ricerca di vulnerabilità, ho sviluppato uno strumento personalizzato in Python. Questo strumento è focalizzato sull'esecuzione del fuzzing e sul test di invarianti nei contratti. Ciò ha consentito di mettere in pratica le conoscenze acquisite durante la ricerca mediante un approccio innovativo.

Iniziamo subito con una breve panoramica dei concetti fondamentali della decentralizzazione,

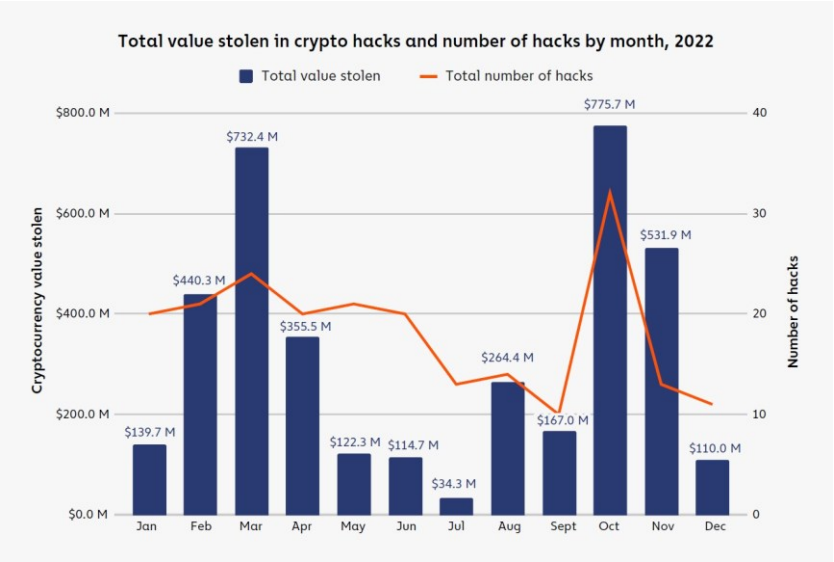

Figura 5: Total vaue stolen in crypto hacks and number of hacks by month 2022 [7]

comprendendo cosa sia una Dapp, cioè un'applicazione decentralizzata, e analizzando i passaggi che hanno portato a questa significativa evoluzione tecnologica, nota come il Web3.

1.2 Applicazioni decentralizzate o (DApp)

La visione dei fondatori di Ethereum non si limitava solamente al concetto di "smart contract", ma si ampliava al concetto di applicazioni decentralizzate , o DApp, considerando quindi di riprogettare l'ecosistema su cui si basa il web e creare un nuovo mondo di DApp, chiamato web3. Gli "smart contract" non sono solo un modo per decentralizzare la logica di controllo delle funzioni di pagamento nelle applicazioni. Le DApp Web3 riguardano il decentramento di tutti gli altri aspetti di un'applicazione: archiviazione, messaggistica, denominazione, ecc.

La struttura di una DApp.

Una DApp è un'applicazione che è per lo più, o interamente, decentralizzata e che considera tutti i possibili aspetti di un'applicazione che può essere decentralizzata:

- Software di back-end (logica dell'applicazione)
- Software di front-end
- Archiviazione dati

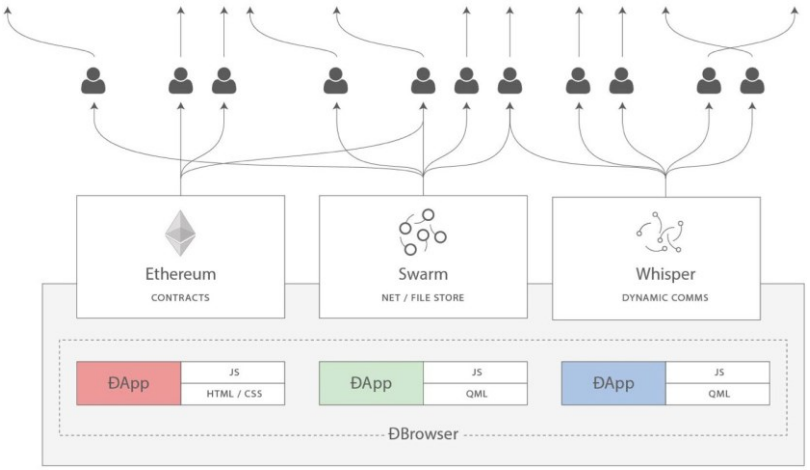

Figura 6: Grafica che rappresenta la DApp
[52]

- Comunicazione
- Risoluzione del dominio

Ognuno di questi aspetti può essere in qualche modo centralizzato o decentralizzato. Ad esempio, il front-end può essere sviluppato come una Web app che viene eseguita su un server centralizzato o come un'app mobile che viene eseguita sul tuo dispositivo. Il back-end e l'archiviazione possono trovarsi su server e database di proprietà, oppure è possibile utilizzare "smart contract" e l'archiviazione con P2P
Ci sono molti vantaggi nella creazione di una DApp che una tipica architettura centralizzata non può fornire:

Resilienza
Poiché la logica dell'applicativo è controllata da uno "smart contract", un backend DApp sarà completamente distribuito e gestito su una piattaforma blockchain. A differenza di un'applicazione distribuita su un server centralizzato, una DApp non avrà tempi di inattività e continuerà a essere disponibile finché la piattaforma è ancora operativa.

Trasparenza
La natura on-chain di una DApp consente a tutti di ispezionare il codice ed essere più sicuri nel suo utilizzo. Qualsiasi interazione con la DApp verrà archiviata per sempre nella blockchain.

Resistenza alla censura
Finché un utente ha accesso a un nodo della Dapp, sarà sempre in grado di interagire con l'applicativo senza interferenza di alcun controllo centralizzato. Nessun fornitore di servizi, e nemmeno il proprietario dello "smart contract", può modificare il codice una volta distribuito sulla rete.

Nell'ecosistema Ethereum, così com'è oggi, ci sono pochissime app veramente decentralizzate. La maggior parte si affida ancora a servizi e server centralizzati per una parte del loro funzionamento (ovvero servizi definiti off-chain). In futuro, ci si aspetta che la tecnologia DApp possa essere gestita in modo completamente decentralizzato[52].

1.2.1 Storia delle Dapp (Decentralized Application)

Questi protocolli però, inizialmente non riuscirono ad ottenere successo a causa della loro dipendenza da un intermediario centralizzato. Nel 1998, la "b-money" teorizzata da Wei Dai[9] divenne la prima proposta che considerava l'idea di creare denaro risolvendo problemi matematici e sfruttando il consenso decentralizzato. Tuttavia questa era povera di dettagli su come il consenso decentralizzato potesse effettivamente essere implementato.

Nel 2005, Hal Finney introdusse il concetto di *"reusable proofs of work"*, un sistema che utilizza le idee implementate da b-money, unite al concetto di risoluzione di problemi computazionalmente difficili progettati da Adam Back per Hashcash, per creare il concetto di criptovaluta. Ma ancora una volta questo progetto non è riuscito ad ottenere caratteristiche di affidabilità sufficienti, dovendosi affidare, a livello di backend, ad un sistema centralizzato di trusted computing. Nel 2009, per la prima volta, è stato implementato da Satoshi Nakamoto, (nominativo la cui identità non è mai stata accertata), Bitcoin, un sistema completamente decentralizzato che combina primitive consolidate, per la gestione della proprietà, attraverso la crittografia a chiave pubblica, con un algoritmo di consenso, per tenere traccia di chi possiede realmente le monete, noto come "proof-of-work".
Il meccanismo alla base del "proof-of-work" è stato un importante passo in avanti perché ha risolto contemporaneamente due problemi legati al processo di consenso. In primo luogo, ha fornito un algoritmo di consenso semplice e moderatamente efficace, consentendo ai nodi della rete di concordare collettivamente una serie di aggiornamenti frequenti allo stato del libro mastro o "ledger" di Bitcoin. Ha poi fornito un meccanismo per consentire un libero ingresso al processo di consenso, risolvendo il problema organizzativo di aggiornare e di decidere chi può influenzare il sistema.

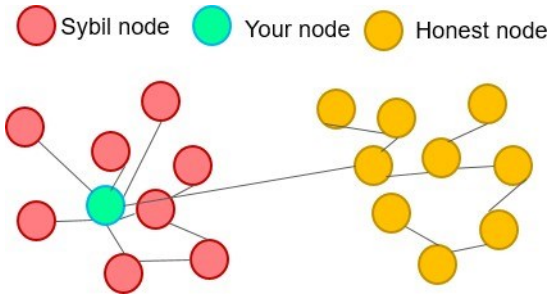

Figura 7: Grafica rappresentativa Sybil attack.
[37]

Tale sistema possiede anche la proprietà di prevenire il cosiddetto "sybil attack", un attacco informatico dove i sistemi di reputazione sono sovvertiti falsificando le identità di una persona

in una rete p2p.
La soluzione a questo problema è stata possibile grazie all'inserimento di una barriera economica per la partecipazione della singola entità, al posto dell'iscrizione dell'individuo ad uno specifico elenco. In questo modo il peso di un singolo nodo nel processo di voto consensuale è direttamente proporzionale alla potenza di calcolo che il nodo apporta alla rete.
Un'altra possibile soluzione all'attacco di Sybil è stata trovata nel "proof-of-stake", che calcola il peso di un nodo come proporzionale alle sue disponibilità di valuta e non alle risorse computazionali. Entrambi gli approcci possono essere utilizzati per fungere da spina dorsale per una criptovaluta.[6]

1.3 Differenza tra i sistemi distribuiti e decentralizzati

Una precisazione tra i sistemi distribuiti e decentralizzati:

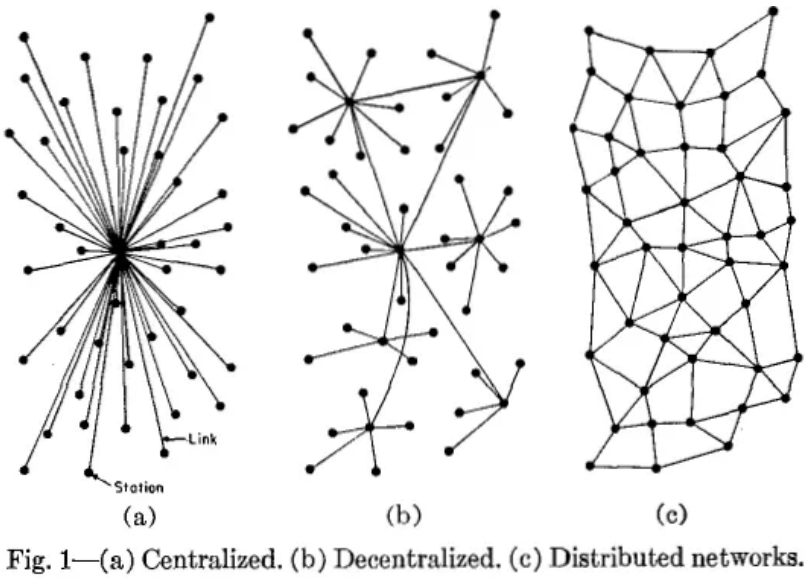

Figura 8: Differenza nei sistemi
[5]

L'elaborazione distribuita ha una lunga storia nell'informatica ed è diventata una branca a sé stante dell'informatica alla fine degli anni '70 e all'inizio degli anni '80. Un sistema distribuito è un modello in cui i componenti situati sui computer collegati in rete comunicano e coordinano le loro azioni passando messaggi.
Con questa definizione un sistema decentralizzato è anche un sistema distribuito. Spesso infatti la differenza tra Distribuito e Decentralizzato non è molto chiara nei sistemi. Sistema distribuito significa che "non tutta l'elaborazione delle transazioni viene eseguita nello stesso posto", mentre decentralizzato significa che "nessuna singola entità ha il controllo su tutta l'elaborazione delle transazioni".

Quando le persone parlano di decentralizzazione del software, in realtà si riferiscono a tre pilastri separati di centralizzazione/decentralizzazione. Anche se in alcuni casi è difficile capire come si possa avere l'uno senza l'altro, in generale sono abbastanza indipendenti l'uno dall'altro. Sono i seguenti:

- (De)centralizzazione architetturale: da quanti computer fisici è composto un sistema? Quanti di questi computer possono tollerare il guasto in una sola volta?
- (De)centralizzazione politica: quanti individui o organizzazioni controllano in definitiva i computer di cui è composto il sistema?

- (De)centralizzazione logica: l' interfaccia e le strutture dati che il sistema presenta e mantiene assomigliano più a un singolo oggetto monolitico o a uno sciame amorfo?
  Ad esempio: se si taglia il sistema a metà, includendo sia i fornitori che gli utenti, entrambe le metà continuerebbero a funzionare pienamente come unità indipendenti?

Possiamo provare a mettere queste tre dimensioni in un grafico:

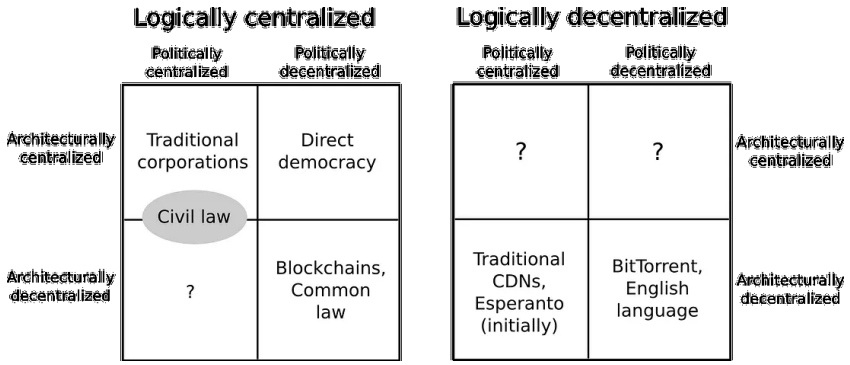

Figura 9: Differenza nelle logiche
[5]

Si faccia presente che molti di questi posizionamenti sono approssimativi e altamente discutibili. Proviamo ad esaminare qualche esempio oltre alla blockchain:

- Le aziende tradizionali sono politicamente centralizzate (un CEO), architetturalmente centralizzate (una sede centrale) e logicamente centralizzate (non possono davvero essere divise a metà).
- Il diritto civile si basa su un organismo legislativo centralizzato, mentre le leggi comuni sono costituite da precedenti stabiliti da molti singoli giudici. Il diritto civile ha ancora una certa decentralizzazione architetturale poiché ci sono molti tribunali che tuttavia dispongono di un'ampia discrezionalità, ma il potere legislativo ne ha di più. Entrambi sono logicamente centralizzati ("la legge è la legge").
- Le lingue sono logicamente decentralizzate; non è necessario che l'inglese parlato tra Alice e Bob e l'inglese parlato tra Charlie e David coincidano affatto. Non è necessaria un'infrastruttura centralizzata affinché una lingua esista, e le regole della grammatica inglese non sono create o controllate da una singola persona.
- BitTorrent è logicamente decentralizzato in modo simile a come lo è l'inglese. Le reti di distribuzione dei contenuti sono simili, ma sono controllate da un'unica società.

Per quanto riguarda le blockchain, seguendo questa classificazione, sono politicamente decentralizzate (nessuno le controlla) e architettonicamente decentralizzate (nessun punto centrale di fallimento infrastrutturale), ma sono logicamente centralizzate (esiste uno stato concordato comunemente e il sistema si comporta come un singolo computer)

1.4 Introduzione a Web3

La centralizzazione ha aiutato a integrare miliardi di persone nel World Wide Web e ha creato l'infrastruttura stabile e solida su cui navigare. Allo stesso tempo, un piccolo gruppo di società centralizzate ha creato un monopolio sul World Wide Web, decidendo unilateralmente cosa dovrebbe e non dovrebbe essere permesso.
Web3 è la risposta a questo problema. Invece di un Web monopolizzato da grandi aziende tecnologiche, Web3 abbraccia la decentralizzazione e viene costruito e gestito principalmente dai suoi utenti. In questo modo Web3 rimette la proprietà dei dati degli utenti nelle loro mani piuttosto che alle società come Google, Meta ed Apple.

1.4.1 Web 1.0: solo lettura (1990-2004)

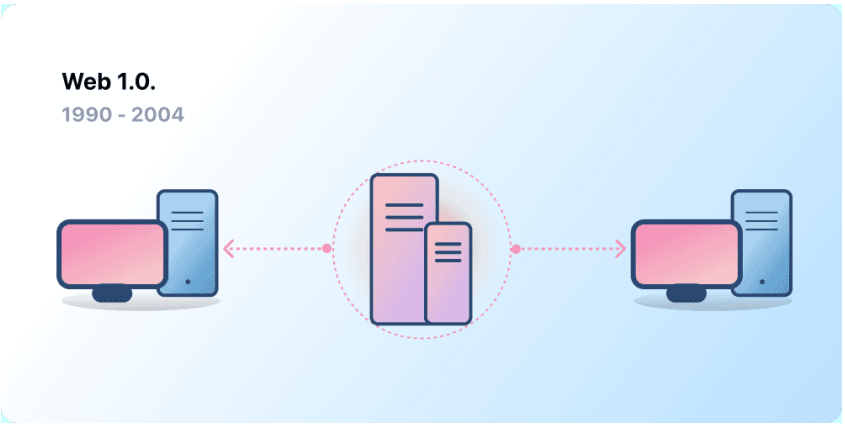

Figura 10: Grafica rappresentativa dell'idea di Web 1.0
[13]

Molte persone pensano al Web come un pilastro continuo della vita moderna. Tuttavia, il Web che molti di noi conoscono oggi è abbastanza diverso da quanto inizialmente immaginato. Per capirlo meglio, è utile suddividere la breve cronologia del Web in periodi storici — Web 1.0 e Web 2.0.
Nel 1989, al CERN, a Ginevra, Tim Berners-Lee era impegnato a sviluppare i protocolli che sarebbero diventati il World Wide Web. Con l'idea di creare protocolli aperti e decentralizzati che consentissero la condivisione delle informazioni in qualsiasi parte della Terra.
La prima fase della creazione di Berners-Lee, ora noto come "Web 1.0", avvenne all'incirca tra il 1990 e il 2004. Il Web 1.0 era principalmente popolato da siti Web statici di proprietà di alcune aziende e non c'era nessuna interazione tra utenti o individui, portando ad essere considerato come il web per la sola lettura.

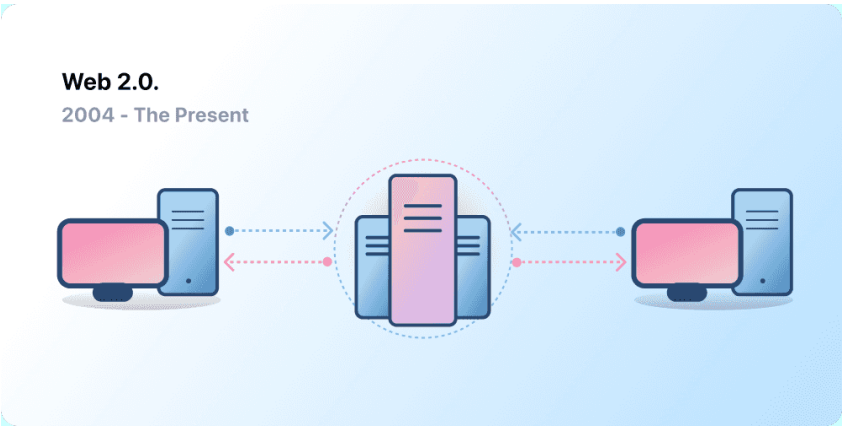

Figura 11: Grafica rappresentativa dell'idea di Web 2.0
[13]

1.4.2 Web 2.0: scrittura a lettura (2004 - oggi)

Il periodo Web 2.0 è iniziato nel 2004 con l'emergere di piattaforme social media. Invece di sola lettura, il web si è evoluto in lettura-scrittura. Fornendo contenuti agli utenti e piattaforme per condividere contenuti generati dagli utenti e impegnarsi in interazioni utente-utente. Con il crescere di un numero maggiore di persone che venivano online, un piccolo numero di grandi aziende ha iniziato a controllare una quantità sproporzionata di traffico e valore generato sul web. Il Web 2.0 ha anche creato il modello di guadagno basato sulla pubblicità, di cui gli utenti che creavano contenuti, non ne possedevano e né beneficiavano della sua monetizzazione.

1.4.3 Web 3.0: Leggi-Scrivi-Possiedi

La premessa che risiede dietro "Web 3.0" è stata coniata dal Ethereum co-fondatore Gavin Wood poco dopo il lancio di Ethereum nel 2014. Gavin ha espresso una soluzione ad un problema che molti tra i primi utenti di criptovalute hanno sentito: il Web richiedeva troppa fiducia, perché la maggior parte del Web che le persone conoscono e usano oggi, si basa sulla fiducia in un piccolo numero di società private che agiscono per il loro interesse.

Web3 è diventato un termine generico per la visione di un nuovo Internet, che sarà possibile con l'utilizzo di blockchain, criptovalute e NFT(not fungible token) permettendo di restituire la proprietà dei dati agli utenti.

Possiamo schematizzare con:

- Web1 era di sola lettura

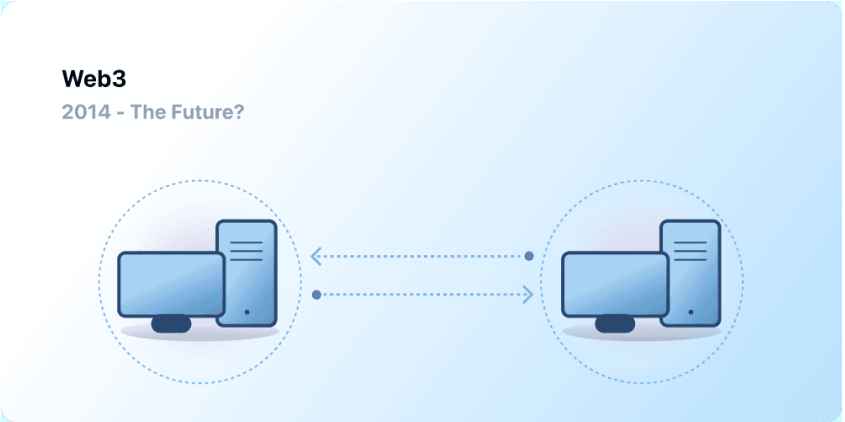

Figura 12: Grafica rappresentativa dell'idea di Web 3.0
[13]

- Web2 è di lettura-scrittura
- Web3 sarà di lettura-scrittura e proprietà

Le idee alla base di Web3
Sebbene sia difficile fornire una definizione di ciò che è Web3 possiamo analizzare alcuni dei suoi pilastri:

- Web3 è decentralizzato: invece di ampie aree di Internet controllate e di proprietà di alcune entità centralizzate, la proprietà viene distribuita tra i suoi costruttori e utenti.
- Web3 è senza autorizzazione: tutti hanno pari accesso per partecipare al Web3 e nessuno viene escluso.
- Web3 ha pagamenti nativi: utilizza la criptovaluta per spendere e inviare denaro online invece di fare affidamento sulla infrastruttura di banche e processori di pagamento.
- Web3 è senza fiducia: opera utilizzando incentivi e meccanismi economici invece di fare affidamento su terze parti fidate.

Le limitazioni di Web3
Nonostante i vantaggi del Web3 nella sua forma attuale, ci sono ancora molte limitazioni che l'ecosistema deve affrontare affinché possa prosperare.

- **Accessibilità:**
  Importanti funzionalità Web3, come Sign-in con Ethereum, sono già disponibili per chiunque da utilizzare a costo zero. Tuttavia, il costo relativo delle transazioni è ancora proibitivo per molti. Web3 ha meno probabilità di essere utilizzato in nazioni in via di sviluppo meno abbienti a causa delle elevate commissioni di transazione.
- **Esperienza utente:**
  La barriera tecnica all'ingresso nell'uso del Web3 è attualmente troppo elevata. Gli utenti devono comprendere i problemi di sicurezza e la complessa documentazione tecnica, e

navigare su interfacce utente non intuitive. Fornitori di ledger, in particolare, stanno lavorando per risolvere, ma sono necessari ulteriori progressi prima che Web3 venga adottato in massa.

- **Formazione:**
  Web3 introduce nuovi paradigmi che richiedono l'apprendimento di modelli mentali diversi da quelli utilizzati in Web2.0. Un simile impulso educativo è avvenuto quando Web1.0 stava guadagnando popolarità alla fine degli anni '90. I sostenitori del world wide web hanno usato una serie di tecniche educative per educare il pubblico utilizzando semplici metafore (l'autostrada dell'informazione, i browser, la navigazione sul web).

L'ecosistema Web3 è giovane e in rapida evoluzione. Di conseguenza, attualmente dipende principalmente dall'infrastruttura centralizzata (GitHub, Twitter, Discord, ecc.). Molte aziende Web3 si affrettano a colmare queste lacune, ma la costruzione di infrastrutture affidabili e di alta qualità richiede tempo.

Un futuro decentralizzato Web3 è un ecosistema giovane ed in evoluzione. Gavin Wood ha coniato il termine nel 2014, ma molte di queste idee sono diventate solo recentemente una realtà. Siamo solo all'inizio della creazione di un Web migliore con Web3, mentre continuiamo a migliorare l'infrastruttura che lo supporterà.[13]

Capitolo 2

Bitcoin

In questo capitolo, esploreremo le tecnologie alla base di Bitcoin e analizzeremo tutti i passaggi e le componenti che hanno contribuito a sviluppare questa tecnologia. Lo faremo attraverso la disamina dei paper che hanno giocato un ruolo significativo nella sua evoluzione.

2.1 Le tecnologie dietro a Bitcoin

L'idea che risiede dietro a Bitcoin è quella di una versione puramente peer-to-peer del contante elettronico che consente di inviare pagamenti online direttamente da una parte all'altra senza passare attraverso un istituto finanziario.
Le firme digitali forniscono parte di questa soluzione, ma i vantaggi principali vengono persi se è ancora necessaria una terza parte fidata per evitare la doppia spesa. La soluzione proposta è una rete contrassegnata dall'ora delle transazioni, inserite in una catena continua di proof-of-work e basate su hash, che forma un record non modificabile senza la ripetizione dell'intero processo. La catena più lunga non serve solo come prova della sequenza degli eventi a cui si sta assistendo, ma è anche la dimostrazione che proviene dal più grande pool di potenza di CPU.
Questo meccanismo garantisce che la maggior potenza di CPU sia controllata da nodi che non cooperano per attaccare la rete ma che porteranno a generare la protezione della catena più lunga. La rete stessa richiede una struttura minima. I messaggi vengono trasmessi in base al massimo sforzo e i nodi possono separarsi e ricongiungersi nella rete a piacimento, accettando la catena più lunga di proof-of-work come prova di ciò che è accaduto.
Questo sistema è pensato come soluzione definitiva ai problemi che oggi sono relegati ai sistemi finanziari moderni.Infatti il commercio su Internet è arrivato a fare affidamento quasi esclusivamente su istituzioni finanziarie che fungono da terze parti fidate per elaborare i pagamenti elettronici.

Sebbene il sistema funzioni bene per la maggior parte delle transazioni, soffre ancora delle debolezze intrinseche del modello basato sulla fiducia tra le parti. Questo perché le transazioni completamente irreversibili non sono realmente possibili, poiché le istituzioni finanziarie non possono evitare di mediare le controversie. Il costo delle relative mediazioni aumenta i costi di ciascuna transazione, limitando la dimensione minima pratica per ciascuna transazione e escludendo la possibilità di piccole transazioni casuali. Con la possibilità dell'inversione della transazione, il bisogno di fiducia tra le parti si diffonde, dai commercianti chiedendo ai loro clienti più informazioni di quelle di cui avrebbero altrimenti realmente bisogno. In questo sistema una certa percentuale di frode è accettata come inevitabile. Questi costi e incertezze sui pagamenti possono essere evitati nelle modalità di pagamento di persona, utilizzando valuta fisica.
Ma prima di questi sistemi non esisteva alcun meccanismo per effettuare pagamenti tramite un canale di comunicazione senza una parte fidata. Per risolvere questo problema è stato proposto un metodo di pagamento elettronico, basato sulla prova crittografica anziché sulla fiducia. Detta prova consente a due parti di effettuare transazioni direttamente tra loro senza la necessità di una terza parte fidata.
Le transazioni che sono computazionalmente impraticabili da annullare, proteggono i venditori da frodi e da meccanismi di deposito a garanzia, e potrebbero essere facilmente implementate per proteggere allo stesso modo gli acquirenti.
La soluzione al problema della doppia spesa può essere affrontata utilizzando un server con gestione di timestamp distribuito peer-to-peer per generare una prova elettronica dell'ordine cronologico delle transazioni. Il sistema è sicuro fintanto che i nodi "trust" controllano collettivamente la maggior potenza di CPU rispetto a qualsiasi altro gruppo cooperante di nodi attaccanti.[41][39]

2.1.1 Le transazioni in Bitcoin

Definiamo ora una moneta elettronica come una catena di firme digitali. Ogni proprietario trasferisce la moneta al successivo firmando digitalmente un hash della transazione precedente e, con la chiave pubblica del proprietario successivo, e aggiungendo tutto il processo alla fine della transazione. In questo modo il beneficiario può controllare le firme con la propria chiave privata assicurandosi della veridicità della catena.

Il problema è che il beneficiario non può verificare se uno dei proprietari ha speso due volte la moneta. Una soluzione comune consiste nell'introdurre un'autorità centrale fidata, o zecca, che controlla la doppia spesa di ogni transazione. Con questo metodo, dopo ogni transazione, la moneta deve essere restituita alla zecca per l'emissione di una nuova moneta, e si ritengono verificate solo le monete emesse direttamente dalla zecca. Il problema con questa soluzione è che il funzionamento dell'intero sistema monetario dipende dalla società che gestisce la zecca, proprio come con una banca. L'unico modo per far sapere al beneficiario di una vendita che i precedenti proprietari non hanno firmato prima altre transazioni, si deve considerare che la prima transazione è l'unica quella che conta, e che non ci si deve preoccupare dei successivi tentativi di doppia spesa. L'unico modo per confermare l'assenza di una transazione è essere a conoscenza di tutte le transazioni avvenute in passato. Nel modello basato sulla zecca, la zecca era a conoscenza di tutte le transazioni e decideva quale arrivava per prima. Per ottenere ciò senza una parte fidata, le transazioni devono essere annunciate pubblicamente, ed è quindi necessario un sistema che consenta ai partecipanti di concordare un'unica cronologia dell'ordine transazioni ricevute. Il beneficiario ha bisogno di una prova che al momento di ogni transazione, la maggior parte dei nodi abbia concordato che fosse la prima ad essere ricevuta.[39]

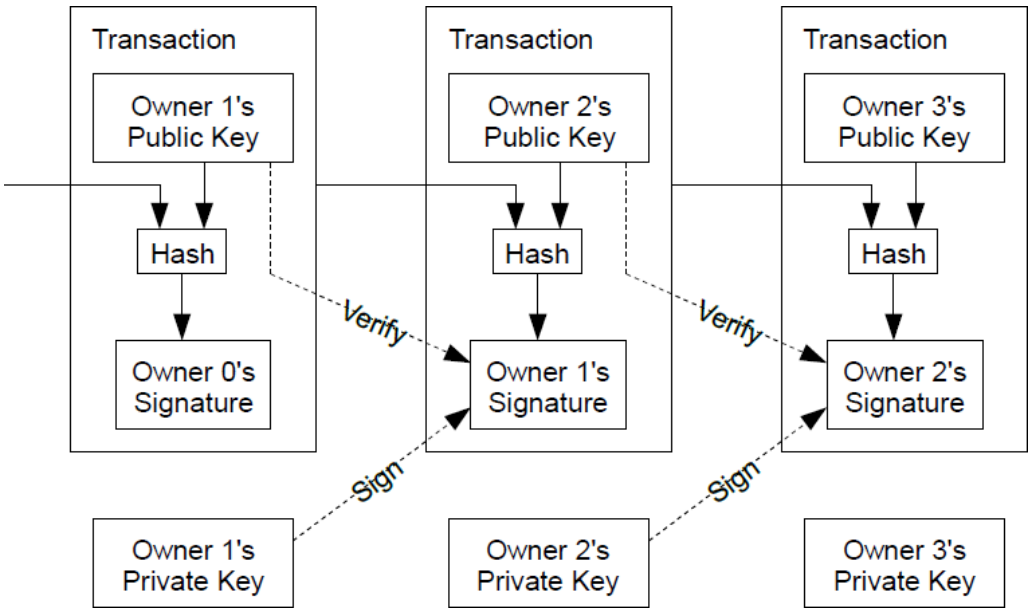

Figura 13: Shema che rappresenta il processo nelle transazioni con Bitcoin [39]

2.1.2 Timestamp

Il crittografo Stuart Haber e e il ricercatore W. Scott Stornetta nell'articolo *How to Time Stamp a Digital Document"* "[30] hanno descritto un prototipo delle blockchain su cui fanno affidamento la maggior parte delle criptovalute attuali. La soluzione a cui sono giunti è stata quella di sottoporre un documento a un algoritmo di hashing crittografico, che produce un ID univoco per ogni documento preso in considerazione da molti del settore che si focalizza sul concetto di time-stamping digitale e sulla progettazione di schemi sicuri per arrestare l'orario di creazione o modifica di documenti digitali.

Citando : *"The prospect of a world in which all text, audio, picture, and video documents are in digital form on easily modifiable media raises the issue of how to certify when a document was created or last changed. The problem is to time-stamp the data, not the medium. We propose computationally practical procedures for digital time-stamping of such documents so that it is infeasible for a user either to back-date or to forward-date his document, even with the collusion of a time-stamping service."[30]*

L'articolo si focalizza sul concetto di time-stamping digitale e sulla progettazione di schemi sicuri per attestare l'orario di creazione o modifica di documenti digitali. Si evidenzia un'importante distinzione tra il processo di firma digitale e il time-stamping: mentre nel primo caso il firmatario ha un incentivo a seguire correttamente le procedure di firma, nel time-stamping, sia

utenti malintenzionati che servizi di time-stamping potrebbero trovare vantaggioso non aderire alle procedure standard. Di conseguenza, è fondamentale concepire schemi di time-stamping che neutralizzano qualsiasi beneficio derivante da comportamenti scorretti.
L'obiettivo principale è di ridurre al minimo gli assunti necessari per garantire la sicurezza nel time-stamping digitale. L'articolo propone che l'unico presupposto ragionevole sia l'esistenza di funzioni di hash unidirezionali, un principio basilare nella crittografia basata sulla complessità.
Il testo esamina attentamente due approcci al time-stamping digitale. Il primo implica l'uso di funzioni di hash e firme digitali per collegare i time-stamp di documenti successivi e diffondere queste informazioni tra gli utenti, rendendo ardua la retrodatazione fraudolenta dei time-stamp. Il secondo metodo richiede che più membri di un pool di clienti forniscano il time-stamp per l'hash di un documento, utilizzando un generatore pseudocasuale che usa l'hash stesso come chiave. Questo secondo approccio impedisce la selezione arbitraria dei clienti che effettuano il time-stamping, offrendo una protezione aggiuntiva contro la manipolazione dei time-stamp.[30]

La soluzione proposta da Bitcoin è l'utilizzo di un server con timestamp. Questo funziona partendo con un primo hash in un blocco di elementi che vengono contrassegnati e pubblicati. Il timestamp generato con l'hash dimostra che i dati dovevano esistere in quel momento. Ogni timestamp include quello precedente nel suo hash, formando una catena, con ogni timestamp aggiuntivo che rafforza la credibilità di quelli precedenti.[39]

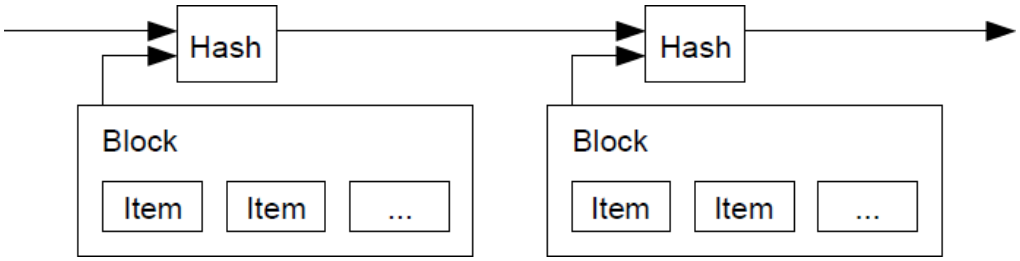

Figura 14: Shema che rappresenta l'inserimento del timestam nel blocco [39]

2.1.3 Lo stato nelle transazioni in Bitcoin

Da un punto di vista tecnico, il libro mastro di una criptovaluta come Bitcoin può essere pensato come un sistema di transizione di blocchi, dove esiste uno "stato" costituito dallo stato di proprietà di tutti i bitcoin esistenti e una "funzione di transizione di stato" che prende uno stato e una transazione e restituisce un nuovo blocco che è il risultato.

In un sistema bancario standard, ad esempio, lo stato è un bilancio, una transazione è una richiesta di spostare un quantitativo di $X da A a B, e la funzione di transizione di stato riduce il valore nel conto di A di $X e aumenta il valore in quello di B conto di $X. Se l'account di A ha meno di $X in primo luogo, la funzione di transizione di stato restituisce un errore. Si può quindi definire formalmente: APPLY(S,TX) → S' or ERROR

Figura 15: Immagine che mostra i passagi di stato durante una transazione [6]

Cos`ı come nel sistema bancario sopra definito:

APPLY({Alice: $50, Bob: $50},→
"send $20 from Alice to Bob" = {Alice: $30,Bob: $70}
OR
APPLY({Alice: $50,Bob: $50}"→
send $70 from Alice to Bob") = ERROR

Lo "stato" in Bitcoin è la raccolta di tutte le monete (transazioni in output non ancora spese o più tecnicamente "unspent transaction outputs" UTXO) che sono state coniate e non ancora spese, con ogni UTXO che ha un suo valore e un suo proprietario (definito da un indirizzo di 20 byte che è essenzialmente una chiave pubblica crittografica).
Una transazione contiene uno o più input, con ciascun input contenente un riferimento a un UTXO esistente e una firma crittografica prodotta dalla chiave privata associata all'indirizzo del proprietario, e uno o più output, con ciascun output contenente un nuovo UTXO da aggiungere per aggiornare lo stato.
La funzione di transizione di stato APPLY(S,TX) → S' può essere definita approssimativamente come segue:

Per ogni ingresso in TX:

- Se l'UTXO a cui si fa riferimento non è in S', restituisce un errore.
- Se la firma fornita non corrisponde al proprietario dell'UTXO, restituire un errore.
- Se la somma del saldo di tutti gli UTXO in ingresso è minore della somma del saldo di tutti gli UTXO in uscita, restituisce un errore.
- Ritorna infine S' con tutti gli UTXO di input rimossi e tutti gli UTXO di output aggiunti.

La prima metà del primo passaggio impedisce ai mittenti di transazioni di spendere monete che non esistono, la seconda metà impedisce ai mittenti di transazioni di spendere monete di altre persone e il secondo passaggio impone la conservazione del valore.
Per utilizzare questo sistema per il pagamento, il protocollo è il seguente. Supponiamo che Alice

voglia inviare 11,7 BTC a Bob. Per prima cosa, Alice cercherà un set di UTXO disponibili di sua proprietà per un totale di almeno 11,7 BTC.
Realisticamente Alice non riuscirà ad ottenere esattamente 11,7 BTC; supponiamo che ottenga 12 BTC dall'unione di tre UTXO 6+4+2=12. Quindi crea una transazione con quei tre input e due output. Il primo output sarà 11,7 BTC con l'indirizzo di Bob come proprietario, e il secondo output sarà il restante "cambio" di 0,3 BTC, con il proprietario Alice stessa.[6]

2.1.4 Network

I passaggi che la rete deve eseguire correttamente sono i seguenti:

1. Le nuove transazioni vengono trasmesse a tutti i nodi.
2. Ogni nodo raccoglie le nuove transazioni di un blocco.
3. Ogni nodo lavora di risolvere il proof-of-work richiesto per il suo blocco.
4. Quando un nodo trova il proof-of-work, trasmette il blocco a tutti gli altri nodi.
5. I nodi accettano il blocco solo se tutte le transazioni in esso contenute sono valide e non già spese.
6. I nodi esprimono la loro accettazione del blocco lavorando alla creazione del blocco successivo nella catena, utilizzando l'hash del blocco accettato come hash precedente.

I nodi considerano sempre la catena più lunga come quella corretta e continueranno a lavorare per estenderla. Se due nodi trasmettono contemporaneamente o a pochi istanti di distanza versioni diverse del blocco elaborato, alcuni nodi potrebbero ricevere prima l'una o l'altra. In tal caso, lavorano sul primo che hanno ricevuto, salvando l'altra versione (ramo) per utilizzarlo nel caso diventi più lungo in futuro. La riunificazione dei due rami avverrà quando sarà trovata la successiva proof-of-work e uno dei due ramo si allungherà e sarà riconosciuto come il più affidabile.
I nodi che stavano lavorando sull'altro ramo passeranno quindi a quello più lungo.
Le trasmissioni di nuove transazioni non devono necessariamente raggiungere tutti i nodi di cui è composta la rete. Finché verrà raggiunto un sufficiente numero di nodi, essi a loro volta si preoccuperanno di cooperare per il coordinamento della rete. Se un nodo non riceve un blocco, lo può richiedere quando riceverà il blocco successivo e si rendendosi conto di averne perso uno.[39]

2.2 Nakamoto Consensus

Gli algoritmi di consenso sono fondamentali per verificare l'autenticità delle piattaforme blockchain distribuite e sono il processo di costruzione di un accordo tra una rete di partecipanti senza fiducia. Creato da Satoshi Nakamoto per Bitcoin, "Nakamoto Consensus" si riferisce all'insieme di regole, in combinazione con il modello di consenso Proof of Work (PoW) nella rete, che governano il meccanismo di consenso e ne garantiscono la natura affidabile. In tal modo, Bitcoin è diventata la prima rete Peer to Peer (P2P) aperta e distribuita, Byzantine Fault Tolerant (BFT),

che utilizza una rete distribuita di nodi anonimi che sono liberi di entrare e uscire dalla rete[8].

2.2.1 Byzantine Fault Tolerant

La "Byzantine Fault Tolerant" è la capacità di una rete di computer di rimanere tollerante ai guasti con un consenso distribuito e valido nonostante informazioni imperfette o componenti guasti nella rete. Prima di Bitcoin, l'unico modo per mantenere una rete P2P BFT era l'utilizzo di un gruppo di nodi chiuso o semichiuso. Inoltre, gli algoritmi BFT tradizionali come Practical Byzantine Fault Tolerance (pBFT) utilizzano un metodo di selezione dei nodi diverso da quello attualmente utilizzato dal "Nakamoto Consensus".

Il problema"Byzantine Generals" presentato dal matematico statunitense Leslie Lamport nella pubblicazione *"The Weak Byzantine Generals Problem"*[33] esprime il raggiungimento di un accordo tra un insieme di processi, alcuni dei quali possono essere difettosi. Può essere definito precisamente come segue:
Dato un insieme di processi da 0 a n - 1 che comunicano tramite dei messaggi inviati l'uno all'altro, trovare un algoritmo mediante il quale il processo 0 possa trasmettere un valore v a tutti i processi in modo che:

1. Se il Processo 0 è non difettoso, allora ogni Processo "i" non difettoso ottiene il valore "v".
2. Se i Processi "i" e "j" sono non difettosi, allora ottengono entrambi lo stesso valore.

Si assume che i processi non difettosi seguano correttamente il loro algoritmo e che quelli difettosi siano imprevedibili. Si assume inoltre che l'assenza di un messaggio sia rilevabile, il che equivale a supporre che un processo difettoso invii qualunque messaggio, anche se potrebbe non essere quello corretto.

Questo problema è noto come "The Byzantine Generals Problem" a causa di una metafora militare, ed è centrale nell'implementazione di sistemi informatici tolleranti ai guasti.
Per spiegare meglio la metafora supponiamo di avere un gruppo di generali bizantini, ognuno con il suo esercito, che devono coordinare un attacco congiunto a una città nemica. I generali devono decidere se attaccare o ritirarsi e devono farlo in modo che:

- Se il generale principale è leale (non difettoso), allora ogni generale deve seguire l'ordine principale.
- Se due generali sono leali, devono ricevere lo stesso ordine dal generale principale.

Il problema espresso in precedenza si verifica se alcuni generali, che potrebbero essere traditori, inviano ordini contrastanti ad altri generali, mettendo in pericolo l'operazione militare.
Una soluzione completa a questo problema è possibile solo se meno di un terzo dei generali è traditore. Se almeno un terzo dei generali è traditore, non esiste un modo sicuro per raggiungere un accordo.

Il problema"Byzantine Generals" ha applicazioni pratiche nella progettazione di sistemi informatici sicuri e resistenti ai guasti, come, in un sistema di elaborazione distribuita, in cui diversi nodi devono comunicare e prendere decisioni in modo coordinato, come nei sistemi blockchain.[33]

Mantenere BFT in una rete aperta e distribuita di grandi dimensioni come quella di Bitcoin, richiede l'uso di regole che si basano sia sulla crittografia, che sul meccanismo della teoria dei giochi, al fine di creare un ambiente senza fiducia. Necessario per facilitare il consenso decentralizzato attraverso una rete di trasferimento di valore.
Nei sistemi pBFT, il modello di consenso funziona solo in piccoli gruppi di nodi chiusi (circa 50), dove c'è una notevole quantità di "overhead" (richieste accessorie) in comunicazione che impedisce a questi modelli di consenso di essere in grado di operare su larga scala. Raggiungere il consenso in un sistema, dove sono ammessi errori arbitrali prima di Bitcoin, avrebbe richiesto un sistema di votazione per aiutare a raggiungere il consenso. Per quanto riguarda le piattaforme di criptovaluta che utilizzano modelli di consenso pBFT, questo meccanismo di voto si basa su un sistema di rotazione dei nodi "leader" in un formato "Round robin scheduling" (RRS), un algoritmo di pianificazione dei lavori, che è considerato molto equo, in quanto utilizza fasce orarie assegnate a ciascun processo che viene inserito in coda.
Se il nodo "leader" agisce maliziosamente, può essere rimosso con la maggioranza degli altri nodi. Tuttavia, questo non si adatta bene in un sistema come Bitcoin, in cui il consenso sullo stato della blockchain e sulla validità di tutte le sue transazioni è distribuito a migliaia di nodi in tutto il mondo che si connettono e si disconnettono continuamente dalla rete. Inoltre, deve esserci un costo intrinseco alla partecipazione a questo sistema di consenso per scoraggiare i partecipanti dall'agire in modo dannoso.7
Quindi, affinché Bitcoin funzioni come una rete P2P Byzantine Fault Tolerant, è stato introdotto un algoritmo di consenso per il mining PoW in combinazione con un insieme specifico di regole che gestiscono il meccanismo al fine di ottenere un consenso senza fiducia in tutta la rete. Questo è stato così definito Nakamoto Consensus[8].

Riassumendo il funzionamento del "Nakamoto Consensus".

Nakamoto Consensus può essere suddiviso in quattro parti:

- Proof of Work (PoW)
- Selezione del blocco
- Scarsità
- Struttura degli incentivi

La combinazione di questi quattro componenti consente a Bitcoin di diventare una rete distribuita per il trasferimento di valore. Con un funzionamento che non richiede alcun consenso o fiducia tra le parti e che rimarrà al sicuro finché la maggior parte della potenza di calcolo gestita dai miner sarà nelle mani di "minatori onesti".

2.2.2 Proof of Work

Per implementare un server timestamp distribuito su base peer-to-peer, dovremmo utilizzare un sistema di proof-of-work simile all' Hashcash di Adam Back.
Il proof-of-work implica la scansione di un valore che, sottoposto ad hashing come ad esempio SHA-256, trova un hash che inizia con un determinato numero di zero bit. Il lavoro medio richiesto è proporzionale al numero di zero bit richiesti e può essere verificato eseguendo un singolo hash.
Per la rete di timestamp, si implementa il proof-of-work incrementando un nonce nel blocco finché non viene trovato un valore che fornisce all'hash del blocco i bit zero richiesti. Una volta che lo sforzo della CPU è stato speso per soddisfare il proof-of-work, il blocco non può essere modificato senza rifare il lavoro fatto. Quando i blocchi successivi vengono concatenati dopo il primo, per modificare un blocco precedente occorre effettuare il ricalcolo dei blocchi successivi.
Il proof-of-work risolve anche il problema di determinare la rappresentanza nel processo decisionale a maggioranza. Se la maggioranza fosse basata sul voto da parte di un indirizzo IP, potrebbe essere sovvertita da chiunque in grado di allocare molteplici indirizzi IP.

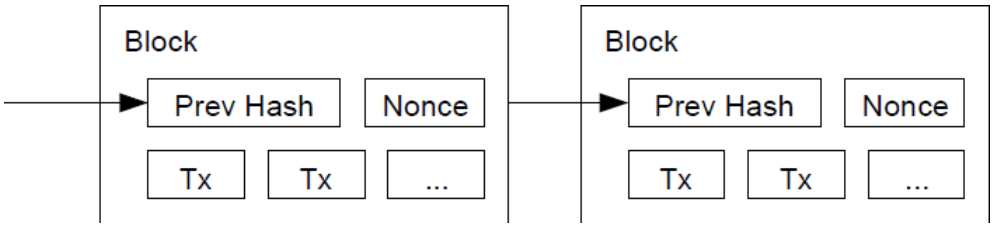

Figura 16: Eredità dell'hash tra i blocchi
[39]

L'idea alla base del Proof of Work(PoW) è stata spiegata per la prima volta nell'articolo *"Pricing via Processing or Combatting Junk Mail"*[12] di Cynthia Dwork e Moni Naor, sebbene il termine "proof of work" verrà cognato più tardi nel tempo. È stato inizialmente proposto come un modo per scoraggiare lo spam, ma oggigiorno è meglio conosciuto attraverso la sua associazione con la criptovaluta.
L'idea alla base era la seguente: se l'invio di e-mail costasse di più, anche una frazione di centesimo per messaggio, ciò potrebbe essere sufficiente a scoraggiare gli spammer. Nell'ipotesi quindi di voler addebitare a chiunque $ 0,005 per inviarti un messaggio di posta elettronica,non si vuole in realtà raccogliere i loro soldi, vuoi solo la prova che sarebbero disposti a spendere qualcosa per inviarti un'e-mail. Non si sta nemmeno tentando di bloccare i bot . Quindi, invece di chiedere un micropagamento, si potrebbe chiedere al mittente di risolvere un "puzzle", qualcosa che richiederebbe circa $ 0,005 di risorse informatiche. Se ricevi ancora troppo spam, potresti aumentare la frequenza assegnando loro un puzzle computazionalmente più difficile.
Dwork e Naor elencano diversi possibili enigmi. La chiave è trovare un puzzle che richieda un discreto sforzo per essere risolto ma che la soluzione sia facile da verificare.
Bitcoin utilizza problemi di hash per i puzzle implementati nel PoW. Le funzioni hash crittografiche sono difficili da prevedere e quindi non esistono metodologie migliori della ricerca con forza bruta se si vuole ottenere input il cui valore hash abbia uno schema specificato.
[12]

Il proof-of-work è essenzialmente la conversione di CPU ad un voto. La decisione della maggioranza è rappresentata quindi dalla catena più lunga, che ha il massimo sforzo di proof-of-work investito in essa, espresso secondo l'algoritmo di Nakamoto. Se la maggior parte della potenza della CPU è controllata da nodi trust, la catena crescerà più velocemente e supererà qualsiasi catena concorrente. Per modificare un blocco passato, un utente malintenzionato dovrebbe rifare il proof-of-work del blocco e di tutti i blocchi successivi e quindi raggiungere e superare il lavoro dei nodi trust.
Mostreremo in seguito che la probabilità che un utente malintenzionato più lento recuperi il lavoro dei nodi trust diminuisce in modo esponenziale con l'aggiunta di blocchi successivi.
Per compensare l'aumento della velocità dell'hardware e il variare dell'interesse nell'esecuzione dei nodi nel tempo, la difficoltà della prova di lavoro è determinata da una media mobile che mira a un numero medio di blocchi all'ora. Se vengono generati troppo velocemente, la difficoltà aumenta [39].

2.2.3 Selezione del Blocco

Il processo di selezione dei blocchi utilizzato dal "Nakamoto Consensus" è una delle componenti chiave che lo differenziano da altri modelli, essendo basato sul PoW. Il processo di selezione del blocco fa riferimento ad una precisa logica in cui i minatori hanno come obiettivo la ricompensa ottenuta dalla risoluzione dell'hash. Questo per procedere con l'estrazione del blocco successivo.

Nel pBFT, il nodo "leader" del blocco viene selezionato attraverso un processo di voto e sostituito tramite un algoritmo di scheduling round robin ad ogni votazione. In questo sistema non vi è mining, e per questo motivo i blocchi vengono selezionati per essere aggiunti alla catena dal leader e devono essere approvati da almeno $\frac{2}{3}$ degli altri nodi.
In Bitcoin non esiste un processo di voto per determinare il leader del blocco, invece, si utilizza un "puzzle" crittografico basato sull'incremento di un nonce nel blocco finché non viene raggiunto il valore corretto che rappresenta l'hash del blocco che soddisfa la richiesta di zero bit all'inizio del nonce (in crittografia il termine "nonce" indica un numero, generalmente casuale o pseudo-casuale).
I "miner" della rete competono tutti per risolvere questo enigma e il primo a trovare la soluzione vince la ricompensa per quel blocco. Il blocco viene quindi propagato dal miner attraverso la rete agli altri nodi che "votano implicitamente" per accettare il blocco come valido aggiungendolo a quella che fino a quel momento è considerata la catena più lunga. Un effetto risultante di questo processo è la rimozione dell'influenza di terzi su un leader perché il processo è casuale e il leader non può essere previsto.
L'unico modo per vincere la ricompensa è contribuire alla potenza di hashing della rete nella speranza di vincere, e quando non vinci, l'energia spesa diventa un costo irrecuperabile, che si aggiunge alla struttura degli incentivi del mining. Diventa quindi un costo intrinseco alla partecipazione, indipendentemente dal fatto che i blocchi vengono estratti con successo o meno[8].

2.2.4 Incentivo per la creazione dei blocchi

Per convenzione, la prima transazione in un blocco è una transazione speciale che avvia una nuova moneta di proprietà del creatore del blocco. Ciò aggiunge un incentivo per i nodi a supportare la rete e fornisce un modo per distribuire inizialmente le monete in circolazione, poiché non esiste un'autorità centrale per emettere le monete.
L'aggiunta costante di una quantità di nuove monete è analoga all'attività dei cercatori d'oro che spendono risorse per aggiungere oro in circolazione. Nel nostro caso, è il tempo di utilizzo della CPU e dell'elettricità che vengono consumati.
L'incentivo può anche essere finanziato con commissioni nelle transazioni. Il valore di un output di una transazione è superiore al suo valore di input, la differenza è la commissione della transazione in questione, che viene aggiunta al valore del blocco che conterrà la transazione generando un incentivo per i "miner".
Una volta che un numero predeterminato di monete è entrato in circolazione, l'incentivo può passare interamente alle commissioni di transazione ed essere completamente privo di inflazione a meno di speculazioni.
L'incentivo può aiutare a incoraggiare i nodi a rimanere trust. Se un attaccante malintenzionato fosse in grado di assemblare più potenza di CPU di tutti i nodi trust, dovrebbe scegliere se usarla per frodare le persone rubando i pagamenti oppure utilizzarla per generare nuove monete.
Il sistema, per come è strutturato, rende più vantaggioso giocare secondo le regole che favoriscono, con il compenso, l'introduzione di nuove monete insieme a tutte le altre già immesse nel sistema, piuttosto che elaborare il proprio sistema.[39]

2.3 Mining

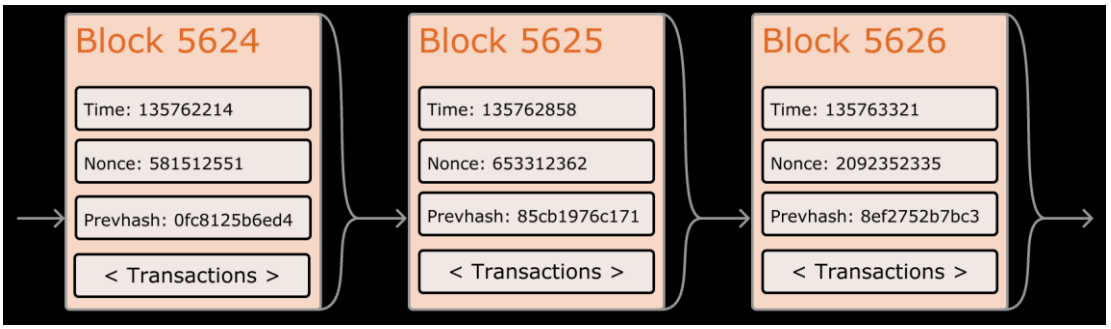

Figura 17: Immagine che rappresenta la sequenzialità nei blocchi
[6]

Ritornando allo stato delle transazioni descritto nel precedente capitolo 2.1.3.
Se avessimo accesso a un servizio centralizzato affidabile, questo sistema sarebbe banale da implementare: potrebbe semplicemente essere codificato esattamente come descritto, con l'utilizzo del disco rigido di un server centralizzato per tenere traccia dello stato.
Tuttavia, con Bitcoin si è cercato di costruire un sistema valutario decentralizzato, e quindi si

dovrebbe combinare il sistema di transazioni di stato con un sistema di consenso al fine di garantire che tutti siano d'accordo sull'ordine delle transazioni.
Il processo di consenso decentralizzato di Bitcoin richiede che i nodi della rete tentino continuamente di produrre pacchetti di transazioni chiamati "blocchi". La rete è destinata a produrre circa un blocco ogni dieci minuti, con ogni blocco contenente un timestamp, un nonce, un riferimento al hash del blocco precedente e un elenco di tutte le transazioni effettuate dal blocco precedente. Nel tempo, questo crea una "catena di blocchi" persistente, in continua crescita, che si aggiorna costantemente per rappresentare l'ultimo stato del ledger di Bitcoin.
L'algoritmo per verificare se un blocco sia valido o meno, deve eseguire il seguente paradigma:

- Verifica se il blocco precedente a cui fa riferimento esiste ed è valido.
- Verificare che il timestamp del blocco sia maggiore di quello del blocco precedente e firmato entro 2 ore dal momento di ricezione
- Verificare che il proof-of-work del blocco sia valido.
- Supponiamo che S[0] sia lo stato alla fine del blocco precedente, prendiamo questo stato come riferimento e supponiamo che tx sia la lista delle transazioni del blocco con n transazioni.
  Per ogni i in 0...n-1, impostare S[i+1] = APPLY(S[i],TX[i]). Se una qualsiasi applicazione restituisce un errore, si chiude e restituisce il valore "false".
- Quando l'algoritmo restituisce il valore "true", registra S[n] come stato finale di questo blocco. In sostanza, ogni transazione nel blocco deve fornire un messaggio di stato valido rispetto a quello che era lo stato canonico prima che la nuova transazione fosse eseguita.

Bisogna notare che lo stato non è codificato in alcun modo nel blocco, ma è soltanto un'astrazione che il nodo che la esegue convalida e ricorda, e che può essere calcolata in sicurezza per ogni blocco solo partendo dallo stato delle transazioni precedenti e applicando sequenzialmente ogni transazione in ogni blocco.
L'ordine in cui il miner include le transazioni nel blocco è importante: se ci sono due transazioni A e B in un blocco, in modo tale che B spenda un UTXO creato da A, il blocco sarà valido solo se A viene prima di B e non viceversa. L'unica condizione di validità presente nell'elenco sopra indicato, che non si trova in altri sistemi, è il requisito "proof-of-work".
La condizione esatta è che l'hash double-SHA256 di ogni blocco, trattato come un numero a 256 bit, debba essere inferiore a un target regolato dinamicamente. Lo scopo è rendere la creazione di blocchi "difficile" dal punto di vista computazionale, impedendo cos`ı ad attacchi Sybil di ricreare l'intera blockchain a loro favore. Dato che SHA256 è progettato per essere una funzione pseudo-casuale completamente imprevedibile, l'unico modo per creare un blocco valido è semplicemente tramite il metodo "brute force", incrementando ripetutamente il nonce e controllando se il nuovo hash coincide.
All'attuale target di $\sim 2^{187}$, la rete deve fare una media di $\sim 2^{69}$ tentativi prima di trovare un blocco valido; in generale, il target viene ricalibrato dalla rete ogni 2016 blocchi in modo che, in media, un nuovo blocco venga prodotto da un nodo nella rete ogni dieci minuti.
Al fine di ricompensare i miner per questo lavoro computazionale, il miner di ogni blocco ha il diritto di includere una transazione assegnandosi a fine lavoro 12,5 BTC creandoli dal nulla. Inoltre, se una o più transazioni hanno un taglio totale superiore nei propri input rispetto agli output (è giusto o è il contrario'), anche la differenza va al miner come "commissione sulle transazioni".

Per inciso, questo è anche l'unico meccanismo con cui vengono emessi nuovi BTC: lo stato genesi ovvero il primo blocco non contiene alcuna moneta. Al fine di comprendere meglio lo scopo del mining, vediamo cosa succede in caso di attacco da parte di un malintenzionato. Poiché la crittografia sottostante di Bitcoin è nota per essere sicura, l'attaccante rivolge la sua attenzione all'unica parte del sistema Bitcoin che non è protetta direttamente dalla crittografia: l'ordine delle transazioni. La strategia dell'attaccante è semplice:

1. Inviare 100 BTC ad un commerciante in cambio di un prodotto
2. Attendere la consegna del prodotto
3. Eseguire un'altra transazione inviando gli stessi 100 BTC a se stesso
4. Cercare di convincere la rete che la transazione che ha inviato a se stesso è quella che è arrivata prima.

Qualche minuto dopo aver eseguito il passaggio 1), qualche miner includerà la transazione in un blocco, ad esempio il blocco numero 270000. Dopo circa un'ora, altri cinque blocchi saranno stati aggiunti alla catena dopo quel blocco, e ciascuno di questi blocchi punterà indirettamente alla transazione, pertanto confermandola. A questo punto, il commerciante accetterà il pagamento come finalizzato e consegnerà il prodotto.
Dal momento che supponiamo che si tratti di un bene digitale, la consegna è immediata. Ora, l'attaccante crea un'altra transazione, inviando i 100 BTC a se stesso. Se l'attaccante inviasse questa transazione semplicemente a caso, la transazione non verrebbe elaborata: i miner cercherebbero di eseguire APPLY(S, X) e noterebbero che TX consuma un UTXO che non è più nello stato.
Invece l'attaccante crea una "fork" o diramazione della blockchain, iniziando a eseguire il mining di un'altra versione del blocco 270000 e puntando al blocco 269999 in quanto principale, ma con la nuova transazione al posto di quella vecchia. Poiché i dati del blocco sono diversi, la situazione richiede una ripetizione del proof-of-work. Inoltre, la nuova versione del blocco 270000 dell'attaccante ha un hash diverso, quindi i blocchi originali da 270001 a 270005 non "puntano" ad esso; cos`ı la catena originale e la nuova catena dell'attaccante sono completamente separate. La regola è che, in caso di diramazione, la blockchain più lunga è considerata quella trust, così i miner legittimi lavoreranno sulla catena 270005 mentre solo l'attaccante lavorerà sulla catena 270000. Per far diventare la propria blockchain più lunga, l'attaccante dovrebbe disporre di più potenza computazionale rispetto al resto della rete combinata, per recuperare il ritardo (da qui l'espressione "attacco del 51%").[6]

2.4 Gestione dello spazio

Una volta che l'ultima transazione di una moneta è stata superata da un numero sufficiente di blocchi, le transazioni spese prima possono essere scartate per risparmiare spazio su disco. Per facilitare ciò senza rompere l'hash del blocco, le transazioni vengono sottoposte ad hashing in un Merkle Tree, con solo la radice inclusa nell'hash del blocco.
I vecchi blocchi possono quindi essere ricompattati "tagliando" i rami dell'albero. Invece gli hash interni non devono essere archiviati.

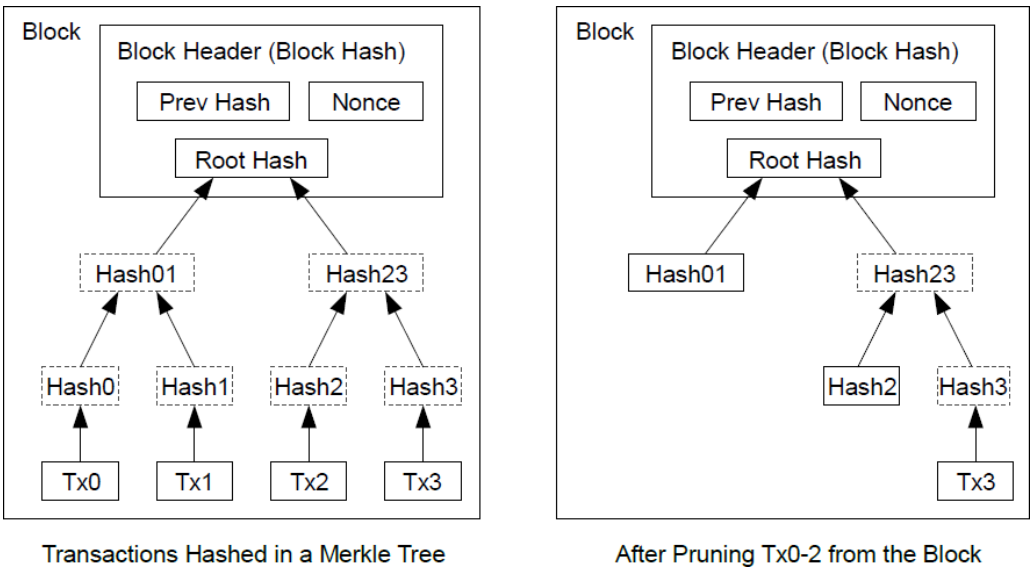

Figura 18: Grafico che rappresenta il meccanismo di selezione in un Merkle Tree [39]

La pre-visualizzazione del costo di memorizzazione del protocollo Bitcoin seguiva i seguenti calcoli: l'intestazione di blocco senza transazioni sarebbe di circa 80 byte. Se supponiamo che i blocchi vengono generati ogni 10 minuti, 80 byte * 6 * 24 * 365 = 4,2 MB all'anno.
Con i sistemi informatici in genere venduti con 2 GB di RAM a partire dal 2008 e la legge di Moore, che prevede una crescita attuale di 1,2 GB all'anno, l'archiviazione non dovrebbe essere un problema anche se le intestazioni dei blocchi devono essere conservate in memoria.[39]

2.4.1 Merkle Tree

Un'importante caratteristica di scalabilità di Bitcoin consiste nel fatto che il blocco è memorizzato in una struttura di dati a multilivello.
L' "hash" di un blocco è in realtà solo l'intestazione del blocco, un dato di circa 200 byte che contiene timestamp, nonce e hash del blocco precedente; questo partendo dall'hash della radice di una struttura dati chiamata albero di Merkle che memorizza tutte le transazioni in quel blocco. Un albero di Merkle è un tipo di albero binario, non bilanciato, composto da un insieme di nodi con un gran numero di nodi foglie nella parte inferiore, I nodi foglia che contengono i dati memorizzati più in basso. La parte centrale dell'albero è un insieme di nodi intermedi nei quali ogni nodo genera l'hash dei suoi due figli. Nella parte superiore vi è un singolo nodo radice che rappresenta l'"head" dell'albero. Lo scopo dell'albero di Merkle è permettere la consegna dei dati in un blocco frammentario: un nodo può scaricare solo l'intestazione del blocco solo dalla sua fonte. Il motivo per cui ciò funziona è che gli hash si propagano verso l'alto: se un utente malintenzionato tenta di scambiare una transazione falsa sul fondo di un albero di Merkle, questa modifica causerà una modifica del nodo sopra, che a sua volta modificherà a del il suo nodo superiore, cambiando alla fine la radice dell'albero e quindi l'hash del blocco, portando

alla registrazione del protocollo come blocco completamente diverso (quasi certamente con una proof-of-work non valida).
Il protocollo dell'albero di Merkle è probabilmente essenziale per la sostenibilità a lungo termine. Un nodo completo nella rete Bitcoin è un nodo che memorizza ed elabora per intero tutti i blocchi e occupa circa 15 GB di spazio su disco nella rete Bitcoin ad aprile 2014, e cresce di oltre un gigabyte al mese. Attualmente, questo è fattibile per alcuni computer desktop e non per gli smartphone, ma in futuro solo aziende e hobbisti saranno in grado di partecipare.
Un protocollo noto come *"simplified payment verification"* (SPV) consente l'esistenza di un'altra classe di nodi, chiamati "nodi leggeri", che scaricano le intestazioni dei blocchi, verificano il "proof-of-work" sulle intestazioni dei blocchi, quindi scaricano solo i rami associati alle transazioni che sono rilevanti per loro.
Questo permette ai nodi leggeri di determinare con una forte sicurezza lo stato di qualsiasi transazione Bitcoin, e il loro saldo corrente, scaricando solo una piccolissima porzione dell'intera blockchain.[6]

Le immagini mostrano quanto sopra spiegato.

Fondamenti di tale struttura sono stati definiti dallo stesso crittografo statunitense Ralph Merkle nella sua pubblicazione del 1979 *"A CERTIFIED DIGITAL SIGNATURE"* [35] in cui viene descritta una struttura ad albero di autenticazione. Viene discusso un protocollo di autenticazione che mira a ridurre i requisiti di archiviazione e le complessità computazionali. Questo protocollo è stato denominato "tree authentication" ed è stato sviluppato per autenticare elementi di dati in un modo che richiede uno spazio di archiviazione minimo.
La struttura ad albero di autenticazione si basa sul principio del "divide et impera", dove una funzione H(ij,x) è utilizzata per autenticare un intervallo di dati dall'elemento i all'elemento j. Questo approccio consente di autenticare in modo selettivo elementi specifici senza richiedere la memorizzazione di copie complete dei dati. L'autenticazione viene eseguita attraverso una serie di chiamate ricorsive, formando un albero binario di chiamate.
L'algoritmo di rigenerazione del percorso è stato sviluppato per ricostruire rapidamente i percorsi di autenticazione senza richiedere una grande quantità di memoria. Questa struttura ad albero di autenticazione offre una soluzione efficiente per problemi di autenticazione e firma digitale senza l'uso di crittosistemi a chiave pubblica e offre la possibilità di implementazioni efficienti sia in termini di tempo che di spazio.[35]

2.4.2 Longest Chain

È possibile verificare i pagamenti senza l'esecuzione di un nodo di rete completo. Un utente deve solo conservare una copia delle intestazioni del blocco della catena di "proof-of-work" più lunga, che può ottenere interrogando i nodi di rete fino a quando non è convinto di avere la catena più

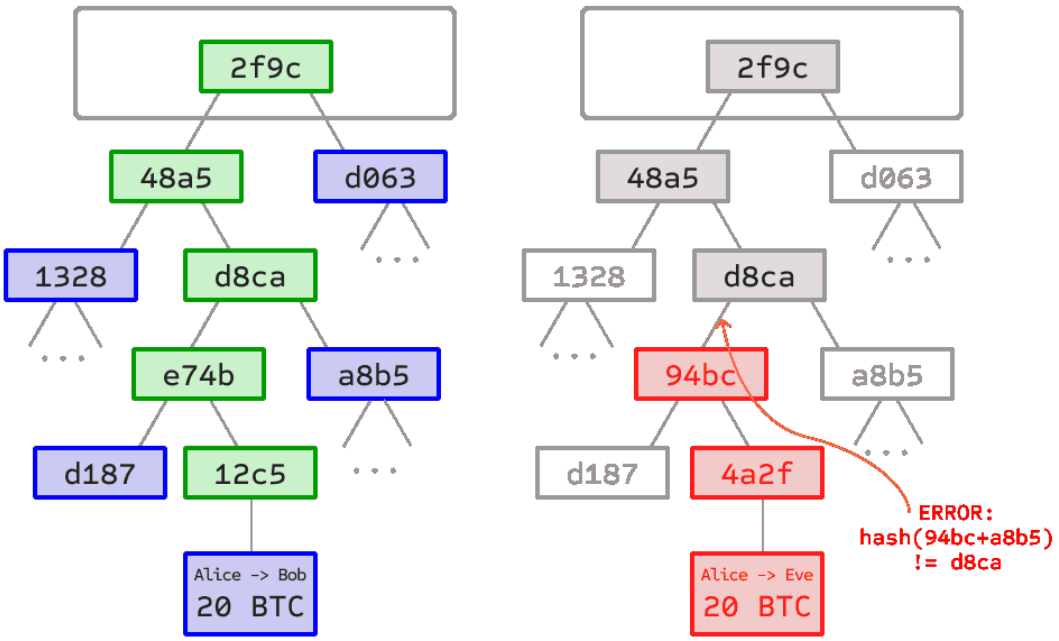

Figura 19: A sinistra: L'immagine mosta che basta aggiungere solo un piccolo numero di nodi in un albero di Merkle per fornire una prova della validità di un ramo.

Figura 20: A destra : L'immagine mosta che ogni tentativo di cambiare una parte dell'albero di Merkle causerà incoerenze ai livelli superiori della catena.
[6]

lunga, e ottenere il ramo Merkle che collega la transazione al blocco.
Un nodo quindi non può controllare la transazione da solo, ma collegando il blocco a un punto della catena, può vedere che un nodo di rete l'ha accettata e che i blocchi aggiunti dopo confermano ulteriormente l'accettazione della transazione.
Mentre i nodi di rete possono verificare le transazioni da soli, il metodo semplificato può essere ingannato dalle transazioni fabbricate da un utente malintenzionato, un eventuale attaccante in questo può continuare a sopraffare la rete.

Una strategia per proteggersi da ciò sarebbe considerare gli avvisi dai nodi di rete quando rilevano un blocco non valido, richiedendo al software dell'utente di scaricare l'intero blocco e segnalare le transazioni per confermare l'incoerenza. Le aziende che ricevono pagamenti frequenti probabilmente vorranno comunque eseguire i propri nodi per una maggiore sicurezza e una verifica più rapida.[39]

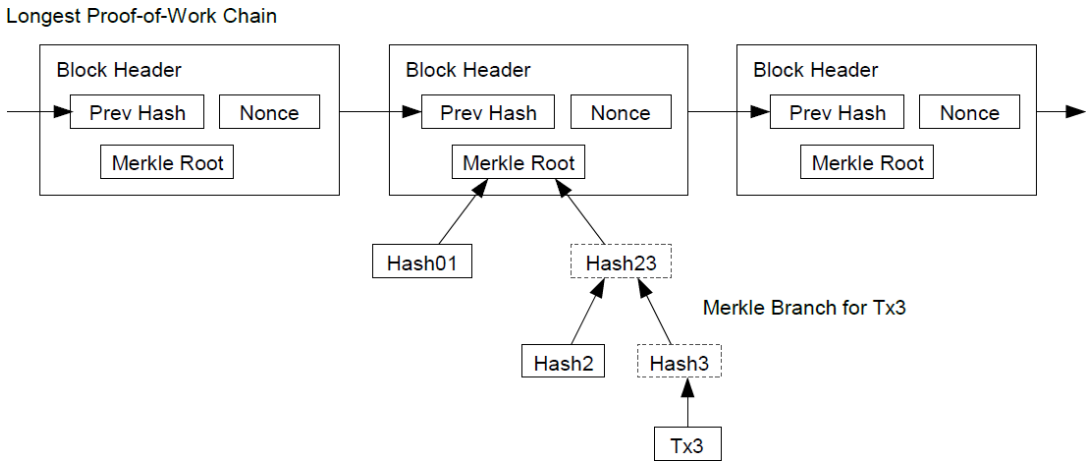

Figura 21: Immagine che mostra il significato di Longest Chain
[39]

2.4.3 Combinare e dividere il valore

Sebbene sia possibile gestire le monete individualmente, sarebbe difficile effettuare una transazione separata per ogni centesimo di ogni singolo trasferimento. Per consentire la suddivisione e

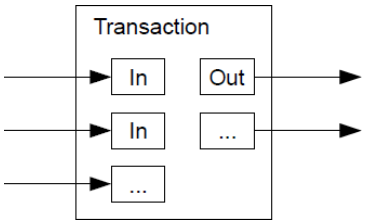

Figura 22: Immagine che schematizza I/O
[39]

la combinazione del valore, le transazioni contengono più input e output. Normalmente ci sarà un singolo input da una transazione precedente più grande o più input che combinano importi di transazioni più piccole, e al massimo due output: uno per il pagamento e uno per restituire l'eventuale resto al mittente.[39]

2.5 Riservatezza

Il modello bancario tradizionale raggiunge un livello di privacy limitando l'accesso alle informazioni alle parti coinvolte e alla terza parte di fiducia. La necessità di annunciare pubblicamente

tutte le transazioni preclude questo metodo, ma la privacy può ancora essere mantenuta interrompendo il flusso di informazioni in un altro luogo: ovvero mantenendo anonime le chiavi pubbliche. Il pubblico può vedere che qualcuno sta inviando un importo a qualcun altro, ma senza alcuna informazione che colleghi la transazione a qualcuno.
Come protezione aggiuntiva, dovrebbe essere utilizzata una nuova coppia di chiavi per ogni transazione per evitare che vengano collegate a un proprietario comune.
Alcuni collegamenti sono ancora inevitabili con le transazioni multi input, che rivelano necessariamente che i loro input fossero di proprietà della stessa persona. Il rischio è che se viene rivelato il proprietario di una chiave, il collegamento potrebbe rivelare altre transazioni che appartenevano alla stessa persona.[39]

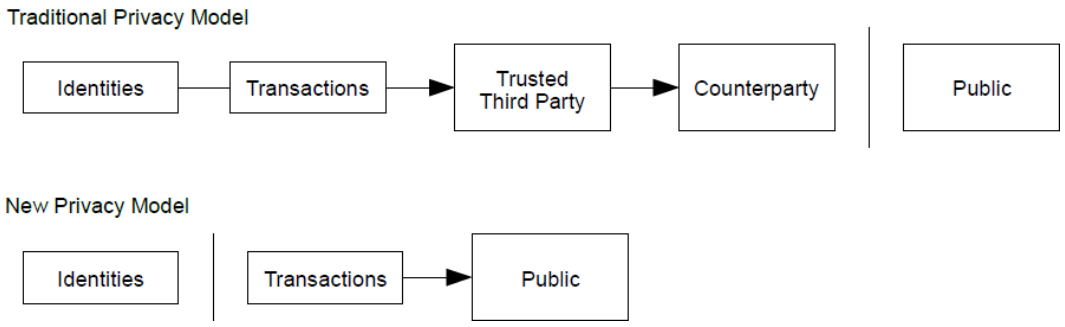

Figura 23: Immagine mostra il modello della Privacy
[39]

2.5.1 Dimostrazione

Consideriamo lo scenario di un utente malintenzionato che tenta di generare una catena alternativa più veloce della catena trust. Anche se ciò venisse realizzato, non espone il sistema a modifiche arbitrarie, come la creazione di valore dal nulla o al prelievo di denaro che non è mai appartenuto all'attaccante.
I nodi non accetteranno una transazione non valida come pagamento e i nodi trust non accetteranno mai un blocco che li contenga. Un utente malintenzionato può solo provare a modificare una delle sue transazioni per riprendersi i soldi che ha speso di recente.
La corsa tra la catena trust e una catena attaccante può essere definita come una passeggiata casuale binomiale. L'evento di successo è che la catena trust venga estesa di un blocco, aumentando il suo vantaggio di +1, e l'evento di fallimento invece è che la catena dell'attaccante venga estesa di un blocco, riducendo il divario di -1.
La probabilità che un utente malintenzionato recuperi da un dato deficit è analoga a un problema di Gambler's Ruin. Supponiamo che un giocatore d'azzardo con credito illimitato inizi con un deficit e giochi potenzialmente un numero infinito di prove per cercare di raggiungere il pareggio. Possiamo calcolare la probabilità che non raggiunga mai il pareggio, coincide con la probabilità per un attaccante di non raggiungere mai la catena trust, come segue:

p = probabilità che un nodo onesto trovi il blocco successivo
q = probabilità che l'attaccante trovi il blocco successivo

qz = probabilità che l'attaccante raggiunga mai z blocchi dietro

$$\begin{cases} 1 & \text{if } p \leq q \\ (q/p)^z & \text{if } p > q \end{cases}$$

[39]
Data la nostra ipotesi che p ¿ q, la probabilità diminuisce in modo esponenziale all'aumentare del numero di blocchi che l'attaccante deve recuperare.
Con le probabilità contro di lui, se non fa un fortunato salto in avanti all'inizio, le sue possibilità diventano irrisorie man mano che rimane indietro.
Consideriamo ora quanto tempo deve attendere il destinatario di una nuova transazione prima di essere sufficientemente certo che il mittente non possa modificare la transazione. Partiamo dal presupposto che il mittente è un utente malintenzionato che vuole far credere al destinatario di averlo pagato per un po' di tempo, per modificare lo stato. Il destinatario verrà avvisato quando ciò accadrà, ma il mittente spera che sia troppo tardi.
Il destinatario genera una nuova coppia di chiavi e consegna la chiave pubblica al mittente poco prima della firma. Ciò impedisce al mittente di preparare una catena di blocchi in anticipo lavorando continuamente fino a quando non ha la fortuna di andare abbastanza avanti, quindi eseguendo la transazione in quel momento. Una volta inviata la transazione, il mittente disonesto inizia a lavorare in segreto su una catena parallela contenente una versione alternativa della sua transazione. Il destinatario attende fino a quando la transazione non è stata aggiunta a un blocco e z blocchi sono stati collegati dopo di esso. Non conosce l'esatta quantità di progressi compiuti dall'attaccante, ma supponendo che i blocchi onesti abbiano impiegato il tempo medio previsto per blocco, il potenziale progresso dell'attaccante sarà una distribuzione di Poisson con valore atteso:

$$\lambda = z\frac{q}{p}$$

Per ottenere la probabilità che l'attaccante possa ancora recuperare ora, moltiplichiamo la densità di Poisson per ogni quantità di progresso che avrebbe potuto fare per la probabilità che potesse recuperare da quel punto:

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \begin{cases} (q/p^{z-k}) & \text{if } k \leq z \\ 1 & \text{if } k > z \end{cases}$$

[39] Riorganizzazione dei calcoli:

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!}(1 - (q/p)^{(z-k)})$$

[39] Conversione in codice C:

```
#include <math.h>

double AttackerSuccessProbability(double q, int z)
{
    double p = 1.0 - q;
    double lambda = z * (q / p);
    double sum = 1.0;
    int i, k;
    for (k = 0; k <= z; k++)
```

```
    {
        double poisson = exp(-lambda);
        for (i = 1; i <= k; i++)
            poisson *= lambda / i;
        sum -= poisson * (1 - pow(q / p, z - k));
    }
    return sum;
}
```

Eseguendo alcuni risultati, possiamo vedere la probabilità diminuire esponenzialmente con z.

q=0.1	**q=0.3**
z=0 P=1.0000000	z=0 P=1.0000000
z=1 P=0.2045873	z=5 P=0.1773523
z=2 P=0.0509779	z=10 P=0.0416605
z=3 P=0.0131722	z=15 P=0.0101008
z=4 P=0.0034552	z=20 P=0.0024804
z=5 P=0.0009137	z=25 P=0.0006132
z=6 P=0.0002428	z=30 P=0.0001522
z=7 P=0.0000647	z=35 P=0.0000379
z=8 P=0.0000173	z=40 P=0.0000095
z=9 P=0.0000046	z=45 P=0.0000024
z=10 P=0.0000012	z=50 P=0.0000006

Risolvendo per P inferiore allo 0,1%

P < 0.001

$q = 0.10 z = 5$

$q = 0.15 z = 8$

$q = 0.20 z = 11$

$q = 0.25 z = 15$

$q = 0.30 z = 24$

$q = 0.35 z = 41$

$q = 0.40 z = 89$

$q = 0.45 z = 340$

[39]

2.6 Applicazioni alternative alla blockchain di Bitcoin

L'idea di prendere il concetto di blockchain e applicarlo ad altri campi ha una lunga storia. Nel 2005, Nick Szabo ha presentato il concetto di "titoli di proprietà protetti con autorità del proprietario", un documento che descrive come "i nuovi progressi nella tecnologia dei database replicati" consentiranno a un sistema basato su blockchain l'archiviazione di un registro di chi possiede qualche terreno, creando un quadro elaborato che includa concetti come fattoria, possesso avverso e tassa fondiaria georgiana. Tuttavia ai tempi non esisteva purtroppo alcun efficace sistema di database replicato e quindi il protocollo non è mai stato realizzato. Dopo il 2009, però, una volta sviluppato il consenso decentralizzato di Bitcoin, un certo numero di applicazioni alternative ha rapidamente iniziato a emergere.

- **Namecoin:**[40] creato nel 2010, Namecoin si può descrivere come un database di registrazione dei nomi decentralizzato. Nei protocolli decentralizzati come Tor, Bitcoin e BitMessage, è necessario un modo per identificare i conti, così che le altre persone possano interagirvi. Ma in tutte le soluzioni esistenti, il solo tipo di identificativo disponibile è un hash pseudo-casuale come 1LW79wp5ZBqaHW1jL5TCiBCrhQYtHagUWy. Idealmente, si dovrebbe poter avere un conto con un nome come "george". Tuttavia, il problema è che se una persona può creare un conto denominato "george", allora qualcun altro può usare lo stesso processo per registrare "george" per sé stesso, oltre a impersonarlo. La soluzione è un paradigma del "primo depositante", dove la prima registrazione riesce e la seconda no, un problema perfettamente adatto al protocollo di consenso di Bitcoin.
  Namecoin è l'implementazione meno recente e più riuscita di un sistema di registrazione dei nomi che usa tale idea.

- **Colored coins:**[44] lo scopo del protocollo colored coins è consentire alle persone di creare le proprie valute digitali o, nell'importante caso di una valuta con un'unità, token digitali, sulla blockchain di Bitcoin. Nel protocollo colored coins, qualcuno "emette" una nuova valuta assegnando pubblicamente un colore a un UTXO di Bitcoin specifico e il protocollo definisce ricorsivamente il colore dell'altro UTXO affinché sia lo stesso degli input spesi dalla transazione che la sta creando (si applicano alcune regole speciali in caso di input di colore misto). Questo consente agli utenti di mantenere i portafogli contenenti solo UTXO di un colore specifico e di inviarli, proprio come i Bitcoin regolari, retrocedendo per la blockchain per determinare il colore di ogni UTXO che ricevono.

- **Metacoins:**[6] l'idea legata al metacoin è di avere un protocollo che risieda su Bitcoin, che utilizzi le transazioni di Bitcoin per memorizzare le transazioni di metacoin, ma avendo una funzione di transizione tra stati differenti, la 'APPLY'. Poiché il protocollo metacoin non può impedire la comparsa di transazioni di metacoin non valide nella blockchain di Bitcoin, viene aggiunta la regola secondo cui, se APPLY'(S,TX) restituisce un errore, il protocollo va al valore predefinito APPLY'(S,TX) = S. Questo fornisce un facile meccanismo per creare un protocollo della criptovaluta arbitrario, potenzialmente con funzionalità avanzate non implementabili nello stesso Bitcoin, ma con un costo di sviluppo molto ridotto poiché le complessità di mining e di rete sono già gestite dal protocollo di Bitcoin.
  I metacoin sono stati usati per implementare delle classi di contratti finanziari, di registrazione dei nomi e di scambio decentralizzato.

Quindi, in generale, ci sono due approcci per creare un protocollo di consenso: creare una rete indipendente e creare un protocollo basato su Bitcoin.

Il primo approccio, pur essendo ragionevolmente riuscito nel caso di applicazioni come Namecoin, è difficile da attuare, dovendo ogni implementazione individuale avviare una blockchain indipendente, oltre a creare e testare tutta la transizione tra stati necessaria e il codice di networking. Inoltre, si prevede che l'insieme di applicazioni per la tecnologia decentralizzata di consenso seguirà una distribuzione della legge di potenza, in cui la stragrande maggioranza delle applicazioni sarebbe troppo piccola per garantire una propria blockchain. Inoltre è evidente che esistono grandi classi di applicazioni decentralizzate, in particolare organizzazioni autonome decentrate, che hanno bisogno di interagire tra loro
L'approccio basato su Bitcoin, d'altra parte, ha il difetto che non eredita le funzionalità semplificate di verifica dei pagamenti di Bitcoin. SPV funziona per Bitcoin perché può utilizzare la profondità della blockchain come delega di validità. A un certo punto, quando i predecessori di una transazione sono abbastanza obsoleti, si può affermare con sicurezza che erano legittimamente parte dello stato.
Meta-protocolli basati sulla blockchain, invece, non possono obbligare la blockchain a non includere le transazioni che non sono valide nel contesto dei propri protocolli. Quindi, un'implementazione completamente sicura del meta-protocollo SPV dovrebbe eseguire una scansione all'indietro fino all'inizio della blockchain Bitcoin per stabilire se alcune transazioni sono valide. Attualmente, tutte le implementazioni leggere dei meta-protocolli basati su Bitcoin fanno riferimento a un server attendibile per fornire i dati, probabilmente un risultato poco ottimale, soprattutto se uno degli scopi principali di una criptovaluta è quello di eliminare la necessità di attendibilità.

2.7 Lo scripting implementato in Bitcoin

Anche senza alcuna estensione, il protocollo di Bitcoin facilità in realtà una versione debole di un concetto di "contratti intelligenti". UTXO in Bitcoin può essere posseduto non solo da una chiave pubblica, ma anche da uno script più complesso, espresso in un semplice linguaggio di programmazione basato su stack. In questo paradigma, una transazione che spende quell'UTXO deve fornire dati che soddisfano lo script. In realtà, anche il meccanismo base di proprietà della chiave pubblica è implementato tramite uno script che accetta una firma a curva ellittica come input, la verifica in base alla transazione e all'indirizzo che possiede l'UTXO, e restituisce 1 se dà esito positivo, altrimenti 0. Esistono altri script più complicati per altri casi d'uso.
Ad esempio, si può creare uno script che richieda le firme da due di tre chiavi private per validare ("multifirma"), una configurazione utile per i conti aziendali, i conti di risparmi sicuri e alcune situazioni di deposito a garanzia del commerciante.
Gli script possono anche essere utilizzati per pagare ricompense per soluzioni a problemi di calcolo. e per consentire lo scambio decentralizzato tra criptovalute diverse.
Il linguaggio di scripting implementato in Bitcoin ha però diverse importanti limitazioni:

- **Mancanza di completezza di Turing:** serve a indicare che, sebbene esista una grande sotto serie di calcolo supportata dal linguaggio di scripting di Bitcoin, questo non ne supporta i principali. La categoria principale che manca sono i cicli. Questa scelta è stata fatta per evitare i cicli infiniti durante la verifica delle transazioni. Teoricamente, è un ostacolo insormontabile per i programmatori di script, poiché ogni ciclo è simulabile semplicemente ripetendo il codice sottostante molte volte con un'istruzione if, ma ciò porta a script molto inefficienti a livello di spazio. Ad esempio, implementare un algoritmo di firma della

curva ellittica alternativo richiederebbe 256 turni di moltiplicazione ripetuti, tutti inclusi individualmente nel codice.

- **Mancata rappresentazione del valore:** non c'è modo per uno script UTXO di fornire un controllo finale sull'importo prelevabile in controvaluta fiat (riconosciuta da uno stato).
- **Mancanza di stato:** un UTXO può essere speso o non speso. Non vi è alcuna possibilità di avere contratti multifase o script con uno stato interno diverso. Cos`ı si rende difficile creare contratti di opzioni multifase, piattaforme di scambio decentralizzate o protocolli d'impegno crittografico a due fasi (necessari per ricompense computazionali sicure). Significa anche che UTXO può essere usato solo per costruire semplici contratti irripetibili e non dei contratti più complessi "di stato", quali organizzazioni decentralizzate, e rende difficile da implementare i meta protocolli. Stato binario abbinato a mancanza di valore significa inoltre che un'altra applicazione importante, i limiti sui prelievi, è possibile.
- **Blockchain invisibile** per gli UTXO i dati blockchain quali il nonce, la marca temporale e l'hash blocco precedente sono invisibili. Questo limita severamente le applicazioni dei giochi d'azzardo e alcune altre categorie, in quanto priva la lingua dello script di fonti di casualità potenzialmente utili.

Quindi abbiamo tre approcci alla creazione di applicazioni per le criptovalute: creare una nuova blockchain, usare lo scripting basato su Bitcoin e creare un framework basato su Bitcoin.
Creare una nuova blockchain offre libertà illimitata nella creazione di un set di funzionalità, ma a scapito del tempo di sviluppo, del lavoro necessario per partire e della sicurezza.
L'utilizzo di script è facile da implementare e standardizzare, ma è molto limitato in termini di capacità e i meta-protocolli, sebbene semplici, peccano in termini di scalabilità.
Con Ethereum, si vuole creare un framework alternativo che fornisce vantaggi anche maggiori in termini di facilità di sviluppo, proprietà di light client più complete, e che consente alle applicazioni allo stesso tempo di condividere un ambiente economico e la sicurezza della blockchain.[6]

Capitolo 3

Ethereum

In questo capitolo, introdurremo la tecnologia alla base del progetto, Ethereum. Esploreremo alcuni dei passi chiave che hanno definito il suo sviluppo, esamineremo la sua struttura e le differenze rispetto a Bitcoin, e forniremo un'approfondita descrizione del funzionamento delle transazioni in Ethereum, compreso l'utilizzo del "Gas". Infine, concluderemo con un breve accenno al PoS (Proof of Stake), che rappresenta la prospettiva futura di questa tecnologia.

3.1 La nascita di Ethereum

Tutte le grandi innovazioni risolvono problemi reali e Ethereum non fa eccezione. Ethereum è stato pensato in un momento in cui i ricercatori riconoscevano il potere del modello Bitcoin e stavano cercando di andare oltre alle applicazioni di criptovaluta.
Gli sviluppatori, tuttavia, riconoscevano i limiti di implementare una nuova struttura basata su Bitcoin. L'insieme limitato dei tipi di transazione e dati, e le dimensioni dell'archiviazione dei dati sembravano limitare le applicazioni che potevano essere eseguite direttamente su Bitcoin. Qualsiasi altra attività necessita di ulteriori livelli off-chain, e questo negava molti dei vantaggi dell'utilizzo di una blockchain pubblica. Per i progetti che avevano bisogno di più libertà e flessibilità rimanendo on-chain, una nuova blockchain era l'unica opzione.
Verso la fine del 2013, Vitalik Buterin, giovane programmatore e appassionato di Bitcoin, ha iniziato a pensare di estendere ulteriormente le capacità di Bitcoin e Mastercoin [51] (un protocollo di sovrapposizione che ha esteso Bitcoin per offrire rudimentali "smart contract").
Nell'ottobre di quell'anno, Vitalik propose un approccio più generalizzato al Mastercoin team, che consentiva contratti flessibili e scriptabili (ma non Turing-complete), sostituendo la lingua contrattuale specializzata di Mastercoin.
Nel dicembre 2013, Vitalik ha iniziato a condividere un white paper che delineava l'idea alla base Ethereum: una blockchain generica Turing-complete..[6]

A partire da dicembre 2013, Vitalik e Gavin (cofondatore di Ethereum e entrato successivamente nel progetto) hanno perfezionato e sviluppato l'idea costruendo il livello di protocollo che è diventato Ethereum.
I fondatori di Ethereum stavano pensando a una blockchain senza uno scopo specifico, che avrebbe supportato un'ampia varietà di applicazioni essendo programmabile. L'idea che vi era alla base era l'utilizzo di una blockchain generica come Ethereum, in cui uno sviluppatore potesse programmare applicazioni senza dover implementare i meccanismi sottostanti di reti peer-to-peer, blockchain, algoritmi di consenso, ecc.

Figura 24: Immagine selezionata dal Libro OUT OF THE ETHER
[34]

La piattaforma Ethereum è stata progettata per astrarre questi dettagli e fornire un determinismo e un ambiente di programmazione sicuro per applicazioni blockchain decentralizzate. Lo sviluppo di Ethereum è stato pianificato in quattro fasi distinte, con importanti cambiamenti che si verificano in ogni fase. Una fase può includere sotto rilasci, noti come "hard fork", che ne modificano la funzionalità in un modo non compatibile con le versioni precedenti.
Le quattro principali fasi di sviluppo sono chiamate in codice *Frontier, Homestead, Metropolis, e Serenity*. Gli hard fork intermedi che si sono verificati o sono stati pianificati, fino ad oggi sono caratterizzati dai seguenti acronimi: *Ice Age, DAO, Tangerine Whistle, Spurious Dragon, Byzantium, e Constantinople*.

Di seguito un elenco che mostra una sequenza temporale, che è "datata" dal numero di blocco:

Blocco #0

Frontier: la fase iniziale di Ethereum, che dura dal 30 luglio 2015 a marzo 2016.

Blocco #200.000

Ice Age: un hard fork per introdurre un aumento esponenziale della difficoltà, per motivare una transizione a PoS (Proof-of-staking) quando sarà pronto.

Blocco #1.150.000

Homestead: la seconda fase di Ethereum, lanciata a marzo 2016.

Blocco #1.192.000

DAO: un hard fork che rimborsava le vittime del contratto DAO violato e ha causato la divisione di Ethereum ed Ethereum Classic in due sistemi concorrenti.

Blocco #2.463.000

Tangerine Whistle: Un hard fork per cambiare il calcolo del gas per certi tipi di I/O e per cancellare lo stato accumulato da un denial-of-service (DoS) che sfruttava il basso costo del gas di tali operazioni.

Blocco #2.675.000

Spurious Dragon: un hard fork per affrontare più vettori di attacco DoS e un'altra bonifica dello stato. Inoltre, un meccanismo di protezione dagli attacchi di riproduzione.

Blocco #4.370.000

Metropolis Byzantium—Metropolis: è la terza fase di Ethereum, lanciato nell'ottobre 2017. Byzantium è il primo di due hard fork previsti per Metropolis.
Dopo Byzantium, è previsto un ulteriore hard fork per Metropolis: Constantinople. Metropolis sarà seguita dalla fase finale di ristrutturazione di Ethereum, con il nome in codice Serenity.[1]

3.1.1 Una Blockchain General-Purpose

La blockchain originale, ovvero la blockchain di Bitcoin, tiene traccia dello stato delle unità di cui è composta e delle loro proprietà. Si può pensare a Bitcoin come ad una macchina a stato di consenso distribuito, in cui le transazioni causano una transizione di stato globale, alterando la proprietà delle monete. Le transizioni di stato sono vincolate da delle regole di consenso, permettendo a tutti i partecipanti di convergere su uno stato comune nel sistema, dopo l'estrazione di diversi blocchi.
Ethereum è anch'essa una macchina a stati distribuita. Ma invece di monitorare solo lo stato di proprietà della valuta, Ethereum tiene traccia di transizioni in uno stato più generico, ovvero può contenere qualsiasi dato esprimibile come tupla chiave-valore. La combinazione chiave-valore permette l'archivio di dati a contenuto arbitrario, ciascuno referenziato ad una chiave; ad esempio, il valore "1984" a cui fa riferimento la chiave "Titolo del libro di George Orwell".

In un certo senso, questo meccanismo fa riferimento al modello di memorizzazione dei dati nella memoria ad accesso casuale (RAM) utilizzato nella maggior parte dei computer general-purpose. Ethereum utilizza la blockchain per tenere traccia di come cambia questa memoria col tempo. Come un computer a programma general-purpose, Ethereum può caricare il codice nella sua macchina a stati ed eseguire quel codice, memorizzando i cambiamenti di stato risultanti nella sua blockchain. Due sono le differenze fondamentali, rispetto alla maggior parte dei computer general-purpose: i cambiamenti di stato di Ethereum sono governati dalle regole del consenso, e lo stato della memoria è distribuito globalmente.[1]

3.2 I Componenti di Ethereum

I componenti di cui è composta Ethereum sono così rappresentati:

- **Rete P2P** Ethereum è indirizzabile sulla porta TCP 30303, ed esegue un protocollo chiamato DEVp2p[32].
- **Consensus rules** Le regole di consenso di Ethereum sono definite nella specifica che fa riferimento al Yellow Paper prodotto da Vitalik Buterin[6].
- **Transazioni** Le transazioni di Ethereum sono messaggi di rete che includono: mittente, destinatario, valore e payload dei dati.
- **Macchina stati** Le transizioni di stato di Ethereum vengono elaborate dalla Ethereum Virtual Machine (EVM), una macchina virtuale basata su stack che esegue bytecode (Istruzione machine-language). I programmi EVM, chiamati "Smart Contract", sono scritti in linguaggio ad alto livello (ad esempio, Solidity) e compilato in bytecode per la compilazione sull'EVM.
- **Strutture dati** Lo stato di Ethereum è memorizzato localmente, su ogni nodo, come in un database che contiene le transazioni e lo stato del sistema tradotto in un hash serializzato collocato in una struttura dati chiamata Merkle Patricia Tree2.4.1.
- **Algoritmo di consenso** Ethereum utilizza il modello di consenso di Bitcoin, Nakamoto Consensus, che utilizza blocchi sequenziali a firma singola, pesati in importanza in base alla PoW(Proof-of-work) per determinare la catena più lunga e quindi lo stato più affidabile. Tuttavia, i piani futuri per Ethereum prevedono di passare a un sistema di validazione PoS(Proof-of-stake)2.2.
- **Sicurezza economica** Ethereum attualmente utilizza un algoritmo PoW chiamato Ethash[38].

3.2.1 Ethereum Turing completo

Il termine si riferisce al matematico inglese Alan Turing, che è considerato uno dei padri fondatori dell'informatica. Nel 1936 creò il modello matematico di un computer costituito da una macchina a stati che manipola i simboli leggendoli e scrivendoli in una memoria sequenziale (simile a

un nastro di lunghezza infinita). Con questo costrutto, Turing ha continuato a fornire una base matematica per rispondere a domande sulla compatibilità universale, trattando tutti i problemi come se fossero risolvibili.
Ha dimostrato che ci sono classi di problemi che non sono calcolabili. Nello specifico, lui dimostrò che il problema dell'arresto, ovvero non può essere scritto un programma che abbia la capacità di decidere se una qualsiasi macchina a stati si arresti o continui a procedere all'infinito, non è risolvibile.
Alan Turing ha inoltre definito un sistema matematico, chiamandolo Turing completo , ovvero un sistema che può essere utilizzato per simulare qualsiasi macchina di Turing. Tale sistema è chiamato macchina universale di Turing (UTM).

La capacità di Ethereum di eseguire un programma memorizzato in una macchina a stati chiamata Ethereum Virtual Machine, mentre legge e scrive i dati nella memoria, la rende un sistema Turing-complete e quindi un UTM (Macchina di Turing universale). Ethereum può calcolare qualsiasi algoritmo calcolabile da qualsiasi macchina di Turing.
L'innovazione rivoluzionaria di Ethereum è quella di combinare il calcolo di una generica architettura per computer con la memorizzazione di un programma su di una blockchain decentralizzata, quindi creare un computer mondiale distribuito a stato singolo (singleton), con l'unica limitazione di produrre uno stato comune che è protetto dalle regole di consenso[1].

Un sistema Turing completo è composto nel seguente modo:

- un alfabeto di simboli A
- 4 stati di cui uno iniziale(s0) e alcuni finali(F)
- 6 funzioni di transizione $\delta : S \times A \rightarrow S \times A \times \{0, s, \}$ con uno stato di definizione lunga solo 22 istruzioni.

3.2.2 Implicazioni della completezza di Turing

Turing ha dimostrato che non è possibile prevedere se un programma terminerà simulando su di un computer. In termini semplici, non possiamo prevedere il percorso di un programma senza eseguirlo. I sistemi Turing completi possono funzionare in "loop infinito"(termine usato per descrivere un programma che non termina). Questo può avvenire anche non intenzionalmente, a causa di complesse interazioni e condizioni tra l'avvio e la terminazione del codice. In Ethereum può essere espresso come la necessità da parte di ogni partecipante alla rete di convalidare ogni transazione, eseguendo tutti gli smart contract che può richiamare.
Ma come ha dimostrato Turing, Ethereum non può prevedere se uno smart contract terminerà o meno, oppure per quanto tempo rimarrà in esecuzione, senza eseguirlo effettivamente. Potrebbe ad esempio verificarsi, per errore o per mano di un attaccante, che uno smart contract tenti di convalidare un nodo all'infinito, Generando un Flooding o un DoS (Deniel of Service), ovvero l'esaurimento delle risorse in un sistema informatico in modo che non sia più in grado di erogare un servizio ai client richiedenti.
Gli sviluppatori di Ethereum si sono posti cos`ı il problema di limitare le risorse utilizzate da uno smart contract. Per rispondere a questa esigenza, Ethereum introduce un meccanismo di

misurazione chiamato gas. Nel momento in cui l'EVM (Ethereum Virtual Machine) esegue uno smart contract, tiene conto attentamente di ogni istruzione (operazioni, accesso ai dati, ecc.). Ogni istruzione ha un costo predeterminato in unità di gas.
Quando una transazione attiva l'esecuzione di uno smart contract, deve includere una quantità di gas che stabilisce il limite massimo che può essere consumato. L'EVM terminerà l'esecuzione se la quantità di gas consumata dallo smart contract dovesse eccedere il gas disponibile per la transazione.

Ricapitolando il gas è il meccanismo che Ethereum utilizza per consentire programmi Turing-complete, limitando il consumo che ogni risorsa può consumare. Per permettere questo tipo di operazioni si presenta il problema di dover pagare questo gas ed è qui che trova utilizzo ether (ETH), la criptovaluta integrata nel protocollo di Ethereum. Il quantitativo di Ether, richiesto per coprire il consumo di gas, deve essere inviato insieme alla transazione e deve essere espressamente destinato per l'acquisto del gas necessario all'operazione [1].

L'obiettivo di Ethereum è quello di creare un protocollo alternativo per sviluppare applicazioni decentralizzate, fornendo una serie di diversi compromessi che potranno essere utilizzati da una vasta classe di applicazioni decentralizzate, con enfasi particolare sui tempi di sviluppo, sulla sicurezza e sulle interazioni tra applicativi diversi.
Ethereum, per permettere queste agevolazioni ha creato implementato una blockchain con un linguaggio di programmazione completo e integrato, consentendo a chiunque di scrivere smart contract e applicazioni decentralizzate, in cui gli sviluppatori possono creare le proprie regole per manipolare le proprietà, i formati di transazione e le funzioni di transizione di stato. Una versione base di Namecoin [40] può essere scritta in due righe di codice, e, altri protocolli, come token (valute) e sistemi di reputazione, sono costruibili in meno di venti.

Gli smart contract sono "scatole" crittografiche contenenti valore, manovrabili esclusivamente alla soddisfazione di certe condizioni, che possono essere descritte per ciascuna piattaforma con una potenza maggiore di quella offerta dallo scripting di Bitcoin, grazie alla completezza di Turing[6].

3.3 L'implementazione di Ethereum

Ethereum vuole implementare un protocollo alternativo per sviluppare applicazioni decentralizzate attraverso una serie di diversi compromessi, con enfasi particolare sui tempi di sviluppo, sulla sicurezza e sulle interazioni tra applicativi diversi. Ethereum ha implementato una blockchain con un linguaggio di programmazione completo e integrato, consentendo a chiunque di scrivere smart contract e applicazioni decentralizzate, nei quali gli sviluppatori possono creare proprie regole, propri formati di transazione e funzioni di transizione di stato. Con la tecnologia messa a disposizione da Ethereum è possibile scrivere Namecoin[40] in due righe di codice, e altri protocolli, come valute e sistemi di reputazione, sono costruibili in meno di venti. Gli smart contract, sono "scatole" crittografiche che contengono valore, che si attivano solo a determinate condizioni e che possono essere descritte su ciascuna piattaforma, con una potenza maggiore di quella offerta dallo scripting di Bitcoin2.1.3, grazie alla completezza di Turing.[1]

3.3.1 Gli Account

In Ethereum lo stato è composto da oggetti detti ”account”, ognuno avente un indirizzo da 20 byte, e da transizioni di stato, ovvero trasferimenti diretti di valore e informazioni tra gli account. Un account di Ethereum contiene quattro campi:

- **Il nonce**, un contatore usato per assicurarsi che ogni transazione sia unica e elaborabile solo una volta
- **Il saldo di ether** corrente dell’account (”Ether” è il metodo di scambio interno di Ethereum ed è usato per pagare le commissioni per le transazioni.)
- **Il codice** del contratto dell’account, se presente
- **La memoria** dell’account (vuota per impostazione predefinita)

In generale, esistono due tipi di account: Externally Owned Accounts (EOAs) e i contract accounts. Gli Owned Accounts sono formati da una chiave privata; avere la chiave privata significa controllare l’accesso ai fondi o ai contratti dell’account. L’altra tipologia, i contract accounts, non ha una chiave privata ma sono controllati dalla logica del codice del proprio ”smart contract”. Come gli EOA i contract accounts possono ugualmente inviare e ricevere ethe. Tuttavia, quando la destinazione di una transazione è un indirizzo di contract accounts, il contratto viene eseguito nell’ EVM, usando come input la transazione e i suoi dati. Un EOA non contiene codice e nessuno può inviare messaggi da un EOA creando e firmando una transazione. Un contract account, invece, ogni volta che riceve un messaggio, avvia il codice di cui è composto permettendogli di leggere e scrivere nella memoria interna, di inviare altri messaggi o creare a sua volta altri contratti.

Si noti che i ”contratti” in Ethereum non dovrebbero essere visti come qualcosa che dovrebbe essere ”soddisfatto”, ma, come delle ”entità autonome” che sono memorizzate all’interno dell’ambiente di esecuzione di Ethereum. I contract accounts hanno un controllo diretto sul proprio saldo ether e sul proprio archivio di chiavi/valori.
Si noti che, non disponendo un account di contratto di una chiave privata, non può avviare una transazione. Solo gli EOA possono avviare le transazioni, ma i contract accounts possono reagire alle transazioni chiamando a loro volta altri contratti, costruendo percorsi di esecuzione complessi.[1]

3.3.2 Introduzione alle transazioni

Il termine ”transazione” viene utilizzato in Ethereum per fare riferimento ad un insieme di dati che vengono firmati e inviati ad un EOA. Le transazioni contengono:

- Il destinatario del messaggio
- Il destinatario del messaggio
- La quantità di ether da trasferire dal mittente al destinatario
- Un campo dati opzionale
- Uno valore STARTGAS, che rappresenta il numero massimo di passaggi computazionali che la transazione può compiere
- Un valore GASPRICE, che rappresenta la commissione che il mittente della transazione deve pagare.

I primi tre campi sono degli standard previsti in ogni criptovaluta.
Il campo dati non ha alcuna funzione predefinita, ma il sistema operativo della macchina virtuale può accedere a questo campo ed interpretarlo. Ad esempio: se un contratto funziona come un servizio di registrazione di domini su blockchain, allora potrebbe voler interpretare i dati che gli vengono trasmessi come contenenti due ”campi”: il primo è il dominio da registrare, il secondo è l’indirizzo IP su cui deve essere registrato. Il contratto leggerebbe tali valori dal campo dati del messaggio e li posizionerebbe in modo appropriato nella memoria.
Lo STARTGAS e il GASPRICE sono campi cruciali per il modello ripudiabile di Ethereum. Questo è necessario al fine di prevenire loop infiniti accidentali, detti flooding attack, o altri sprechi computazionali nel codice. In ogni transazione è impostato un limite che corrisponde al numero massimo di fasi computazionali per l’esecuzione del codice. L’unità fondamentale per questa operazione è il ”gas”. Ogni operazione computazionale costa 1 gas, ma questo valore può variare con l’aumentare dei dati che devono essere archiviati come parte dello stato. Vi è poi un’ulteriore commissione di 5 gas per ogni byte nei dati di transazione.
L’obiettivo del sistema è chiedere ad un eventuale utente malintenzionato di pagare per ogni risorsa , potenza di calcolo, larghezza di banda e archiviazione di cui fa richiesta. In questo modo una transazione che avviene nella rete, avrà delle commissioni che avranno un andamento approssimativamente proporzionale all’incremento del consumo delle risorse in gioco[6].

3.3.3 Introduzione ai messaggi

I messaggi sono oggetti non serializzati, non sottoposti ad un processo di scomposizione e ricomposizione, che esistono solo nell’ambiente d’esecuzione di Ethereum e che possono essere inviati da un account ad un’altro . Un messaggio è così composto:

- Il mittente del messaggio (implicito)

- Il destinatario del messaggio
- L'importo di ether da trasferire con il messaggio
- Un campo di dati facoltativo
- Il valore di STARGAS

Essenzialmente, un messaggio è come una transazione, ma può essere prodotto unicamente da un contratto e non da un attore esterno. Un messaggio è il risultato di una chiamata (call) prodotta da un contratto. Come con una transazione, un messaggio permette ad un account di eseguire il proprio codice esternamente.
Occorre tener presente che il quantitativo di gas, associato ad una transazione o ad un contratto, si applica al consumo totale per quella specifica transazione e anche per le esecuzioni secondarie ad essa collegate. Ad esempio, se un contract accounts A invia una transazione a B di 1000 gas, ma prima di raggiungere B consuma 600 gas per inviare un messaggio a C che utilizza 300 gas, a B arriveranno solo 100 gas[6].

3.4 Le transazioni in Ethereum

Le transazioni sono messaggi firmati originati da un EOA, trasmessi dalla rete Ethereum e registrati sulla blockchain di Ethereum. Sono le uniche operazioni che possono effettuare un cambiamento di stato o causare l'esecuzione di un contratto nell'EVM. Ethereum è una macchina a stati singleton globale e le transazioni sono ciò che fa "muovere" quella macchina a stati, cambiando il suo stato.

Innanzitutto, esaminiamo la struttura di base di una transazione, cos`ı come viene serializzata e trasmessa sulla rete Ethereum. Ogni client, o applicazione, che riceve una transazione serializzata la memorizzerà utilizzando la propria struttura dati interna, eventualmente arricchita con metadati. La serializzazione sulla rete rappresenta l'unico formato standard di una transazione. Una transazione è un messaggio binario serializzato che contiene i seguenti dati:

- **Nonce**
  Un numero di sequenza, emesso dall' EOA di origine, utilizzato per impedire la riproduzione del messaggio.
- **Gas price**
  Il prezzo del gas (in wei) che il mittente è disposto a pagare.
- **Gas limit**
  La quantità massima di gas che il mittente è disposto ad acquistare per questa transazione.
- **Destinatario**
  L'indirizzo Ethereum di destinazione.

- **Valore**
  La quantità di ether da inviare alla destinazione.
- **Dati**
  Il payload di dati binari di lunghezza variabile.
- **v,r,s**
  I tre componenti di una firma digitale ECDSA dell' EOA di origine.

La struttura del messaggio di transazione viene serializzata utilizzando lo schema di codifica Recursive Length Prefix (RLP), che è stato creato appositamente per una serializzazione dei dati semplice e perfetta per i byte in Ethereum. Tutti i numeri in Ethereum sono codificati come interi in modalità big-endian, in multipli di 8 bit di lunghezza.
Si noti che le etichette dei campi sono mostrate qui per chiarezza, ma non fanno parte dei dati serializzati della transazione, che contengono i valori del campo con codifica RLP. In generale, RLP non contiene delimitatori di campo o etichette. Il prefisso di lunghezza di RLP viene utilizzato per identificare la lunghezza di ciascun campo. Qualunque bit oltre la lunghezza definita appartiene al campo successivo nella struttura.
Questa è l'effettiva struttura delle transazioni trasmesse. La maggior parte delle rappresentazioni interne e delle interfacce utente vengono fornite con informazioni aggiuntive, derivate dalla transazione o dalla blockchain.
Si noti, ad esempio, che non vi sono dati che fanno riferimento all'indirizzo che identifica l'EOA del mittente. Questo perché la chiave pubblica dell' EOA può essere derivata dai componenti v,r,s della firma ECDSA. Altri metadati aggiunti frequentemente alla transazione dal software client includono il numero di blocco, una volta estratto e incluso nella blockchain, e un ID transazione, hash calcolato. Ancora una volta, questi dati derivano dalla transazione e non fanno parte del messaggio della transazione stessa[1].

3.4.1 Il Nonce nella transazione

Il numero di serie è uno dei componenti più importanti e meno compresi di una transazione. La definizione presente nel Yellow Paper recita:

*Il **nonce** è un valore scalare uguale al numero di transazioni inviate da questo indirizzo o, nel caso di un account con codice associato, il numero di creazioni di contratti effettuate da questo account[53].*

In altri termini, il nonce è un attributo dell'indirizzo di origine, ovvero, ha significato solo nel contesto dell'indirizzo di invio. Tuttavia, il nonce non è memorizzato esplicitamente come parte dello stato di un account sulla blockchain, ma viene calcolato dinamicamente, contando il numero di transazioni confermate che sono state originate da un determinato indirizzo.
Ci sono due scenari in cui l'esistenza di un nonce, che tiene traccia di una transazione, risultano essere importanti: la caratteristica di utilizzo delle transazioni che vengono incluse nell'ordine di creazione, e la protezione dalla duplicazione delle transazioni. Vediamo un esempio di scenario per ognuno di questi:

Nell'ipotesi di voler effettuare due transazioni: un pagamento importante di 6 ether, oltre a un altro pagamento di 8 ether. Firmi e trasmetti la transazione di 6 ether per prima, perché è quella più importante, e poi firmi e trasmetti la seconda transazione di 8 ether. Purtroppo, hai trascurato il fatto che il tuo account contiene solo 10 ether, quindi la rete non può accettare entrambe le transazioni: una delle due fallirà. Poiché hai inviato per prima quella di 6 ether, ci si aspetta che quella vada a buon fine e che la transazione di 8 ether sia rifiutata. Tuttavia, in un sistema decentralizzato come Ethereum, i nodi possono ricevere le transazioni in qualsiasi ordine; non c'è garanzia che un nodo in particolare abbia una transazione diffusa prima dell'altra. Pertanto, è quasi certo che alcuni nodi ricevano prima la transazione di 6 ether, e altri la transazione di 8 ether. Senza il nonce, sarebbe casuale il rifiuto o l'accettazione di una delle due transazioni. Tuttavia, con il nonce incluso, la prima transazione che hai inviato avrà un nonce esempio di 3, mentre la transazione di 8 ether ha il valore di nonce successivo, cioè 4. Quindi questa transazione verrà ignorata fino a quando le transazioni con i nonce da 0 a 3 non saranno elaborate, anche se la transazione con il nonce 4 viene ricevuta per prima.

Come altro esempio immaginiamo ora che una persona abbia un account con 100 ether. Qualcuno online accetta il pagamento in ether per un prodotto che vuol comprare. Dopo il pagamento di 2 ether ti invia il prodotto. Per effettuare il pagamento di 2 ether, viene firmata una transazione che invia 2 ether dall' account del compratore a quello del venditore, e che viene poi hai trasmessa alla rete Ethereum per essere verificata e inclusa nella blockchain. Senza un valore nonce nella transazione, una seconda transazione che invia 2 ether allo stesso indirizzo una seconda volta avrà lo stesso aspetto della prima transazione. Ciò significa che chiunque veda la tua transazione sulla rete Ethereum, incluso il destinatario o un potenziale attaccante, può ripetere la transazione più volte fino a quando tutto il tuo saldo ether sarà vuoto. Questo andando semplicemente a copiare e incollare la tua transazione originale e inviandola nuovamente alla rete. Tuttavia, con il valore nonce incluso nei dati della transazione, ogni singola transazione è diversa, anche quando si invia più volte la stessa quantità di ether allo stesso indirizzo del destinatario. Pertanto, avendo il nonce incrementale come parte della transazione, non è possibile per nessuno "duplicare" un pagamento che hai effettuato.

In sintesi, è importante notare che l'utilizzo del nonce è effettivamente vitale per un protocollo basato sugli account, a differenza del meccanismo "Unspent Transaction Output" (UTXO) del protocollo Bitcoin[1].

3.4.2 Implicazioni applicate al nonce

La rete Ethereum elabora le transazioni in sequenza, in base al nonce. Ciò significa che se trasmetti una transazione con nonce 0 e poi trasmetti una transazione con nonce 2, la seconda transazione non sarà inclusa in alcun blocco. Questo verrà archiviato nel mempool (buffer temporaneo in cui vengono archiviate momentaneamente le transazioni dell'utente, poi selezionate dai miner per elaborarle, mentre la rete Ethereum attende che appaia il nonce mancante. In questo modo i nodi penseranno che il nonce sia semplicemente in ritardo e che la transazione con nonce 2 sia stata ricevuta fuori sequenza.
Una volta trasmessa la transazione col nonce mancante, entrambe le transazioni (nonce 1 e 2) verranno elaborate e incluse, ovviamente se valide. Una volta colmato il vuoto, la rete può

estrarre la transazione fuori sequenza che conteneva nel mempool.
Ciò significa che se si creano diverse transazioni in sequenza e una di esse non viene ufficialmente inclusa in alcun blocco, tutte le transazioni successive rimarranno "bloccate", in attesa del nonce mancante. Una transazione può creare un "gap" involontario nella sequenza nonce perché non è valida o ha gas insufficiente. Per rimettere in moto le cose, occorre trasmettere una transazione valida con il nonce mancante. Quando una transazione con il nonce "mancante" viene convalidata dalla rete, tutte le transazioni di trasmissione con i successivi nonce diventeranno valide in modo incrementale. non essendo possibile "ritirare" una transazione.
Se invece viene duplicato un nonce, ad esempio trasmettendo due transazioni con lo stesso nonce ma a destinatari o valori diversi, allora una di esse verrà confermata e l'altra rifiutata. La transazione che sarà confermata sarà determinata dalla sequenza di arrivo al primo nodo di convalida che li riceve, e cioè sarà casuale[1].

3.4.3 La concoorenzza nelle transazioni

La concorrenza è un aspetto complesso dell'informatica e a volte si presenta inaspettatamente, specialmente nei sistemi decentralizzati e distribuiti come Ethereum.
In termini semplici, la concorrenza è quando si ha un calcolo simultaneo da parte di più sistemi indipendenti. Questi possono trovarsi nello stesso programma, ad es. multithreading, sulla stessa CPU, ad es. multiprocessing, o su computer diversi, ad es. sistemi distribuiti. Ethereum, per definizione, è un sistema che consente la concorrenza delle operazioni, nodi, client, DApp, ma impone uno stato singleton attraverso il consenso.
Di seguito alcuni esempi che mostrano la debolezza dei sistemi concorrenti nei sistemi distribuiti.
Si ipotizzi, di avere più applicazioni con exchange indipendenti che generano transazioni dallo stesso indirizzo. Un esempio di tale situazione potrebbe essere uno scambio che elabora i prelievi dall' exchange con un hot wallet, un wallet le cui chiavi sono archiviate online. Idealmente, bisognerebbe avere più di un computer che elabora i prelievi, in modo che non diventi un collo di bottiglia con un singolo punto di errore. Tuttavia, questo diventa rapidamente problematico, poiché avere più di un computer che produce prelievi comporterà alcuni problemi di concorrenza, non ultimo dei quali, la selezione dei nonce. Come si coordinano più computer che generano, firmano e trasmettono transazioni dallo stesso account hot wallet?
Una possibile soluzione potrebbe essere quella di utilizzare un singolo computer per assegnare nonce, in base all'ordine di arrivo, per tutti computer che firmano le transazioni. Tuttavia, questo computer è ora un singolo punto di errore. Questa soluzione però non risolve il problema anzi, se vengono assegnati diversi nonce e uno di essi non viene mai utilizzato, a causa di un errore nel computer che elabora la transazione con quel nonce, tutte le transazioni successive si bloccano.
Un altro approccio sarebbe quello di generare le transazioni, ma non assegnare loro un nonce, parte integrante e che le autentica, lasciandole non firmate, proponendo come soluzione il creare in coda alle transazioni un singolo nodo che li firma e tiene anche traccia dei nonce. Anche in questo caso, questo sarebbe un punto di strozzatura nel processo. La firma e il monitoraggio dei nonce è la parte dell'operazione che ha più probabilità di diventare congestionata sotto carico, mentre la generazione della transazione non firmata è la parte che in realtà non andrebbe parallelizzata, poiché si otterrebbe una certa concorrenza, ma mancherebbe una parte critica del processo.
Questi problemi di concorrenza, oltre alla difficoltà di tenere traccia dei saldi dei conti e delle

conferme delle transazioni nei processi indipendenti, costringono la maggior parte delle singole implementazioni a evitare la concorrenza e a creare colli di bottiglia come un singolo processo che gestisce tutto[1].

3.4.4 Il destinatario nelle transazioni

Il destinatario di una transazione è specificato nel campo a**??**. Ethereum non convalida ulteriormente questo campo. Qualsiasi valore di 20 byte è considerato valido. Se il valore di 20 byte corrisponde a un indirizzo senza chiave privata corrispondente o senza contratto corrispondente, la transazione è ancora valida. Ethereum non ha modo di sapere se un indirizzo è stato correttamente derivato da una chiave pubblica, e quindi da una chiave privata?, esistente.
L'invio di una transazione all'indirizzo sbagliato probabilmente brucerà l'ether inviato, rendendolo per sempre inaccessibile, non spendibile, poiché la maggior parte degli indirizzi non ha una chiave privata nota e quindi non può essere generata alcuna firma per spenderla. In effetti, ci sono una serie di validi motivi per bruciare ether, ad esempio come disincentivo a imbrogliare nei canali di pagamento e poiché la quantità di ether è finita, il bruciare ether distribuisce efficacemente il valore bruciato a tutti i possessori di ether, in proporzione alla quantità di ether che detengono[1].

3.5 Funzione dello stato di transizione di Ethereum

In riferimento a quanto già visto nel precedente capitolo con Bitcoin 2.1.3 ora ne esamiamo le differenze con Ethereum.

La funzione dello stato di transizione di Ethereum, APPLY(S,TX) $-> S'$ può essere definito come segue:

1. Si verifica se la transazione è ben formata, con il numero corretto di parametri, se la firma è valida e se il nonce corrisponde a quello del conto del mittente. Se non è così, viene restituito un errore.
2. Si calcola la commissione di transazione come STARTGAS * GASPRICE, si determina l'indirizzo del mittente dalla firma. Si sottrae la commissione dal saldo del conto del mittente e si incrementa il nonce del mittente. Se il saldo non è sufficiente, viene restituito un errore.

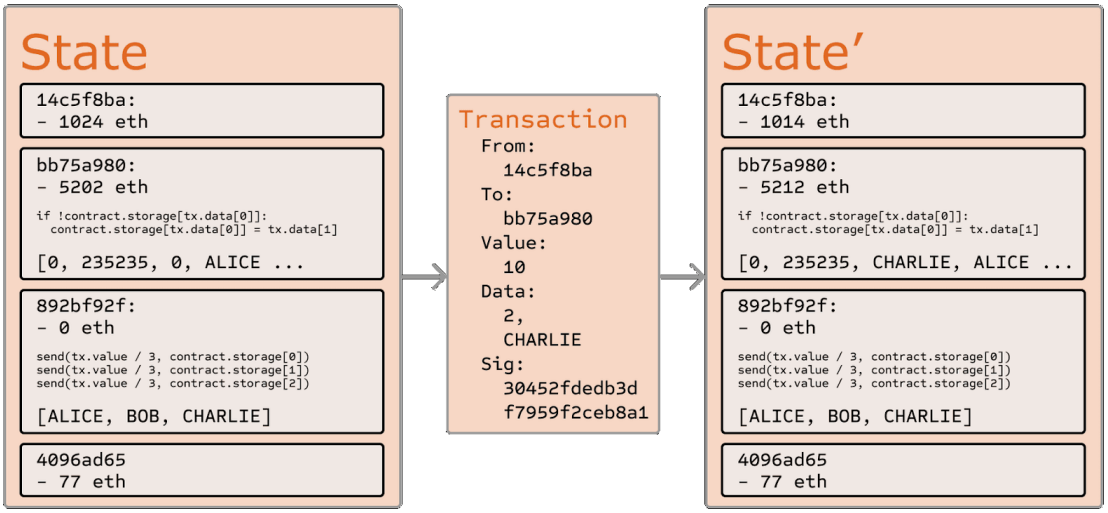

Figura 25: Rappresentazione dei passaggi di stato nelle transazioni di Ethereum [6]

3. Si inizializza GAS = STARTGAS e si preleva un dato importo di gas, per byte, per pagare il costo della transazione.

4. Si trasferisce il valore della transazione dal conto del mittente al conto ricevente. Se non viene trovata l'occorrenza del conto del destinatario, lo si crea. Se il conto del ricevente è un contratto, si esegue il codice del contratto fino al suo completamento o finché il processo non termina il gas.

5. Se il trasferimento di valore fallisce per la mancanza di disponibilità economica del mittente, o perché l'esecuzione del codice ha terminato il gas, vengono ripristinate tutte le modifiche di stato tranne il pagamento delle commissioni le quali vengono aggiunte al conto del miner. In caso di successo invece, vengono rimborsate le commissioni per tutto il gas non speso al mittente e vengono inviate le commissioni al miner.

[6]

3.5.1 Esecuzione della transazione

Il codice nei contratti di Ethereum è scritto in un linguaggio bytecode basato sullo stack di basso livello, detto "codice della macchina virtuale di Ethereum" o "codice EVM". Il codice consiste in una serie di byte, dove ogni byte rappresenta un'operazione. In generale, l'esecuzione del codice è un ciclo infinito che consiste nella ripetuta esecuzione dell'operazione al contatore corrente del programma, che inizia da zero, e poi nell'aumento del contatore del programma, di una unità, finché non viene raggiunta la fine del codice, si verifica un errore o viene rilevata l'istruzione STOP o RETURN. Le operazioni hanno accesso a tre tipi di spazio in cui archiviare i dati:

- Lo **stack**, un contenitore last-in-first-out in cui è possibile eseguire il push e il pop dei valori
- La **memoria**, un array di byte espandibili infinitamente
- Lo **storage** è il luogo di termine del contratto, un archivio di chiavi/valori. A differenza dello stack e della memoria, che si ripristinano al termine delle operazioni, l'archivio persiste a lungo termine.

Il codice può anche accedere al valore, al mittente, ai dati del messaggio in arrivo e ai dati dell'intestazione del blocco. Può anche restituire una matrice di dati di byte come output.
Il modello di esecuzione formale del codice dell'EVM è sorprendentemente semplice. Mentre la macchina virtuale Ethereum è in esecuzione, il suo stato di calcolo completo può essere definito dalla tupla (block-state, transaction, message, code, memory, stack, pc, gas), in cui block-state è lo stato globale, che contiene tutti i conti e include i saldi e l'archiviazione. All'inizio di ogni ciclo di esecuzione, l'istruzione corrente viene trovata prendendo l'ennesimo pc del bytecode, o 0 se $pc >= len(code)$, e questa contiene le modalità per influenzare definisce come influisce sulla tupla. Ad esempio, *ADD* estrae due elementi dallo stack e ne restituisce la somma, riduce gas di 1 e incrementa pc di 1 e SSTORE espelle i primi due elementi dallo stack.
A questo punto inserisce il secondo elemento nell'archiviazione del contratto all'indice specificato dal primo elemento. Sebbene ci siano molti modi per ottimizzare l'esecuzione della macchina virtuale di Ethereum tramite una compilazione *just-in-time*, un'implementazione di base di Ethereum è eseguibile in poche centinaia di righe di codice[6].

3.5.2 La propagazione delle transazioni

La rete Ethereum utilizza un protocollo "flood routing". Ogni client Ethereum funziona da nodo in una rete peer-to-peer (P2P), che, idealmente, forma una rete mesh. Nessun nodo di rete è speciale: agiscono tutti come peer uguali. Con il termine "nodo" si fa riferimento ad un client Ethereum che è connesso e che partecipa alla rete P2P.
La propagazione della transazione inizia con il nodo Ethereum di origine che crea, o riceve da offline, una transazione firmata. La transazione viene convalidata e quindi trasmessa a tutti gli altri nodi Ethereum direttamente collegati al nodo di origine. In media, ogni nodo Ethereum mantiene connessioni ad almeno altri 13 nodi. Ogni nodo "vicino" convalida la transazione non appena la riceve. Se concordano sulla sua validità, né memorizzano una copia e la propagano a tutti i nodi raggiungibili. Di conseguenza, la transazione si propaga verso l'esterno dal nodo di origine, "propagandosi" attraverso la rete, finché tutti i nodi della rete non ne avranno una copia. I nodi possono filtrare i messaggi che propagano, ma l'impostazione predefinita è propagare tutti i messaggi di transazione validi che ricevono.
In pochi secondi, una transazione Ethereum si propaga a tutti i nodi Ethereum. Dal punto di vista di ciascun nodo, non è possibile distinguere l'origine della transazione. Il vicino che lo ha inviato al nodo potrebbe essere l'originatore della transazione o potrebbe averlo ricevuto da uno dei suoi vicini. Per poter tracciare le origini delle transazioni o interferire con la propagazione, un utente malintenzionato dovrebbe controllare una percentuale significativa di nodi. Questo fa parte del design di sicurezza e privacy delle reti P2P, in particolare se applicato alle reti blockchain[1].

3.5.3 Registrazione sulla Blockchain

Sebbene tutti i nodi di Ethereum siano considerati alla pari, alcuni di essi sono gestiti da minatori che alimentano transazioni e blocchi per le mining farm, che sono computer con unità di elaborazione grafica (GPU) ad alte prestazioni. I computer di mining aggiungono transazioni a un blocco candidato e tentano di trovare una proof-of-work che renda valido il blocco candidato. Una volta minate in un blocco, le transazioni modificano anche lo stato del singleton di Ethereum, sia modificando il saldo del conto, nel caso di un semplice pagamento, sia invocando contratti che ne modificano lo stato interno. Queste modifiche sono registrate insieme alla transazione, sotto forma di "ricevuta" della transazione, che può includere anche eventi[1].

3.5.4 Blockchain e mining

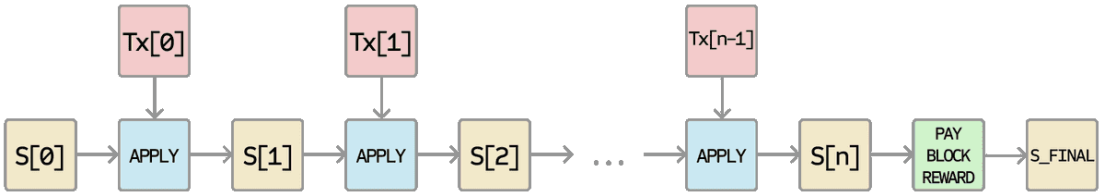

Figura 26: Rappresentazione grafica della struttura dati di Ethereum [6]

La blockchain di Ethereum è per molti versi simile a quella di Bitcoin, ma con alcune differenze. La differenza principale tra Ethereum e Bitcoin, riguardo l'architettura della blockchain, è che, Bitcoin contiene solo una copia dell'elenco delle transazioni, mentre i blocchi di Ethereum hanno una copia sia dell'elenco delle transazioni che dello stato più recente. Inoltre, sono anche memorizzati nel blocco, anche altri due valori, il numero del blocco ricavabile dal nonce e la difficoltà. L'algoritmo di convalida del blocco di base in Ethereum è il seguente:

1. Verifica se il blocco a cui si fa riferimento in precedenza esiste ed è valido
2. Verifica che l'indicatore temporale"timestamp" del blocco sia maggiore di quella del blocco precedente di riferimento e meno di 15 minuti nel futuro
3. Verifica che il numero del blocco, la difficoltà, la radice della transazione, la radice dell'ommer e il limite di gas, siano validi.
4. Verifica che il proof-of-work del blocco sia valido.

Esempio : Supponiamo che $S[0]$ sia lo stato alla fine del blocco precedente. Consideriamo TX l'elenco delle transazioni del blocco, con n transazioni. Per tutti gli i in $0...n$ 1, si imposta $S[i + 1] = APPLY\ (S[i],\ TX[i])$. Se in qualsiasi fase della applicazione mette in evidenza un errore, o se il gas totale consumato nel blocco fino a questo punto supera il GASLIMIT, la transazione restituisce un errore.
Viene emesso $S_F INALS[n]$, aggiungendo la ricompensa del blocco pagata al miner.

5. Verifica che la radice dell'albero di Merkle dello stato S_{FINAL} sia uguale alla radice dello stato finale fornita nell'intestazione del blocco. In caso affermativo, il blocco è valido; in caso negativo, no.

A prima vista l'approccio può sembrare poco efficace, perché deve memorizzare l'intero stato con ogni blocco, ma in realtà l'efficienza dovrebbe essere paragonabile a quella di Bitcoin. Il motivo è che lo stato è memorizzato nella struttura ad albero e dopo ogni blocco è necessario modificare solo una piccola parte di esso. Quindi, in generale, tra due blocchi adiacenti la grande maggioranza dell'albero dovrebbe essere la stessa, e quindi i dati possono essere archiviati una volta e referenziati due volte usando i puntatori, es. hash di sottoalberi. Per questo motivo viene usato un tipo di albero speciale, noto come "Patricia tree", includendo una modifica al concetto di albero di Merkle che consente ai nodi di essere inseriti ed eliminati, e non solo modificati, in modo efficiente. Inoltre, poiché tutte le informazioni sullo stato fanno parte dell'ultimo blocco, non è necessario archiviare l'intera cronologia della blockchain: una strategia che, se potesse essere applicata a Bitcoin, potrebbe fornire un risparmio calcolato di spazio di 5-20 volte.
Una domanda frequente è "dove" viene eseguito il codice del contratto, in termini di hardware fisico.
La risposta è semplice: il processo di esecuzione del codice del contratto fa è parte della definizione della funzione di transizione tra stati, che a sua volta fa parte dell'algoritmo di convalida del blocco.Pertanto se una transazione viene aggiunta nel blocco B il codice di esecuzione generato da quella transazione verrà eseguito da tutti i nodi, ora e in futuro, che scaricano e convalidano il blocco B[6].

3.6 Gas

Nelle sezioni precedenti è stato nominato il concetto di gas, di seguito si cercherà di entrare più nel dettaglio sul ruolo dei componenti gasPrice e gasLimit in una transazione.
Il gas è il carburante di Ethereum. Il gas non è ether, è una valuta virtuale separata con un proprio tasso di cambio rispetto all'ether. Ethereum utilizza il gas per controllare la quantità di risorse che una transazione può utilizzare, poiché verrà elaborata su migliaia di computer in tutto il mondo. Il modello di calcolo a tempo non deterministico, Turing-complete, richiede una qualche forma di misurazione per evitare attacchi denial-of-service o transazioni inavvertitamente onerose in ambito di risorse. Il gas è separato dall'ether per proteggere il sistema dalla volatilità che potrebbe insorgere insieme ai rapidi cambiamenti del valore dell'ether, e anche come un modo per gestire gli importanti e sensibili rapporti tra i costi delle varie risorse che il gas paga, vale a dire, calcolo, memoria e archiviazione.
Il valore gasPrice, in una transazione, consente all'emittente della transazione di impostare il prezzo che è disposto a pagare in gas. Il prezzo è misurato in wei per unità di gas. I portafogli possono regolare il gasPrice nelle transazioni che originano per ottenere una conferma più rapida delle transazioni. Più alto è il gasPrice, più velocemente è probabile che la transazione venga confermata. Al contrario, le transazioni con priorità inferiore possono comportare un prezzo ridotto, con conseguente rallentamento della conferma. Il valore minimo su cui gasPrice può essere impostato è zero, il che significa una transazione senza commissioni. Durante i periodi di bassa domanda di spazio in un blocco, tali transazioni potrebbero benissimo essere minate.
Il secondo campo importante relativo al gas è il gasLimit. Il gasLimit fornisce il numero massimo di unità di gas che il mittente della transazione è disposto ad acquistare per completare la

transazione. Per i pagamenti semplici, ovvero transazioni che trasferiscono ether da un EOA a un altro EOA, la quantità di gas necessaria è fissata a 21.000 unità di gas.
Se l'indirizzo di destinazione della tua transazione è un contratto, la quantità di gas necessaria può essere stimata ma non può essere determinata con precisione. Questo perché un contratto può valutare diverse condizioni che portano a diversi percorsi di esecuzione, con diversi costi totali del gas. Il contratto può eseguire solo un semplice calcolo o uno più complesso, a seconda di condizioni che sono al di fuori del controllo del mittente e non possono essere previste. Per dimostrarlo, proviamo a fare un esempio. Possiamo scrivere uno smart contract che incrementa un contatore ogni volta che viene chiamato ed esegue un particolare ciclo un numero di volte pari al conteggio delle chiamate. Questo smart contract alla centesima chiamata distribuisce un premio speciale, come in una lotteria, che per essere eseguito deve effettuare operazioni aggiuntive per calcolare il premio. Se chiami il contratto 99 volte ci sarà una certa risposta alla centesima la risposta sarà diversa.
Un'analogia comune a chi utilizza Ethereum per descrivere il gasLimit, può essere quella di pensare alla capacità del serbatoio del carburante di un'auto (l'auto è la transazione). Il serbatoio viene riempito con tutto il gas necessario per il viaggio (il calcolo necessario per convalidare la transazione). L'importo può essere stimato, ma potrebbero esserci modifiche impreviste al viaggio, come una deviazione (un percorso di esecuzione più complesso), che aumentano il consumo di carburante.
Tuttavia, l'analogia con un serbatoio di carburante è in qualche modo fuorviante. In realtà è più simile a un conto di credito per una società di stazioni di servizio, dove si paga dopo il completamento del viaggio, in base alla quantità di carburante che è stata effettivamente utilizzata. Quando si trasmette una transazione, uno dei primi passaggi di convalida è verificare che l'account da cui ha avuto origine abbia abbastanza ether per pagare la tariffa gasPrice * gas. Ma l'importo non viene effettivamente detratto dal tuo conto fino al termine dell'esecuzione della transazione. Viene addebitato solo il gas effettivamente consumato dalla transazione, ma si deve avere un saldo sufficiente per l'importo massimo che si è disposti a pagare prima di inviare la transazione[1].

3.6.1 Valuta ed emissione

La rete Ethereum include una propria valuta integrata, l'ether, che ha il duplice scopo di fornire un livello di liquidità primaria per consentire uno scambio efficiente tra vari tipi di risorse digitali e, soprattutto, un meccanismo per il pagamento delle commissioni sulle transazioni. Per maggiore chiarezza definiamo qui di seguito i vari tagli presenti in ether:

- 1: wei
- 10^{12} : szabo
- 10^{15} : inney
- 10^{18} : ether

Deve essere considerata come una versione ampliata del concetto di ”dollari” e ”centesimi” o ”BTC” e ”satoshi”.

Tabella 1: Tabella dimostrativa delle commissioni

Gruppo	Al lancio	Dopo 1 anno	Dopo 5 anni
Unità di valuta	1,198X	1,458X	2,498X
Acquirenti (%)	83,5	68,6	40,0
Riserva spesa pre-vendita (%)	8,26	6,79	3,96
Riserva utilizzata post-vendita (%)	8,26	6,79	3,96
Miner (%)	0	17,8	52,0

[6]

3.7 Proof of Stake (PoS)

Proof of Stake (PoS) è una categoria di algoritmi di consenso per blockchain pubblici che dipendono dalla quota economica di un validatore nella rete.
Nelle blockchain pubbliche basate su proof of work (PoW), ad es. Bitcoin e l’attuale implementazione di Ethereum, l’algoritmo premia i partecipanti che risolvono enigmi crittografici per convalidare le transazioni e creare nuovi blocchi, ad es. mining.
Nelle blockchain pubbliche basate su PoS . Una serie di validatori propongono a turno e votano il blocco successivo, e il peso del voto di ogni validatore dipende dalla dimensione del suo deposito.I vantaggi significativi del PoS includono la sicurezza, la riduzione del rischio di centralizzazione e l’efficienza energetica.
In generale, un algoritmo di Proof of Stake si presenta come segue. La blockchain tiene traccia di una serie di validatori e chiunque detenga la criptovaluta di base della blockchain, ether nel caso di Ethereum può diventare un validatore inviando un tipo speciale di transazione che blocca il proprio ether in un deposito.Il processo di creazione e accettazione di nuovi blocchi viene quindi eseguito tramite un algoritmo di consenso a cui possono partecipare tutti i validatori attuali.
Esistono molti tipi di algoritmi di consenso e molti modi per assegnare ricompense ai validatori che partecipano all’algoritmo di consenso, quindi ci sono diverse tipologie di proof of stake.
Da un punto di vista algoritmico, ci sono due tipi principali: il **chain-based proof of stake** e il **BFT-style proof of stake**.

Nel **chain-based proof of stak**, l’algoritmo seleziona in modo pseudo-casuale un validatore durante ogni fascia oraria, ad es. un periodo di 10 secondi potrebbe essere una fascia oraria, e assegna a quel validatore il diritto di creare un singolo blocco. Questo blocco deve puntare ad ù uno blocco precedente, ù di solito quello alla fine della più lunga catena precedentemente . In questo modo, nel tempo, la maggior parte dei blocchi convergono in un’unica catena in costante crescita.

Nel **BFT-style proof of stake**, ai validatori viene assegnato in modo casuale il diritto di proporre blocchi, ma l’accordo su quale blocco non è standardizzato ed è effettuato attraverso un processo a più round in cui ogni validatore invia un ”voto” per un blocco specifico durante ogni round.

Alla fine del processo tutti i validatori concordano in modo permanente se un dato blocco faccia o meno parte della catena. Si noti che i blocchi possono ancora essere concatenati insieme. La differenza fondamentale è che il consenso su un blocco può rientrare in un blocco e non dipende dalla lunghezza o dalla dimensione della catena successiva.

3.7.1 I vantaggi del Proof of Stake rispetto al Proof of work

Di seguito schematizzato in breve i vantaggi principali:

- Non c'è bisogno di consumare grandi quantità di elettricità per proteggere una blockchain, ad esempio, si stima che sia Bitcoin che Ethereum brucino oltre 1 $Mio in costi di elettricità e hardware al giorno come parte del loro meccanismo di consenso.
- A causa della mancanza di un elevato consumo di elettricità, non è necessario emettere tante nuove monete per motivare i partecipanti a continuare a partecipare alla rete. In teoria potrebbe anche essere possibile avere un'emissione di cripto valuta netta negativa, in cui una parte delle commissioni di transazione viene "bruciata" e quindi l'offerta diminuisce nel tempo.
- La prova del Proof of Stake porta a una gamma più ampia di tecniche che utilizzano la progettazione di meccanismi basati sulla teoria dei giochi per scoraggiare meglio la formazione di cartelli (lobby) centralizzati e, se questi si formano, dall'agire in modi che sono dannosi per la rete.
- Riduzione dei rischi di centralizzazione, poiché le economie di scala sono un problema molto minore.
- Possibilità di utilizzare sanzioni economiche per rendere le varie forme di attacchi del 51% molto più costose da eseguire rispetto al proof of work

Capitolo 4

Smart Contract e Solidity

In questo capitolo, vengono descritte le basi che definiscono la blockchain di Ethereum ovvero: gli Smart Contract e uno dei linguaggi maggiormente utilizzati dai programmatori ovvero Solidity. Cominceremo con una panoramica su cosa sono gli Smart Contract e come Solidity li renda operativi, con particolare enfasi su funzioni, eventi e modifiers. Sarà presentato un esempio concreto di uno Smart Contract per comprenderne il funzionamento.

Nella seconda parte del capitolo, presenteremo la Macchina Virtuale Ethereum (EVM) e la tecnica con cui avviene la compilazione di Solidity in bytecode EVM. Questo passaggio è fondamentale per garantire l'esecuzione corretta degli Smart Contract.

Infine, esploreremo l'analisi del codice Solidity, approfondendo il processo di reversing. Vedremo come sia possibile esaminare e comprendere l'interno di uno Smart Contract, evidenziando opportunità sia per la verifica che per la sicurezza. Attraverso esempi specifici di reversing del codice Solidity e mediante l'utilizzi di alcuni tools, metteremo in luce le sfide e le opportunità di questa pratica, fornendo una panoramica su come analizzare e comprendere a fondo gli Smart Contract basati su Ethereum.

4.1 Smart Contract

Il termine Smart contract è stato utilizzato nel corso degli anni per descrivere un'ampia varietà di progetti. Negli anni '90, il crittografo Nick Szabo coniò il termine e lo definì come *"The contract: a set of promises agreed to in a "meeting of the minds", is the traditional way to formalize a relationship"*.[45] Da allora, il concetto di contratti intelligenti si è evoluto, soprattutto dopo

l'introduzione delle piattaforme blockchain decentralizzate con l'invenzione di Bitcoin nel 2009. Nella piattaforma di Ethereum utilizziamo il termine "smart contract" per riferirci a software immutabili che vengono eseguiti in modo deterministico nel contesto di una macchina virtuale come parte del protocollo di rete. Analizziamo separatamente questa definizione:

Software

Gli smart contract sono semplicemente programmi. La parola "contratto" non ha alcun significato giuridico in questo contesto.

Immutabile

Una volta implementato, il codice di uno smart contract non può cambiare. A differenza del software tradizionale, l'unico modo per modificare uno smart contract è distribuire una nuova istanza.

Deterministico

Il risultato dell'esecuzione di uno smart contract è lo stesso per tutti coloro che lo eseguono.

Contesto EVM

Gli smart contract operano con un contesto di esecuzione molto limitato. Possono accedere al proprio stato, al contesto della transazione che li ha chiamati e ad alcune informazioni sui blocchi più recenti.

Computer mondiale decentralizzato

L'EVM viene eseguito come istanza locale su ogni nodo Ethereum, ma poiché tutte le istanze dell'EVM operano partendo dallo stesso stato iniziale e producono lo stesso stato finale, il sistema nel suo insieme funziona come un singolo "computer mondiale".[1]

4.2 Ciclo di vita di uno Smart Contract

Gli smart contract vengono solitamente scritti in un linguaggio di alto livello come Solidity, ma per l'esecuzione devono essere compilati in bytecode di basso livello, che sarà interpretato dall'EVM. Una volta compilati, vengono distribuiti sulla piattaforma Ethereum tramite una speciale transazione di creazione del contratto. Ciascun contratto viene identificato da un indirizzo Ethereum che deriva dalla transazione di creazione del contratto, ed è basato sull'account di origine e sul nonce. Questo indirizzo Ethereum può essere utilizzato in una transazione, sia per inviare fondi al contratto, sia per richiamare una delle sue funzioni. Al contrario degli account esterni controllati da chiavi private (EOA), non esistono chiavi associate agli account creati per nuovi smart contract. In qualità di creatore del contratto, non si ottiene nessun privilegio speciale a livello di protocollo.

È importante notare che i contratti vengono eseguiti solo quando vengono richiamati da una

Figura 27: Immagine selezionata dal Libro OUT OF THE ETHER [34]

transazione. In altre parole, tutti gli smart contracts in Ethereum vengono avviati ed eseguiti da una transazione iniziale inviata da un account esterno controllato da una chiave privata (EOA). Anche se un contratto richiama un altro contratto, e quest'ultimo richiama a sua volta un terzo contratto, il primo contratto nella catena di esecuzione è sempre stato avviato da una transazione proveniente da un EOA. I contratti non eseguono mai operazioni "da soli" o in background. È inoltre importante notare che gli smart contract non vengono eseguiti in modo "parallelo" in alcun senso; il computer che esegue il software Ethereum può essere considerato come una macchina a thread singolo.

Le transazioni in Ethereum sono atomiche, indipendentemente dal numero di contratti coinvolti o dalle operazioni che devono eseguire quando vengono richiamate. Le transazioni vengono eseguite integralmente e qualsiasi modifica allo stato globale (contratti, account, ecc.) viene registrata solo se l'esecuzione termina con successo. L'esecuzione è considerata riuscita solo se il programma viene eseguito senza errori e raggiunge la fine dell'esecuzione. In caso di errore, tutte le modifiche di stato vengono "annullate" come se la transazione non fosse mai stata eseguita.[1][11]

4.3 Costruire uno Smart contract con solidity

Solidity è un linguaggio di programmazione utilizzato per creare smart contract su piattaforme blockchain, come Ethereum. In termini più semplici, è uno strumento utilizzato per scrivere programmi che vengono archiviati ed eseguiti su una rete decentralizzata di computer. Solidity è stato ideato dal Dr. Gavin Wood, co-fondatore di Ethereum, come linguaggio specifico per

la scrittura di smart contract con funzionalità atte a supportare direttamente l'esecuzione nell'ambiente decentralizzato di Ethereum. È un linguaggio di alto livello, il che significa che è più facile da leggere e scrivere rispetto ai linguaggi di basso livello come il codice macchina. È stato progettato per essere semplice e facile da imparare, ma anche abbastanza potente da creare smart contract complessi.

Solidity è un linguaggio compilato con funzionalità integrate come i modificatori di funzioni, che consentono agli sviluppatori di specificare le condizioni che devono essere soddisfatte prima che una funzione possa essere eseguita. Ciò aiuta a prevenire l'accesso non autorizzato e protegge dalle vulnerabilità della sicurezza come gli attacchi di reentrancy. Solidity è anche un linguaggio tipizzato staticamente, il che significa che le tipologie delle variabili sono definite prima del runtime. In un linguaggio tipizzato staticamente come Solidity, le variabili e le funzioni hanno un tipo di dati definito che viene determinato in fase di compilazione prima che il codice venga eseguito. Ciò significa che il compilatore Solidity controlla il codice per eventuali errori relativi al tipo prima che venga distribuito sulla blockchain. Ad esempio, se una variabile è definita come numero intero nel codice Solidity, non può essere assegnato un valore stringa in fase di runtime.. Al contrario, i linguaggi tipizzati dinamicamente come JavaScript non richiedono che le variabili abbiano alcuna tipologia di dati definita e questa può cambiare in fase di esecuzione. Ciò può rendere più difficile individuare gli errori relativi al tipo prima che il codice venga eseguito, il che può portare a vulnerabilità della sicurezza. Essendo tipizzato staticamente, Solidity può rilevare errori relativi al tipo in fase di compilazione prima che il codice venga distribuito sulla blockchain. Ciò aiuta a prevenire una serie di vulnerabilità della sicurezza, inclusi attacchi di buffer overflow, attacchi di coercizione che dipendono dalla tipologia, e altri exploit che possono verificarsi quando i tipi di dati non sono definiti correttamente. Uno dei "prodotti" principali del progetto Solidity, infatti è il compilatore Solidity, solc, che converte i programmi scritti nel linguaggio Solidity in bytecode EVM. Il progetto gestisce anche l'importante standard ABI (Application Binary Interface) per gli smart contract di Ethereum, che esploreremo in dettaglio in questo capitolo. Ciascuna versione del compilatore Solidity crea una versione specifica del linguaggio Solidity.

Esplorazione dei concetti fondamentali nella programmazione di Solidity.

Funzioni

In Solidity, le funzioni sono blocchi di codice che possono essere chiamati per eseguire un'attività specifica o restituire un valore specifico. Le funzioni sono un elemento fondamentale dei contratti Solidity e sono ampiamente utilizzate nello sviluppo di smart contract. Le funzioni in Solidity possono essere definite utilizzando la parola chiave "function", seguita dal nome della funzione, un insieme di parametri di input (se presenti) e il codice da eseguire quando questa viene chiamata. Ecco un esempio di funzione Solidity che aggiunge due numeri interi:

```
function addNumbers(uint a,uint b) public returns (uint)        {
return a+ b;
}
```

In questo esempio, la funzione si chiama "addNumbers," accetta due parametri di input (a e

b) e restituisce la somma dei due valori di input come un intero senza segno. La parola chiave ”public” nella definizione della funzione specifica indica che questa può essere chiamata dall’esterno del contratto, mentre la parola chiave ”returns” specifica il tipo di dato restituito dalla funzione.

Le funzioni in Solidity possono anche avere parole chiave aggiuntive che ne modificano il comportamento. Ad esempio, la parola chiave ”view” può essere utilizzata per indicare una funzione che non modifica lo stato del contratto, mentre la parola chiave ”pure” può essere utilizzata per indicare che una funzione che non legge né modifica lo stato del contratto.

Ecco un esempio di funzione ”view” che calcola l’area di un rettangolo:

```
function calculateArea(uint width, uint height)
                        public view returns (uint) {
return width * height;
}
```

In questo esempio, la funzione si chiama ”calculate Area”, accetta due parametri di input (width e height) e restituisce il prodotto dei due valori di input come un numero intero senza segno. La parola chiave ”view” nella definizione della funzione specifica che questa non modifica lo stato del contratto. Un esempio di una funzione Solidity che utilizza un tipo di dati personalizzato:

```
struct Person {
 string name;
 uint age;
}

function createPerson(string memory Name, uint Age)
                        public returns (Person memory)    {
 Person memory newPerson = Person(Name, Age);
 return newPerson;
}
```

In questo esempio, definiamo un tipo di dati personalizzato chiamato ”Person” che ha due campi: un campo ”name” di tipo string e un campo ”age” di tipo uint. La funzione ”createPerson” accetta due parametri di input (Name e Age) e restituisce un oggetto di tipo ”Person.”
Quando viene chiamata la funzione ”createPerson,” essa crea un nuovo oggetto ”Person” utilizzando i parametri di input per poi restituirli. La parola chiave ”memory” nella definizione della funzione specifica che il nuovo oggetto ”Person” deve essere archiviato in memoria, anziché in storage.
Questo esempio mostra come utilizzare le funzioni Solidity per lavorare con tipi di dati personalizzati e restituire oggetti complessi. Definendo tipi di dati personalizzati e funzioni che operano su di essi, gli sviluppatori Solidity possono creare contratti intelligenti e DApp più complesse e potente.

Modifiers I modifiers sono funzioni speciali del linguaggio di programmazione Solidity che consentono di applicare precondizioni e postcondizioni ad altre funzioni. I modifiers possono essere utilizzati per verificare determinate condizioni prima di eseguire una funzione o per eseguire controlli o operazioni aggiuntive dopo che una funzione è stata eseguita.
Aggiungendo pre e post condizioni alle funzioni, si può garantire che il codice si comporti come previsto e possa aiutare a prevenire vulnerabilità di sicurezza. Ecco un esempio di un modificatore Solidity che aggiunge un controllo per l'indirizzo del mittente prima di eseguire una funzione:

```
// define a modifier that checks the sender's address
modifier onlyOwner() {
    require(msg.sender == owner, "Only the contract owner can call this function");
    _;
}
// define a function that uses the modifier
function updateData(uint256 newData) public onlyOwner {
    data = newData;
}
```

In questo esempio, definiamo un modificatore chiamato "onlyOwner" che controlla se "msg.sender" , l'indirizzo dell'account che ha chiamato la funzione, è uguale a questo del proprietario ("owner"). Se la condizione non viene soddisfatta, la funzione "require()" genererà un errore e la funzione non verrà eseguita.
Definiamo quindi una funzione chiamata "updateData" che accetta un parametro "newData" di tipo uint256. Utilizzando il modificatore "onlyOwner" per limitare l'accesso a questa funzione, in modo che solo il titolare del contratto possa richiamarla.
Per utilizzare un modificatore, bisogna aggiungerlo prima della definizione della funzione, separato da uno spazio, come mostrato nella funzione "updateData". Il carattere di sottolineatura (_) è una sintassi speciale che dice a Solidity di eseguire il corpo della funzione modificata dopo che il codice modificatore è stato eseguito.[1][11]

4.4 Scrittura di un contratto

Per il nostro primo esempio, scriveremo un contratto che controlla un faucet . Un faucet gestisce un meccanismo semplice: distribuisce ether a qualsiasi indirizzo che lo richiede e può essere riempito periodicamente. Puoi implementare un faucet come un portafoglio controllato da un essere umano o da un server web.

Esempio 1. **Faucet.sol**: https://github.com/ethereumbook/ethereumbook/

```
// SPDX-License-Identifier: CC-BY-SA-4.0
// Version of Solidity compiler this program was written for
pragma solidity 0.6.4;
// Our first contract is a faucet!
contract Faucet {
    // Accept any incoming amount
    receive() external payable {}
```

```
    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {
        // Limit withdrawal amount
        require(withdraw_amount <= 100000000000000000);


        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
}
```

Selezione di un compilatore Solidity e di una versione del linguaggio Come si vede dal codice del contratto, viene compilato correttamente con la versione 0.4.21 di Solidity per esprimere meglio in quale versione il contratto è stato scritto. Il linguaggio offre una direttiva per il compilatore nota come pragma di versione che indica quale specifica di linguaggio aspettarsi. Diamo un'occhiata ad un esempio:

```
solidity del pragma ^0.4.19;
```

Le direttive pragma non sono compilate nel bytecode EVM. Vengono utilizzati solo dal compilatore per verificarne la compatibilità.
La riga dove inizia il contratto è :

```
contract Faucet {
```

La dichiarazione di un oggetto contratto è simile a quella usata per dichiarare le classi in altri linguaggi orientati agli oggetti. La definizione del contratto include tutte le linee tra le parentesi graffe ({}) che definiscono un ambito , proprio come quelle che vengono utilizzate in molti altri linguaggi di programmazione.
Successivamente, dichiariamo la prima funzione del contratto Faucet:

```
function withdraw(uint withdraw_amount) public {
```

La funzione è denominata *withdraw* e accetta un argomento intero senza segno (uint) denominato *withdraw_amount*. È dichiarata come funzione pubblica, il che significa che può essere richiamata da altri contratti. Segue la definizione della funzione, indicata tra parentesi graffe. La prima parte della funzione di prelievo stabilisce un limite ai prelievi:

```
require(withdraw_amount <= 100000000000000000);
```

Utilizza la funzione Solidity incorporata per testare una precondizione, ovvero che il valore di prelievo sia inferiore o uguale a 100.000.000.000.000.000 wei, che è l'unità di base dell'ether che equivale a 0,1 etere. Se la funzione di prelievo viene chiamata con un *withdraw_amount* maggiore di tale importo, la funzione require causerà l'interruzione e il fallimento dell'esecuzione

del contratto con un'eccezione . Questa parte del contratto è la logica principale del nostro faucet. Controlla il flusso di fondi fuori dal contratto ponendo un limite ai prelievi.

```
msg.sender.transfer(withdraw_amount);
```

L'oggetto msg è uno degli input a cui possono accedere tutti i contratti. Rappresenta la transazione che ha dato luogo all'esecuzione del contratto. L'attributo *sender* è l'indirizzo del mittente della transazione. La funzione *transfer* è una funzione integrata che trasferisce ether dal contratto corrente all'indirizzo del mittente. Leggendolo al contrario, ciò significa trasferire al mittente il messaggio che ha attivato l'esecuzione di questo contratto. La funzione di trasferimento accetta un importo come unico argomento. Passiamo il valore *withdraw_amount* che era il parametro alla funzione di prelievo dichiarata poche righe prima.
La riga successiva è la parentesi graffa di chiusura, che indica la fine della definizione della nostra funzione di ritiro.
Successivamente, dichiariamo un'altra funzione:

```
function () public payable {}
```

Questa funzione è una cosiddetta funzione di fallback o predefinita , che viene chiamata se la transazione che ha attivato il contratto non nominava nessuna delle funzioni dichiarate nel contratto, o alcuna funzione. Ecco perché è definita una funzione pubblica e pagabile, il che significa che può accettare etere nel contratto. Non fa altro che accettare l'etere, come indicato dalla definizione vuota tra parentesi graffe (**{}**). Se effettuiamo una transazione che invia ether all'indirizzo del contratto, come se fosse un portafoglio, questa funzione lo gestirà. Dalla versione di Solidity 0.6.x è stata introdotta la funzione di receive() per rendere i contratti più espliciti quando vengono chiamate le loro funzioni di fallback.[1] [11]

4.5 Compilazione con il Solidity Compiler (solc)

Gli sviluppatori di Ethereum che scrivono degli smart contract in Solidity devono eseguire il loro codice attraverso un compilatore in modo che l' Ethereum Virtual Machine (EVM) possa comprendere ed eseguire i comandi appropriati. Il compilatore Solidity converte il codice in una raccolta di istruzioni byte. L'attuale compilatore Solidity, solc, è scritto in C++.
Prima che gli smart contract possano essere implementati, devono essere compilati in bytecode per l' EVM (Ethereum Virtual Machine) . Il bytecode è l'informazione in cui viene "tradotto" il codice Solidity e contiene istruzioni binarie del computer.

In Ethereum, il bytecode è ciò che viene distribuito sulla blockchain. La compilazione genera anche una ABI (Application Binary Interface) che è un'interfaccia tra due moduli di programma, spesso tra sistemi operativi e programmi utente.

I contratti vengono compilati nel bytecode che viene eseguito sull'EVM. Questo bytecode viene distribuito sulla blockchain e salvato in un indirizzo. Viene quindi reso pubblico e disponibile per chiunque possa interagire. Un indirizzo identifica l'istanza del contratto distribuito, che

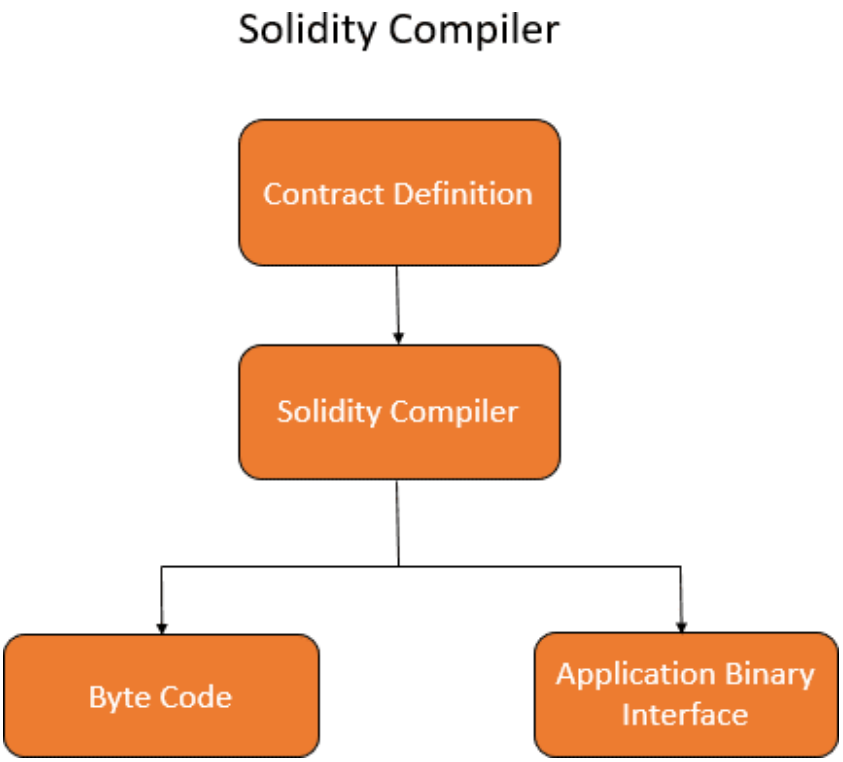

Figura 28: Rappresentazione grafica del Solidity Compiler
[31]

contiene anche un'archiviazione persistente per mantenere lo stato interno del contratto.

Ora utilizzeremo il compilatore Solidity sulla riga di comando in un sistema operativo Linux per fare una prova per compilare direttamente il nostro contratto.
Usiamo gli argomenti –bin e –optimize di solc per produrre un binario ottimizzato del nostro contratto di esempio:

```
$ solc --ottimizza --bin Faucet.sol
======= Faucet.sol:Faucet =======
Binario:
6060604052341561000 f57600080 fd5b60cf8061001d6000396000
f3006060604052600436106 03e5763ffffffff7c01000000000000
0000000000000000000000000000000000000000000600035041663
2e1a7d4d81146040575b005b3415604a57600080fd5b603e600435
67016345785d8a00008111156063576000080fd5b73ffffffffffff
ffffffffffffffffffffffffffffffff3316811561 08fc02826040
5160006040518083818858888 f19350505050151560 a057600080
fd5b505600a165627a7a723058203556d79355f2da19e773a9551e
```

```
95f1ca7457f2b5fbbf4eacf7748ab59d2532130029
```

Il risultato prodotto da solc è un binario serializzato esadecimale che può essere inviato alla blockchain di Ethereum.[1]

4.6 Application Binary Interface

Un ABI (Application Binary Interface) definisce il modo in cui si accede alle strutture e alle funzioni dei dati nel codice macchina . L'ABI è quindi il modo principale di codificare e decodificare i dati dentro e fuori il codice macchina. In Ethereum, l'ABI viene utilizzato per codificare le chiamate di contratto per l'EVM e per leggere i dati dalle transazioni. Lo scopo di un ABI è definire le funzioni nel contratto che possono essere invocate e descrivere come ciascuna funzione accetterà argomenti e restituirà il proprio risultato. L'ABI di un contratto è specificato come un array JSON di descrizioni di funzioni ed eventi. Una descrizione di funzione è un oggetto JSON con campi *type*, *name*, *inputs*, *outputs*, *constant* e *payable*. Un oggetto evento dispone dei campi *type*, *name*, *inputs* e *anonymous*.
Utilizziamo il compilatore Solidity da riga di comando solc per produrre l'ABI per il nostro contratto di esempio Faucet.sol :

```
$ solc --abi Rubinetto.sol
======= Faucet.sol:Faucet =======
Contratto JSON ABI
[{"costante":false,"inputs":
    [{"nome":"prelievo importo","tipo":"uint256"],}    \
"name":"withdraw","outputs":
    [],"payable":false,"stateMutability":"nonpayable", \
"type":"function"},{"payable":true,"stateMutability":"payable", \
"tipo":"fallback"}]
```

Come si può vedere, il compilatore produce un array JSON che descrive le due funzioni definite da Faucet.sol . Questo JSON può essere utilizzato da qualsiasi applicazione che desideri accedere al contratto Faucet una volta distribuito.[1]

4.7 La macchina virtuale di Ethereum

Al centro del protocollo e del funzionamento di Ethereum c'è la Ethereum Virtual Machine, o EVM in breve. In questa sezione esamineremo in dettaglio l'EVM, compreso il set di istruzioni, la struttura e il funzionamento, facendo alcuni esempi di reversing del codice.
L'EVM (Ethereum Virtual Machine) è la componente di Ethereum responsabile della gestione dell'implementazione e dell'esecuzione degli smart contract. In pratica, le transazioni di trasferimento di valore semplici tra account esterni (EOA) non convergono direttamente nell' EVM, ma ogni altra attività comporta un aggiornamento dello stato che viene prima calcolato dall'EVM. A

un livello più astratto, l'EVM in esecuzione sulla blockchain di Ethereum può essere paragonato a un computer globale decentralizzato, composto da milioni di programmi eseguibili, ciascuno con il proprio archivio dati permanente.
L'EVM è una macchina quasi Turing completa, con il termine "quasi" che riflette il fatto che le esecuzioni sono limitate a un numero finito di passaggi computazionali, determinato dalla quantità di gas disponibile per ciascuna esecuzione di un contratto intelligente. Questo limite risolve il problema dell'arresto, evitando situazioni in cui un'attività potrebbe protrarsi all'infinito,e causare un'interruzione dell'intera piattaforma Ethereum, sia per problemi accidentali che intenzionali.

Dal punto di vista architetturale, l'EVM utilizza uno stack per immagazzinare i valori in memoria. Lavora con parole di 256 bit, principalmente per agevolare operazioni native come l'hashing e la crittografia a curva ellittica. La memoria è così strutturata:

1. Una ROM (Read-Only Memory) contenente il codice del programma immutabile, caricato con il bytecode dello smart contract da eseguire.
2. Una memoria volatile, in cui ogni posizione è esplicitamente inizializzata a zero.
3. Uno spazio di archiviazione permanente, che fa parte dello stato di Ethereum ed è anch'esso inizializzato a zero.

Durante l'esecuzione, sono disponibili diverse variabili e dati ambientali che saranno esaminati in dettaglio in seguito in questo capitolo.La figura qui di seguito mostra l'architettura e il contesto di esecuzione dell'Ethereum Virtual Machine.29

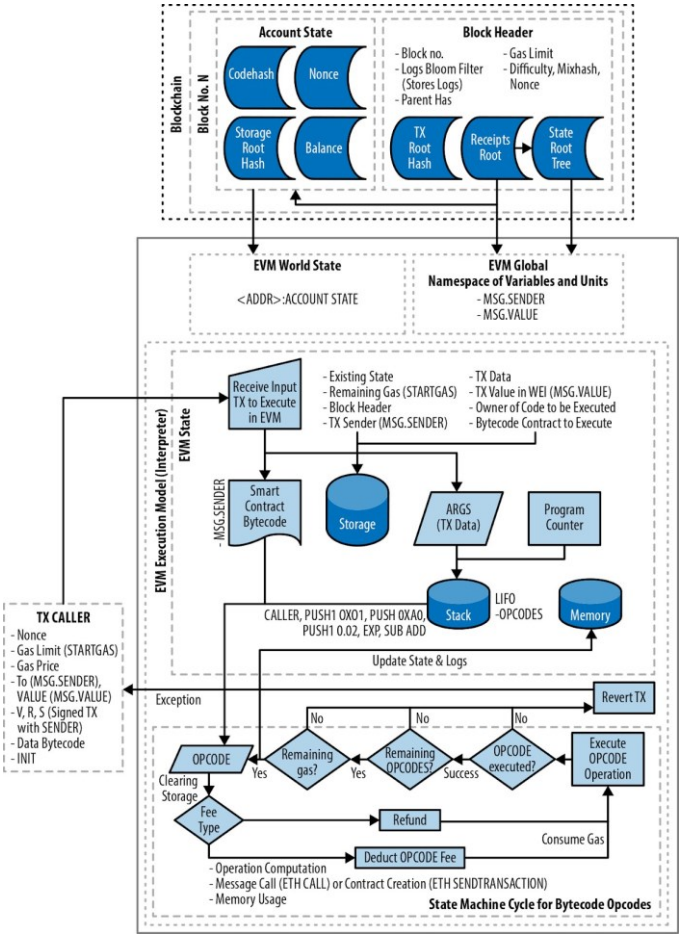

Figura 29: Architettura e contesto di esecuzione della Ethereum Virtual Machine (EVM). [1]

Confronto con alcune tecnologie esistenti Il termine ”macchina virtuale” viene spesso applicato alla virtualizzazione di un computer reale, tipicamente tramite un ”hypervisor” come VirtualBox o QEMU, o di un’intera istanza del sistema operativo, come KVM di Linux. Questi devono fornire un’astrazione software, rispettivamente, dell’hardware reale, delle chiamate di sistema e di altre funzionalità del kernel.
L’EVM opera in un dominio molto più limitato: è solo un motore di calcolo e come tale fornisce un’astrazione di semplice calcolo e archiviazione, simile ad esempio alla specifica Java Virtual Machine (JVM). Da un punto di vista di alto livello, la JVM è progettata per fornire un ambiente runtime indipendente dal sistema operativo host o dall’hardware sottostante, consentendo la compatibilità tra un’ampia varietà di sistemi. I linguaggi di programmazione di alto livello come Java o Scala (che utilizzano JVM) o C# (che utilizza .NET) vengono compilati nel set di istruzioni bytecode della rispettiva macchina virtuale. Allo stesso modo, l’EVM esegue il proprio set di istruzioni bytecode (descritto nella sezione successiva), in cui sono compilati linguaggi di programmazione di contratti intelligenti di livello superiore come LLL, Serpent, Mutan o Solidity.

L'EVM, quindi, non ha capacità di pianificazione, perché l'ordine di esecuzione è organizzato esternamente ad esso: i client Ethereum eseguono transazioni a blocchi verificate per determinare quali contratti intelligenti devono essere eseguiti e in quale ordine. In questo senso, il computer mondiale di Ethereum è a thread singolo, come JavaScript. L'EVM non dispone nemmeno di alcuna "interfaccia di sistema" o di "supporto hardware", perchè non esiste una macchina fisica con cui interfacciarsi. Il computer mondiale di Ethereum è completamente virtuale.[1]

4.8 Il set di istruzioni EVM (operazioni bytecode)

Il set di istruzioni dell'EVM offre la maggior parte delle operazioni che ci si potrebbe aspettare, tra cui:

- Operazioni aritmetiche e operazioni logiche bit a bit;
- Richieste relative al contesto di esecuzione;
- Accesso allo stack, alla memoria e all'archiviazione;
- Controllo delle operazioni di flusso;
- Registrazione, chiamate e altri operatori;

Oltre alle tipiche operazioni bytecode, l'EVM ha anche accesso alle informazioni relative all'account, come l'indirizzo e il saldo, e alle informazioni relative al blocco, come il numero di blocco e il prezzo attuale del gas.
Iniziamo la nostra esplorazione più approfondita dell'EVM esaminando i codici operativi disponibili e le relative funzionalità. Come ci si potrebbe aspettare, tutti gli operandi vengono prelevati dallo stack e il risultato, quando applicabile, viene spesso posto in cima allo stack.

I codici operativi disponibili possono essere suddivisi nelle seguenti categorie:

ADD	Add the top two stack items
MUL	Multiply the top two stack items
SUB	Subtract the top two stack items
DIV	Integer division
SDIV	Signed integer division
MOD	Modulo (remainder) operation
SMOD	Signed modulo operation
ADDMOD	Addition modulo any number
MULMOD	Multiplication modulo any number
EXP	Exponential operation
SIGNEXTEND	Extend the length of a two's complement signed integer
SHA3	Compute the Keccak-256 hash of a block of memory

Operazioni sullo stack Istruzioni per la gestione dello stack, della memoria e dell'archiviazione:

POP	Remove the top item from the
MLOAD	Load a word from memory
MSTORE	Save a word to memory
MSTORE8	Save a byte to memory
SLOAD	Load a word from storage
SSTORE	Save a word to storage
MSIZE	Get the size of the active memory in bytes
PUSHx	Place x byte item on the stack, where x can be any integer from
DUPx	Duplicate the x-th stack item, where x can be any integer from
SWAPx	Exchange 1st and (x+1)-th stack items, where x can be any

Operazioni del flusso di processo Istruzioni per il flusso di controllo:

STOP	Halt execution
JUMP	Set the program counter to any value
JUMPI	Conditionally alter the program counte
PC	Get the value of the program counter (prior to the increment
JUMPDEST	Mark a valid destination for jumps

Operazioni di sistema Codici operativi utilizzati per il sistema che esegue il programma:

LOGx	Append a log record with x topics, where x is any integer
CREATE	Create a new account with associated code
CALL	Message-call into another account, i.e. run another
CALLCODE	Message-call into this account with another
RETURN	Halt execution and return output data
DELEGATECALL	Message-call into this account with an alternative
STATICCALL	Static message-call into an account
REVERT	Halt execution, reverting state changes but returning
INVALID	The designated invalid instruction
SELFDESTRUCT	Halt execution and register account for deletion

Operazioni logiche Codici operativi che gestiscono confronti e logica bit a bit:

LT	Less-than comparison
GT	Greater-than comparison
SLT	Signed less-than comparison
SGT	Signed greater-than comparison
EQ	Equality comparison
ISZERO	Simple NOT operator
AND	Bitwise AND operation
OR	Bitwise OR operation
XOR	Bitwise XOR operation
NOT	Bitwise NOT operation
BYTE	Retrieve a single byte from a full-width 256-bit word

Operazioni ambientali Codici operativi che gestiscono le informazioni sull'ambiente di esecuzione:

GAS	Get the amount of available gas (after the reduction for
ADDRESS	Get the address of the currently executing account
BALANCE	Get the account balance of any given account
ORIGIN	Get the address of the EOA that initiated this EVM
CALLER	Get the address of the caller immediately responsible
CALLVALUE	Get the ether amount deposited by the caller responsible
CALLDATALOAD	Get the input data sent by the caller responsible for
CALLDATASIZE	Get the size of the input data
CALLDATACOPY	Copy the input data to memory
CODESIZE	Get the size of code running in the current environment
CODECOPY	Copy the code running in the current environment to
GASPRICE	Get the gas price specified by the originating
EXTCODESIZE	Get the size of any account's code
EXTCODECOPY	Copy any account's code to memory
RETURNDATASIZE	Get the size of the output data from the previous call
RETURNDATACOPY	Copy data output from the previous call to memory

Operazioni di blocco Codici operativi utilizzati per accedere alle informazioni sul blocco corrente:

BLOCKHASH	Get the hash of one of the 256 most recently completed
COINBASE	Get the block's beneficiary address for the block reward
TIMESTAMP	Get the block's timestamp
NUMBER	Get the block's number
DIFFICULTY	Get the block's difficulty
GASLIMIT	Get the block's gas limit

[1]

4.9 Stato di Ethereum

Il compito dell' EVM è quello di aggiornare lo stato di Ethereum mediante il calcolo delle transizioni di stato valide, come definito dal protocollo Ethereum, in seguito all'esecuzione del codice degli Smart Contract. Questo concetto riflette l'idea di Ethereum come una macchina a stati basata sulle transazioni dove gli account esterni, come i titolari di conti e i miner, avviano le transizioni di stato attraverso la creazione, l'accettazione e l'ordinamento delle transazioni. Prima di proseguire, è utile comprendere cosa costituisce lo stato di Ethereum.

Al livello più alto, troviamo lo "stato globale" di Ethereum, una mappatura degli indirizzi Ethereum (di 160 bit) verso tutti gli account. Nel dettaglio ogni indirizzo Ethereum viene rappresentato con:

- Un account, il quale include un saldo di ether, memorizzato come il numero di wei posseduti dall'account;
- Un nonce che rappresenta il numero di transazioni inviate con successo da quell'account se si tratta di un EOA, o il numero di contratti creati da quell'account se è un account di contratto;
- Lo "storage" dell'account è un archivio dati permanente utilizzato solo dagli smart contract) e il codice del programma dell'account (presente solo se l'account è uno smart contract.

È importante notare che un EOA non contiene sempre del codice e ha uno spazio di archiviazione vuoto.

Quando una transazione dà come risultato l'avvio del codice di uno smart contract, viene istanziata una EVM con tutte le informazioni richieste in relazione allo stato del blocco corrente creato e alla transazione specifica in fase di elaborazione. In questo processo, la ROM dell'EVM viene caricata con il codice dell'account chiamato, Il contatore del programma viene inizializzato a zero, lo storage viene aggiornato con lo storage della memoria dell'account, la memoria viene impostata su zeri e tutti i blocchi e le variabili di ambiente vengono inizializzati. Un parametro chiave è la fornitura di gas per questa esecuzione, che è inizializzata con la quantità di gas pagata dall'invio della transazione. Man mano che l'esecuzione del codice procede, la fornitura di gas viene ridotta in base al costo delle operazioni eseguite. Se in qualsiasi momento la fornitura di gas si esaurisce, scaturisce un'eccezione "Out of Gas" (OOG), causando l'interruzione immediata dell'esecuzione e l'abbandono della transazione. In tal caso, non si verifica alcuna modifica allo stato di Ethereum, ad eccezione del nonce del mittente, che viene incrementato, e del suo saldo ether che diminuisce per pagare il beneficiario del blocco per le risorse utilizzate per eseguire il codice fino al punto di arresto. Da notare che poiché uno Smart Contract può avviare transazioni, l'esecuzione del codice è un processo ricorsivo. Un contratto può chiamare altri contratti, e ogni chiamata comporta l'avvio di un nuovo processo ad un'altra EVM.

Compilazione di Solidity in EVM Bytecode

La compilazione di un file sorgente Solidity nel bytecode EVM può essere eseguita tramite diversi metodi. In questo capitolo utilizzeremo l'eseguibile solc che abbiamo presentato precedentemente dalla riga di comando.
La generazione del flusso di codice operativo, non elaborato di un file sorgente Solidity, è facilmente ottenibile con l'opzione della riga di comando –opcodes.. Ad esempio, la compilazione di un file Solidity di esempio, Example.sol, e l'invio dell'output del codice operativo in una directory denominata BytecodeDir viene eseguita con il seguente comando:

```
$ solc -o BytecodeDir --opcodes Example.sol
```

Il seguente comando produrrà il bytecode binario per il nostro programma di esempio:

```
$ solc -o BytecodeDir --bin Example.sol
```

I file di codice operativo di output generati dipenderanno dai contratti specifici contenuti nel file sorgente di Solidity. Il nostro semplice file Solidity Example.sol ha un solo contratto, denominato example:

```
pragma solidity ^0.4.19;

contract example {

  address contractOwner;

  function example() {
    contractOwner = msg.sender;
  }
}
```

Come si può vedere, tutto ciò che fa questo contratto è contenere una variabile di stato persistente, che viene impostata come indirizzo dell'ultimo account per eseguire questo contratto.
Una volta eseguito il comando verrà generato nella directory BytecodeDir il file opcode, example.opcode , che contiene le istruzioni del codice operativo EVM del contratto example. L'apertura del file example.opcode in un editor di testo mostrerà quanto segue:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO
PUSH1 0xE JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST CALLER PUSH1 0x0 DUP1 PUSH2 0x100
EXP DUP2 SLOAD DUP2 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF MUL NOT
AND SWAP1 DUP4 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND MUL OR
SWAP1 SSTORE POP PUSH1
0x35 DUP1 PUSH1 0x5B PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN
STOP PUSH1 0x60 PUSH1
0x40 MSTORE PUSH1 0x0 DUP1 REVERT STOP LOG1
```

```
PUSH6 0x627A7A723058 KECCAK256 JUMP
0xb9 SWAP14 0xcb 0x1e 0xdd RETURNDATACOPY
0xec 0xe0 0x1f 0x27 0xc9 PUSH5
0x9C5ABCC14A NUMBER 0x5e INVALID EXTCODESIZE
0xdb 0xcf EXTCODESIZE 0x27
EXTCODESIZE 0xe2 0xb8 SWAP10 0xed 0x
```

Per interpretare quanto sta succedendo nel nostro programma in dettaglio possiamo utilizzare l'elenco dei codici operativi fornito in " The EVM Instruction Set (Bytecode Operations)"e confrontarlo con il nostro risultato . Tuttavia,questo lavoro risulta piuttosto oneroso anche con un programma cos`ı semplice quindi iniziamo esaminando le prime quattro istruzioni:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE
```

Qui abbiamo PUSH1 seguito da un byte di valore 0x60. Questa istruzione EVM prende il singolo byte che segue l'opcode nel codice del programma, come valore letterale, e lo inserisce nello stack. È possibile inserire nello stack valori di dimensione fino a 32 byte, come in:

```
PUSH32 0x436f6e67726174756c6174696f6e732120536f6f ...
```

Il secondo codice operativo PUSH1, da example.opcode, memorizza 0x40 in cima allo stack, spingendo lo 0x60 già presente di uno slot.
Il prossimo è MSTORE, che è un'operazione di archiviazione della memoria che salva un valore nella memoria dell'EVM. Richiede due argomenti e, come la maggior parte delle operazioni EVM, li ottiene dallo stack. Per ogni argomento lo stack viene "estratto". Il valore in cima allo stack viene tolto e tutti gli altri valori sullo stack vengono spostati di una posizione verso l'alto. Il primo argomento per MSTORE è l'indirizzo della parola in memoria dove verrà inserito il valore da salvare. Per questo programma abbiamo 0x40 in cima allo stack, quindi viene rimosso dallo stack e utilizzato come indirizzo di memoria. Il secondo argomento è il valore da salvare, che qui è 0x60. Dopo l'esecuzione dell'operazione MSTORE il nostro stack è di nuovo vuoto, ma abbiamo il valore 0x60 (96 in decimale) nella posizione di memoria 0x40.
Il successivo codice operativo è CALLVALUE, che è un codice operativo ambientale che spinge in cima allo stack la quantità di ether, misurata in wei, inviata con la chiamata del messaggio che ha avviato questa esecuzione.

Potremmo continuare a seguire questo programma in questo modo finché non avremo una piena comprensione dei cambiamenti di stato di basso livello che questo codice effettua, ma non ci aiuterebbe in questa fase. Torneremo su questo argomento più avanti nel capitolo.

Codice di distribuzione del contratto

Esiste una distinzione tra il codice utilizzato durante la creazione e l'implementazione di un nuovo contratto sulla piattaforma Ethereum e il codice del contratto stesso. Per la creazione di un nuovo contratto, è richiesta una transazione speciale con il campo "indirizzo" impostato

su un valore speciale, 0x0, e il campo ”dati” impostato al codice iniziale del contratto. Quando una transazione di creazione di un contratto di questo tipo viene elaborata, il codice per il nuovo conto contrattuale non proviene dal campo ”dati” della transazione stessa, ma, viene istanziata una EVM con il codice presente nel campo ”dati” della transazione, che viene caricato nella ROM del codice del programma. L’output dell’esecuzione di questo codice di distribuzione viene poi utilizzato come il codice per il nuovo account contrattuale. Questo permette ai nuovi contratti di essere inizializzati a livello di codice, utilizzando lo stato di Ethereum al momento dell’implementazione, consentendo l’impostazione di valori nella memoria del contratto, l’invio di ether o la creazione di ulteriori nuovi contratti.
Quando si compila un contratto offline, ad esempio utilizzando il compilatore solc dalla riga di comando, è possibile ottenere il ”bytecode di distribuzione” o il ”bytecode di runtime”.
Il ”bytecode di distribuzione” è utilizzato per gestire ogni aspetto dell’inizializzazione di un nuovo account contrattuale, compreso il bytecode che verrà effettivamente eseguito quando le transazioni richiameranno il nuovo contratto, cioè il ”bytecode di runtime”, e il codice necessario per inizializzare tutto in base al costruttore del contratto.
Il ”bytecode di runtime”, invece, rappresenta esattamente il codice che verrà eseguito quando il nuovo contratto verrà chiamato. In quanto non include il codice necessario per l’inizializzazione del contratto durante la distribuzione.

Per comprendere meglio questo concetto, prendiamo come esempio il semplice contratto Faucet.sol che abbiamo creato ad inizio capitolo:

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.4.19;

// Our first contract is a faucet!
contract Faucet {

  // Give out ether to anyone who asks
  function withdraw(uint withdraw_amount) public {

      // Limit withdrawal amount
      require(withdraw_amount <= 100000000000000000);

      // Send the amount to the address that requested it
      msg.sender.transfer(withdraw_amount);
    }

  // Accept any incoming amount
  function() public payable {}

}
```

Per ottenere il bytecode di distribuzione, eseguiremo solc –bin Faucet.sol. Se invece volessimo solo il bytecode di runtime, eseguiremo solc –bin-runtime Faucet.sol.
Se si confronta l’output di questi comandi, si vedrà che il bytecode di runtime è un sottoinsieme del bytecode di distribuzione. In altre parole, il bytecode di runtime è interamente contenuto nel

bytecode di distribuzione.[1]

4.10 Disassemblare il Bytecode

Disassemblare il bytecode EVM è un ottimo modo per capire come agisce Solidity di alto livello nell'EVM. Ci sono alcuni disassemblatori che si possonoi usare per fare questo.
Ora facciamo un esempio utilizzando Ethersplay ossia un plug-in EVM per Binary Ninja.
Dopo aver ottenuto il bytecode di runtime di Faucet.sol , può essere inserito in Binary Ninja, dopo aver caricato il plug-in Ethersplay per vedere come appaiono le istruzioni EVM.

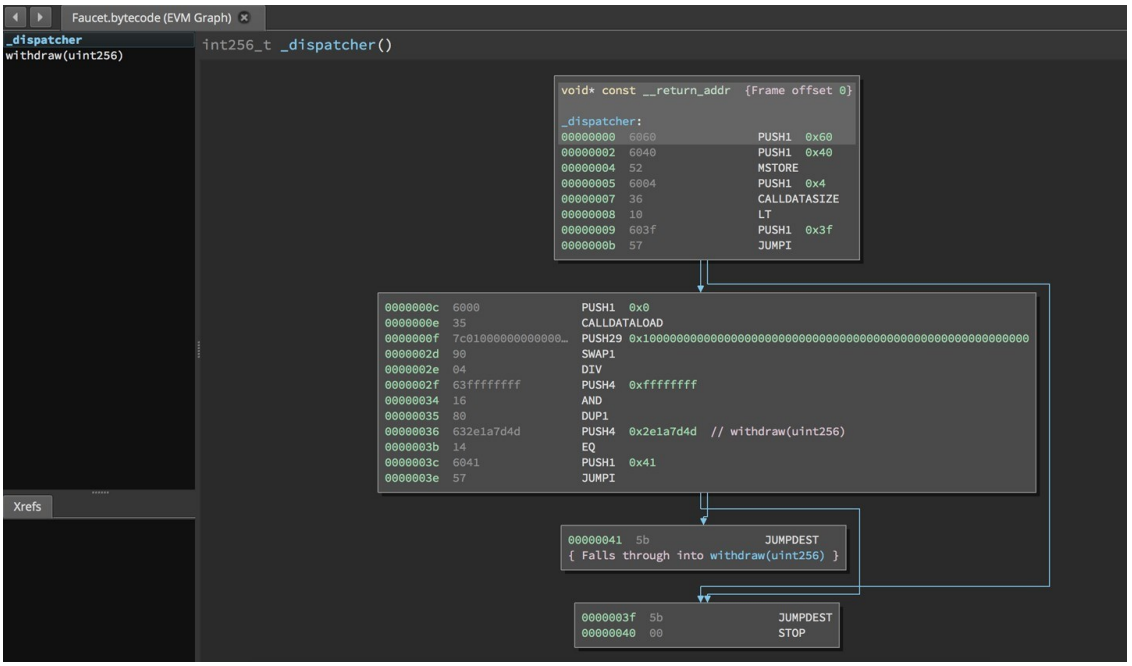

Figura 30: Screenshot di Binary Ninja

Quando si invia una transazione ad uno smart contract compatibile con ABI, la transazione interagisce prima con il dispatcher di quello smart contract . Il dispatcher legge il campo dati della transazione e invia la parte rilevante all'apposita funzione. Possiamo vedere un esempio di dispatcher all'inizio del nostro bytecode runtime Faucet.sol disassemblato. Dopo l'istruzione di archiviazione della memoria vista precedentemente MSTORE, vediamo le seguenti istruzioni:

PUSH1 0 x4
CALLDATASIZE
LT
PUSH1 0x3f
JUMPI

Come abbiamo visto, PUSH1 0x4 posiziona 0x4 in cima allo stack che altrimenti sarebbe vuoto. CALLDATASIZE ottiene la dimensione in byte dei dati inviati con la transazione "call-data" e inserisce quel numero nello stack. Dopo che queste operazioni sono state eseguite, lo stack appare cos`ı:

STACK
0x4
length of calldata from tx

L'istruzione successiva è LT, abbreviazione di "less than". La quale controlla se l'elemento in cima allo stack è inferiore all'elemento successivo nello stack. Nel nostro caso, controlla se il risultato di CALLDATASIZE è inferiore a 4 byte.
Infatti, ciascuna funzione è identificata dai primi 4 byte del suo hash Keccak-256. Inserendo il nome della funzione e gli argomenti che richiede una funzione hash keccak256, possiamo dedurre il suo identificatore di funzione. Nel nostro caso, abbiamo:

```
keccak256("withdraw(uint256)") = 0x2e1a7d4d ...
```

Pertanto, l'identificatore della funzione withdraw(uint256) è 0x2e1a7d4d, poiché questi sono i primi 4 byte dell'hash risultante. Un identificatore di funzione è sempre lungo 4 byte, quindi se l'intero campo dati della transazione inviata al contratto è inferiore a 4 byte, non esiste alcuna funzione con cui la transazione potrebbe comunicare, a meno che non venga definita una funzione di fallback . Poiché abbiamo implementato tale funzione di fallback in Faucet.sol , l'EVM passa a questa funzione quando la lunghezza della calldata è inferiore a 4 byte.
LT estrae i primi due valori dallo stack e, se il campo dati della transazione è inferiore a 4 byte, vi inserisce 1. Altrimenti, inserisce 0. Nel nostro esempio, supponiamo che il campo dati della transazione inviata al nostro contratto fosse inferiore a 4 byte.

L'istruzione PUSH1 0x3f inserisce il byte 0x3f nello stack. Dopo questa istruzione, lo stack appare cos`ı:

STACK
1
0x3f

L'istruzione successiva è JUMPI, che corrisponde a "jump if". Funziona così:

```
jumpi(label, cond) // Jump to "label" if "cond" is true
```

Nel nostro caso, l'etichetta è 0x3f, che è dove risiede la nostra funzione di fallback nello smart contract. L'argomento cond è 1, che era il risultato dell'istruzione LT precedente. Per esprimere a parole l'intera sequenza, il contratto passa alla funzione di fallback se i dati della transazione sono inferiori a 4 byte.

A 0x3f segue solo un'istruzione STOP perché, sebbene abbiamo dichiarato una funzione di fallback, l'abbiamo mantenuta vuota.

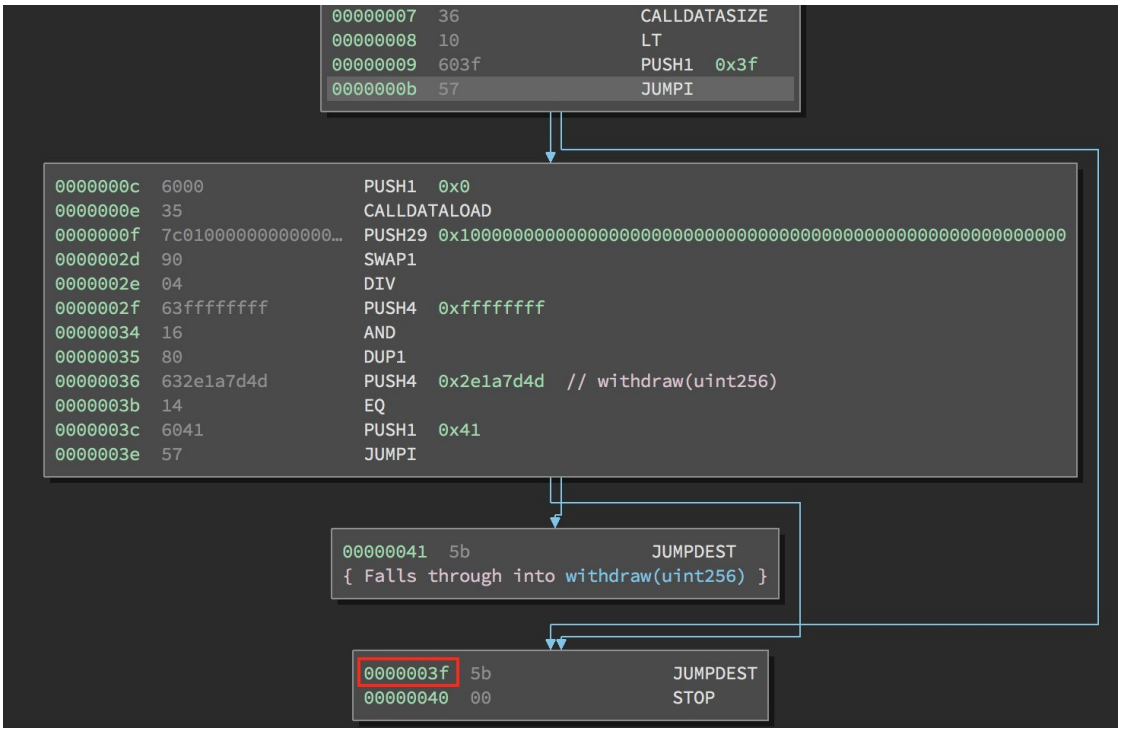

Figura 31: Screenshot di Binary Ninja

Esaminiamo il blocco centrale del dispatcher. Supponendo di aver ricevuto dati di chiamata di lunghezza superiore a 4 byte, l'istruzione JUMPI non passerebbe alla funzione di fallback. Invece, l'esecuzione del codice procederebbe con le seguenti istruzioni:

```
PUSH1 0 x0
CALLDATALOAD
PUSH29 0 x1000000 . . .
SWAP1
DIV
PUSH4 0 x f f f f f f f f
AND
DUP1
PUSH4 0x2e1a7d4d
EQ
PUSH1 0 x41
JUMPI
```

PUSH1 0x0 inserisce 0 nello stack, che altrimenti sarebbe di nuovo vuoto. CALLDATALOAD accetta come argomento un indice all'interno dei calldata inviati allo smart contract e legge 32 byte da quell'indice, in questo modo:

```
calldataload(p) //load 32 bytes of calldata starting from byte position p
```

Poiché 0 era l'indice passato dal comando PUSH1 0x0, CALLDATALOAD legge 32 byte di calldata a partire dal byte 0, quindi li inserisce in cima allo stack (dopo aver estratto lo 0x0 originale). Dopo l'istruzione PUSH29 0x1000000..., lo stack è quindi:

STACK
32 bytes of calldata starting at byte 0
0x1000000... (29 bytes in length)

SWAP1 scambia l'elemento in cima allo stack con l' elemento i -esimo dopo di esso. In questo caso, scambia 0x1000000... con calldata. Il nuovo stack è:

STACK
0x1000000... (29 bytes in length)
32 bytes of calldata starting at byte 0

L'istruzione successiva è DIV, che funziona come segue:

```
div(x, y) // integer division x / y
```

Il risultato del DIV (l'identificatore della funzione) viene inserito nello stack e il nostro stack ora è:

STACK
function identifier sent in data

Poiché le istruzioni PUSH4 0xffffffff e AND sono ridondanti, possiamo ignorarle completamente, poiché lo stack rimarrà lo stesso dopo averle eseguite. L'istruzione DUP1 duplica il primo elemento dello stack, che è l'identificatore della funzione.*withdraw(uint256)* L'istruzione successiva, PUSH4 0x2e1a7d4d, inserisce nello stack l'identificatore di funzione precalcolato della funzione. Lo stack ora è:

STACK
0x2e1a7d4d
function identifier sent in data
function identifier sent in data

L'istruzione successiva, EQ, estrae i primi due elementi dello stack e li confronta. È qui che il dispatcher svolge il suo lavoro principale: confronta se l'identificatore della funzione inviato nel campo msg.data della transazione corrisponde a quello di *withdraw(uint256)*. Se sono uguali, EQ inserisce 1 nello stack, che verrà infine utilizzato per eseguire la funzione di prelievo. Altrimenti, EQ inserisce 0 nello stack.
Supponendo che la transazione inviata al nostro contratto sia effettivamente iniziata con l'identificatore di funzione per ritiro(uint256), il nostro stack è diventato:

STACK
function identifier sent in data (now known to be 0x2e1a7d4d)
1

Successivamente, abbiamo PUSH1 0x41, che è l'indirizzo in cui risiede la funzione di withdraw(uint256) nel contratto. Dopo questa istruzione, lo stack appare cos`ı:

STACK
function identifier sent in msg.data
0x41
1

Segue l'istruzione JUMPI, che accetta ancora una volta i primi due elementi dello stack come argomenti. In questo caso, abbiamo jumpi(0x41, 1), che dice all'EVM di eseguire il salto alla posizione della funzione withdraw(uint256) e l'esecuzione del codice di quella funzione può procedere.[1]

Capitolo 5

Sicurezza negli Smart contract

In questo capitolo, delineeremo le principali categorie di vulnerabilità che, negli ultimi anni, hanno costituito il vettore d'attacco predominante per gli smart contract scritti in Solidity. Inoltre, forniremo esempi di codice vulnerabile relativi a queste categorie e, in alcuni casi, presenteremo casi concreti in cui tale codice è stato sfruttato per attacchi sulla blockchain di Ethereum. L'obiettivo di questa sezione è fornire una panoramica completa di queste tipologie di attacco, che successivamente verranno utilizzate come base per testare i nostri strumenti automatici di analisi.

La sicurezza è una degli aspetti più importanti da considerare quando si scrivono smart contract. Nel campo della programmazione degli smart contract, gli errori sono facilmente sfruttabili e possono portare a costi indotti. In questo capitolo esamineremo dei tipici pattern di progettazione vulnerabili, ovvero le pratiche e i pattern che possono introdurre vulnerabilità negli smart contract per i quali andremo a testare successivamente gli strumenti di analisi automatica.
Come per gli altri programmi, uno smart contract eseguirà esattamente ciò che è stato scritto, il che non è sempre quello che il programmatore aveva progettato. Inoltre, tutti gli smart contract sono pubblici e qualsiasi utente può interagire con essi semplicemente creando una transazione. Qualsiasi vulnerabilità può essere sfruttata e le perdite sono quasi sempre impossibili da recuperare. È quindi fondamentale seguire le migliori pratiche e utilizzare pattern di progettazione ben testati.

5.1 Reentrancy

Una delle caratteristiche degli smart contract su Ethereum è la loro capacità di invocare e interagire con il codice di altri contratti esterni. Inoltre, i contratti spesso gestiscono anche ether, che

possono essere inviati a terzi. Tali operazioni richiedono che i contratti siano in grado di effettuare chiamate esterne. Tuttavia, queste chiamate esterne possono essere manipolate da aggressori che cercano di indirizzare i contratti a eseguire altro codice. Uno di questi passa attraverso una funzione di fallback, la quale può provare un ciclo di chiamate verso se stessi.

La vulnerabilità

Questo tipo di attacco può verificarsi quando un contratto invia ether a un indirizzo sconosciuto. Un utente malintenzionato può costruire con attenzione un contratto con un indirizzo esterno contenente un codice dannoso nella funzione di fallback. Pertanto, quando un contratto invia ether a questo indirizzo, invocherà il codice dannoso. In genere il codice dannoso esegue una funzione sul contratto vulnerabile, effettuando operazioni non previste dallo sviluppatore. Il termine ”reentrancy” deriva dal fatto che il contratto dell’attaccante esterno chiama una funzione sul contratto vulnerabile e il percorso di esecuzione del codice ” rientra ” in esso.

Per chiarire questo, si consideri il semplice contratto vulnerabile in EtherStore.sol , che funge da deposito di Ethereum e che consente ai depositanti di prelevare solo 1 ether a settimana.

```
contract EtherStore {

    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

Questo contratto ha due funzioni pubbliche, depositFunds e withdrawFunds. La funzione depositFunds incrementa semplicemente il saldo del mittente. La funzione withdrawFunds consente al mittente di specificare la quantità di wei da prelevare. Questa funzione è destinata ad avere successo solo se l’importo richiesto da prelevare è inferiore ad 1 ether e non si è verificato un prelievo nell’ultima settimana.
La vulnerabilità è nella riga 17, dove il contratto invia all’utente la quantità di ether richiesta.

Consideriamo ora un utente malintenzionato che ha creato il contratto in Attack.sol .

```
import "EtherStore.sol";

contract Attack {
  EtherStore public etherStore;

  // intialize the etherStore variable with the contract address
  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }

  function attackEtherStore() public payable {
      // attack to the nearest ether
      require(msg.value >= 1 ether);
      // send eth to the depositFunds() function
      etherStore.depositFunds.value(1 ether)();
      // start the magic
      etherStore.withdrawFunds(1 ether);
  }

  function collectEther() public {
      msg.sender.transfer(this.balance);
  }

  // fallback function
  function () payable {
      if (etherStore.balance > 1 ether) {
          etherStore.withdrawFunds(1 ether);
      }
  }
}
```

Analizziamo ora il modo in cui si verifica l'exploit, innanzitutto l'attaccante crea il contratto dannoso (diciamo all'indirizzo 0x0...123) con l'indirizzo del contratto di EtherStore come unico parametro del constructor. In questo modo si inizializza la variabile etherStore all'indirizzo del contratto da attaccare.

L'attaccante chiamerebbe quindi la funzione attackEtherStore, con una quantità di ether maggiore o uguale a 1, ad esempio 1 ether. In questo esempio, assumiamo anche che un numero di altri utenti abbia depositato ether in questo contratto, in modo tale che il suo saldo attuale sia 10 ether. L'attacco quindi produrrà il seguente risultato:

1. Attack.sol , riga 15: La funzione depositFunds del contratto EtherStore sarà chiamata con un msg.value di 1 ether (e molto gas). Il mittente (msg.sender) sarà il contratto dell'attaccante (0x0...123). In questo modo il risultato sarà, balances[0x0..123] = 1 ether.

2. Attack.sol , riga 17: il contratto dell'attaccante chiamerà quindi la funzione withdrawFunds del contratto EtherStore con un parametro di 1 ether. Ciò soddisfa tutti i requisiti (righe 12-16 del contratto EtherStore) poiché non sono stati effettuati prelievi precedenti.
3. EtherStore.sol , riga 17: il contratto spedirà 1 ether al contratto attaccante.
4. Attack.sol , riga 25: il pagamento al contratto attaccante eseguirà quindi la funzione di fallback.
5. Attack.sol , riga 26: il saldo totale del contratto EtherStore era 10 ether ed è ora aggiornato a 9 ether, soddisfando cos`ı il requisito dell'if.
6. Attack.sol , riga 27: la funzione fallback richiama nuovamente la funzione withdrawFunds del contratto EtherStore " rientrando " quindi nel contratto EtherStore
7. EtherStore.sol , riga 11: in questa seconda chiamata a withdrawFunds, il saldo del contratto di attacco è ancora 1 ether in quanto la riga 18 non è stata ancora eseguita. Quindi, abbiamo ancora balances[0x0..123] = 1 ether. Questo vale anche per la variabile lastWithdrawTime.
8. EtherStore.sol , riga 17: Il contratto attaccante ritira un altro 1 ether.
9. I passaggi 4–8 si ripetono fino a quando è verificata la condizione EtherStore.balance ¿ 1, come dettato dalla riga 26 in Attack.sol .
10. Attack.sol , riga 26: una volta che nel contratto EtherStore è rimasto 1 o meno ether, questa istruzione if fallirà. Ciò consentirà poi di eseguire le righe 18 e 19 del contratto EtherStore.
11. EtherStore.sol , righe 18 e 19: le mappature balances e lastWithdrawTime verranno impostate e l'esecuzione terminerà.

Il risultato finale è che l'attaccante ha ritirato tutti gli ether tranne 1 dal contratto EtherStore in un'unica transazione

Tecniche preventive

Esistono numerose tecniche più o meno comuni che aiutano a evitare potenziali vulnerabilità di reentrancy negli smart contract. Uno di questi metodi richiede di integrare, dove fosse possibile, una funzione di trasferimento quando si inviano ether a contratti esterni. La funzione di trasferimento limita l'invio di solo 2300 gas con una chiamata esterna, il che non è sufficiente, per l'indirizzo/contratto di destinazione, per richiamare un altro contratto.

La seconda tecnica è garantire che tutta la logica che modifica le variabili di stato avvenga prima che l'ether venga inviato fuori dal contratto (o per qualsiasi chiamata esterna). Nell'esempio EtherStore, le righe 18 e 19 di EtherStore.sol devono essere inserite prima della riga 17. Una pratica consigliabile per qualsiasi codice è quella di eseguire chiamate esterne a indirizzi sconosciuti come ultima operazione.

Una terza tecnica consiste nell'introdurre un mutex, ovvero aggiungere una variabile di stato che blocca il contratto durante l'esecuzione del codice, impedendo le chiamate rientranti.

L'applicazione di queste tecniche nel precedente contratto EtherStore.sol , restituisce il seguente contratto:

```
contract EtherStore {

    // initialize the mutex
    bool reEntrancyMutex = false;
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint 256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(!reEntrancyMutex);
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
            // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        balances[msg.sender] -= weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
        // set the reEntrancy mutex before the externalcall
        reEntrancyMutex = true;
        msg.sender.transfer(_weiToWithdraw);
        //release the mutex after the external call
        reEntrancyMutex = false;
    }
}
```

Esempio reale: DAO Hack

L'attacco DAO (Organizzazione Autonoma Decentralizzata) è stato uno dei principali attacchi avvenuti nelle prime fasi dello sviluppo di Ethereum. A quel tempo, il contratto ammontava a oltre 150 milioni di dollari. Il rientro ha svolto un ruolo importante nell'attacco, che alla fine ha portato all'hard fork che ha creato Ethereum Classic (ETC).6.2[10][27][34][1]

5.2 Over/Underflow aritmetici

La macchina virtuale Ethereum specifica tipi di dati a dimensione fissa per i numeri interi. Ciò significa che una variabile intera può rappresentare solo un certo intervallo di numeri. A uint8, ad esempio, può memorizzare solo numeri nell'intervallo [0,255]. Il tentativo di archiviare 256 in a uint8 risulterà in 0. Se non si presta attenzione, le variabili in Solidity possono essere sfruttate

se l'input dell'utente non è controllato e se vengono eseguiti calcoli che danno come risultato numeri che si trovano al di fuori dell'intervallo che li memorizza.

La vulnerabilità

Un overflow/underflow si verifica quando viene eseguita un'operazione che richiede che una variabile di dimensione fissa memorizzi un numero (o una porzione di dati) che non rientra nell'intervallo del tipo di dati della variabile stessa.
Ad esempio, sottraendo 1 da una uint8 variabile (intero senza segno di 8 bit; cioè non negativo) il cui valore è 0 si otterrà il numero 255. Questo è considerato un ciclo insufficiente. Abbiamo inatti assegnato un numero inferiore all'intervallo previsto da un uint8, quindi il risultato si ripete e fornisce un numero più grande di quello che può essere memorizzato in un uint8. Allo stesso modo, l'aggiunta 2^8=256 ad a uint8 lascerà la variabile invariata, poiché abbiamo coperto l'intera lunghezza del file uint. Una possibile analogia per questo comportamento sono i contachilometri delle automobili, che misurano la distanza percorsa essi infatti si resettano a 000000, dopo che viene superato il numero più grande, ovvero 999999.

L'aggiunta di numeri maggiori dell'intervallo del tipo di dati viene chiamata overflow . Per chiarezza, aggiungendo 257 a uint8 che attualmente ha un valore 0 si otterrà il numero 1. A volte è costruttivo pensare alle variabili di dimensione fissa come se fossero cicliche, dove si ricomincia da zero se aggiungiamo numeri sopra il numero più grande possibile memorizzato, si inizia il conto alla rovescia dal numero più grande se sottraiamo da zero. Nel caso dei int tipi con segno, che possono rappresentare numeri negativi, si ricomincia una volta raggiunto il valore negativo più grande. Ad esempio, se proviamo a sottrarre 1 da a il uint8 cui valore è -128, otterremo 127.

Questi tipi di trucchi numerici consentono agli aggressori di utilizzare in modo improprio il codice e creare flussi logici inaspettati. Ad esempio, considera il contratto TimeLock in TimeLock.sol .

Example 3. TimeLock.sol

```
contract TimeLock {

    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }
```

```
    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        balances[msg.sender]=0;
        msg.sender.transfer(balance);
    }
}
```

Questo contratto è progettato per funzionare come una cassaforte a tempo: gli utenti possono depositare ether nel contratto che verrà bloccato per almeno una settimana. L'utente può estendere il tempo di attesa a più di 1 settimana se lo desidera, ma una volta depositato, l'utente può essere sicuro che il suo ether sarà bloccato in modo sicuro per almeno una settimana, come prevede questo contratto.
Nel caso in cui un utente sia costretto a consegnare la propria chiave privata, un contratto come questo potrebbe essere utile per garantire che il proprio ether non sia ottenibile per un breve periodo di tempo. Ma se un utente avesse bloccato 100 ether in questo contratto e avesse consegnato le proprie chiavi ad un possibile attaccante, l'aggressore potrebbe utilizzare un overflow per ricevere l'ether, indipendentemente dal dominio lockTime.

L'aggressore potrebbe determinare il corrente valore di lockTime dell'indirizzo di cui ora detiene la chiave .. Potrebbero quindi richiamare la funzione increaseLockTime e passare come argomento il numero 2^{256} - userLockTime. Questo numero verrebbe aggiunto a quello corrente userLockTime e causerebbe un overflow, ripristinando lockTime[msg.sender] a 0. L'aggressore potrebbe quindi semplicemente richiamare la funzione withdraw per ottenere la ricompensa.

Example 4. Underflow vulnerability example from Ethernaut challenge.

```
pragma solidity ^0.4.18;

contract Token {

  mapping(address => uint) balances;
  uint public totalSupply;

  function Token(uint _initialSupply) {
    balances[msg.sender] = totalSupply = _initialSupply;
  }

  function transfer(address _to, uint _value) public returns (bool) {
    require(balances[msg.sender] - _value >= 0);
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    return true;
  }

  function balanceOf(address _owner) public constant returns(uint balance)    {
    return balances[owner];
```

```
  }
}
```

L'errore di programmazione in questo contratto consiste nella funzione di transfer. L'istruzione require alla riga 13 può essere aggirata utilizzando un underflow. Consideriamo ora un utente con saldo pari a zero. Potrebbe quindi chiamare la funzione transfer con qualsiasi valore diverso da zero _value, e passare l'istruzione require alla riga 13. Questo perché balances[msg.sender]è 0 (e a uint256), quindi sottraendo qualsiasi importo positivo (escluso 2^{256}), come descritto in precedenza, si otterrà un numero positivo. Ciò vale anche per la riga 14, dove il saldo verrà accreditato con un numero positivo. Pertanto, in questo esempio, un utente malintenzionato può ottenere token gratuiti a causa di una vulnerabilità di underflow.

Tecniche preventive

L'attuale tecnica convenzionale per proteggersi dalle vulnerabilità di under/overflow consiste nell'utilizzare o creare librerie matematiche che sostituiscono gli operatori matematici standard di addizione, sottrazione e moltiplicazione (la divisione è esclusa poiché non causa over/underflow e l'EVM ripristina la divisione per 0).Il problema dell'overflow/underflow ha portato gli sviluppatori di solidity a sviluppare specifici accorgimenti nelle ultime versione del linguaggio.

Per dimostrare come vengono utilizzate queste librerie in Solidity, correggiamo il TimeLock contratto utilizzando la SafeMath libreria distribuita dalla comunity di OpenZeppelin. La versione senza overflow del contratto è:

```
library SafeMath {


  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0)  {
      return 0;
    }
    uint256 c = a * b;
    assert(c/a == b);
    return c;
  }


  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    //assert(a == b *c + a%b); // This holds in all cases
    return c;
  }


  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
```

```
    return a - b;
  }


  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}


contract TimeLock {
    using SafeMath for uint; // use the library for uint type
    mapping(address => uint256) public balances;
    mapping(address => uint256) public lockTime;


    function deposit() public payable {
        balances[msg.sender] = balances[msg.sender].add(msg.value);
        lockTime[msg.sender] = now.add(1 weeks);
    }


    function increaseLockTime(uint256 _secondsToIncrease) public {
        lockTime[msg.sender] = lockTime[msg.sender].add(secondsToIncrease);
    }


    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        balances[msg.sender] = 0;
        msg.sender.transfer(balance);
    }
}
```

Esempi reali: PoWHC

La Proof of Weak Hands Coin (PoWHC), originariamente concepita come una sorta di scherzo, era uno schema Ponzi creato un collettivo di Internet. Sfortunatamente sembra che gli autori del contratto non abbiano mai visto over/underflow prima, e di conseguenza 866 ether sono stati liberati dal contratto.[3] [1]

5.3 Unexpected Ether

Tipicamente, quando l'ether viene inviato ad un contratto viene eseguita la funzione di fallback o un'altra funzione definita nel contratto che permette il pagamento. Ci sono due eccezioni a questo, dove l'ether può essere presente in un contratto senza l'esecuzione di alcun codice. I contratti che non si basano sull'esecuzione su codice per cui l'ether inviato non è previsto possono essere vulnerabili ad attacchi in cui l'ether viene inviato forzatamente.

La vulnerabilità

Una tecnica di programmazione difensiva comune, utile per imporre transizioni di stato corrette o operazioni di convalida, è il controllo dell'invariante ,caratterizzato da una permanenza e una costanza di alcuni valori o proprietà. Questa tecnica prevede la definizione di un insieme di invarianti (metriche o parametri che non dovrebbero cambiare nel tempo) e il loro controllo. Un esempio di invariante è quello della totalSupply in un token ERC20 a emissione fissa e a controllo del bilancio del contratto . Poiché nessuna funzione dovrebbe modificare questa invariante, si potrebbe aggiungere un controllo alla funzione che assicuri che il bilancio rimanga invariato, per garantire che la funzione transfertotalSupply funzioni correttamente.

Tale vulnerabilità solitamente si presenta a causa di un approccio sbagliato con Solidity da parte degli sviluppatori. Infatti è un'idea sbagliata che un contratto possa accettare o ottenere ether solo tramite funzioni che accettano un pagamento. Questo malinteso può portare a contratti che contengono al loro interno false proprietà che accettano condizioni basate sull'ether, cosa che può portare a una serie di vulnerabilità. La prova decisiva di questa vulnerabilità è l'uso errato di this.balance.

Esistono due modi in cui ether può essere inviato forzatamente ad un contratto senza utilizzare una funzione payable o eseguire alcun codice su di esso:

Self-destruct/suicide

Qualsiasi contratto è in grado di implementare la funzione di selfdestruct, che rimuove tutto il bytecode dall'indirizzo del contratto e invia tutto l'ether in esso memorizzato all'indirizzo specificato dal parametro. Se anche questo indirizzo specificato è un contratto, non verrà richiamata alcuna funzione (incluso il fallback). Pertanto, la funzione di selfdestruct può essere utilizzata per inviare forzatamente ether a qualsiasi contratto e indipendentemente dal codice presente in essa, anche in assenza di funzioni payabel. Ciò significa che qualsiasi utente malintenzionato può creare un contratto con una funzione selfdestruct, inserire in essa ether, chiamare selfdestruct(target) e forzare l'invio di ether ad un contratto target.

Self-destruct/suicide

Un altro modo per inserire ether in un contratto è precaricare l'indirizzo del contratto con ether. Gli indirizzi del contratto sono deterministici: infatti, l'indirizzo viene calcolato dall'hash Keccak-256 (comunemente sinonimo di SHA-3) all'indirizzo che crea il contratto e dal nonce della

transazione che lo crea. Formalmente possiamo definire

```
address = sha3(rlp.encode([account\ address, transaction\ nonce]))
```

Ciò significa che chiunque può calcolare quale sarà l'indirizzo di un contratto prima che venga creato e inviare ether a quell'indirizzo. Quando il contratto viene creato avrà un saldo ether diverso da zero.

Esploriamo alcune insidie che possono sorgere da questo tipo di vulnerabilità. Consideriamo ora un contratto eccessivamente semplice in EtherGame.sol .

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;

    mapping(address => uint) redeemableEther;
    //Users pay 0.5 ether. At specific milestones,
    function play() public payable { // credit their accounts.
        require(msg.value == 0.5 ether); // each play is 0.5 ether
        uint currentBalance = this.balance + msg.value;
        // ensure no players after the game has finished
        require(currentBalance <= finalMileStone);
        // if at a milestone, credit the player's account
        if(currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }

    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
```

```
}
```

Questo contratto rappresenta un gioco semplice (che naturalmente implicherebbe condizioni di gara) in cui i giocatori inviano 0,5 ether al contratto nella speranza di essere il giocatore che raggiunge per primo uno dei tre traguardi. Il primo a raggiungere il traguardo può reclamare una parte dell'ether al termine del gioco. Il gioco termina quando viene raggiunto il traguardo finale (10 ether); a questo punto gli utenti possono quindi richiedere i loro premi.

I problemi con il contratto EtherGame derivano dallo sbagliato utilizzo di this.balance in entrambe le linee 14 (e per associazione la riga 16) e 32. Un utente malintenzionato potrebbe inviare con la forza una piccola quantità di ether, ad esempio 0,1 ether, tramite la funzione selfdestruct ,discussa in precedenza, per impedire ad eventuali futuri giocatori il raggiungimento di un traguardo. In questo modo this.balance non sarà mai un multiplo di 0,5 ether grazie a questo contributo di 0,1 ether, poiché tutti i "giocatori legittimi" possono inviare solo incrementi di 0,5 ether. Ciò impedisce che tutte le condizioni "if" sulle righe 18, 21 e 24 possano verificarsi.

Un ulteriore vettore che ha mancato un traguardo potrebbe inviare con la forza 10 ether (o una quantità equivalente di ether che porterebbe il saldo del contratto sopra il finalMileStone), bloccando per sempre tutte le ricompense del contratto. Questo perché la funzione claimReward funzione verrà sempre ripristinata, a causa della richiesta sulla riga 32 (questo perché this.balance è maggiore di finalMileStone).

Tecniche preventive

Questo tipo di vulnerabilità deriva in genere dall'uso improprio di this.balance. La logica contrattuale, quando possibile, dovrebbe evitare di dipendere da valori esatti del saldo del contratto, perché può essere manipolata artificialmente. Se si applica la logica basata sull' this.balance, bisogna fare quindi i conti con equilibri di bilancio imprevisti.

Se sono richiesti valori esatti di ether depositato, dovrebbe essere utilizzata una variabile autodefinita che viene incrementata nelle funzioni pagabili, per tracciare in modo sicuro l'ether depositato. Questa variabile non sarà influenzata dall'ether forzato inviato tramite una chiamata selfdestruct.

Tenendo presente questo, una versione corretta del contratto EtherGame potrebbe assomigliare a:

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
```

```
    uint public finalReward = 5 ether;
    uint public depositedWei;

    mapping (address => uint) redeemable Ether;

    function play() public payable {
        require(msg.value == 0.5 ether);
        uint currentBalance = depositedWei + msg.value;
        // ensure no players after the game has finished
        require(currentBalance <= finalMileStone);
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        depositedWei += msg.value;
        return;
    }

    function claimReward() public {
        // ensure the game is complete
        require(depositedWei == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
}
```

Qui abbiamo creato una nuova variabile, depositedEther, che tiene traccia dell'ether depositato, sarà questa la variabile che utilizziamo per i nostri test. In questo modo non si fa nessun riferimento all'utilizzo di this.balance.[1]

5.4 Default Visibilities

Le funzioni in Solidity sono dotate di specificatori di visibilità che definiscono le modalità con cui possono essere invocate. Questa visibilità stabilisce se una funzione è accessibile agli utenti esterni, ai contratti derivati, solo internamente o solo esternamente. Di default, le funzioni sono impostate come pubbliche (public), il che consente agli utenti di invocarle dall'esterno. Esamineremo ora come un uso improprio degli specificatori di visibilità possa condurre a vulnerabilità significative all'interno dei smart contract.

La vulnerabilità

La visibilità predefinita per le funzioni è pubblica (public); quindi, le funzioni che non specificano la loro visibilità possono essere richiamate da utenti esterni. Il problema sorge quando gli sviluppatori accidentalmente omettono gli specificatori di visibilità su funzioni che dovrebbero essere private (cioè richiamabili solo all'interno del contratto stesso). Esaminiamo un esempio :

```
contract HashForEther {

    function withdrawWinnings() {
        // Winner if the last 8 hex characters of the address are 0
        require(uint32(msg.sender) == 0);
        _sendWinnings();
    }

    function _sendWinnings() {
        msg.sender.transfer(this.balance);
    }
}
```

Questo contratto è stato sviluppato come un gioco di scommesse in cui bisogna indovinare un indirizzo. Per ottenere il saldo del contratto come vincita, l'utente deve generare un indirizzo Ethereum con gli ultimi 8 caratteri esadecimali pari a 0. Una volta fatto ciò, può invocare la funzione withdrawWinnings per riscuotere la sua vincita. Purtroppo, non è stata specificata la visibilità delle funzioni. In particolare, la funzione _sendWinnings è stata impostata come pubblica (la visibilità predefinita), consentendo così a qualsiasi indirizzo di richiamare questa funzione e sottrarre la vincita.

Tecniche preventive

È sempre consigliabile specificare la visibilità di tutte le funzioni presenti in un contratto, anche nel caso in cui siano intenzionalmente pubbliche.

Esempi reali: Parity Multisig Wallet

Nel primo attacco di tipo Parity multisig, sono stati sottratti Ether per un valore di circa 31 milioni di dollari, prevalentemente da tre portafogli. In sostanza, il portafoglio multisig è costruito su un contratto base chiamato Wallet, che richiama un contratto di libreria contenente le principali funzionalità . Il contratto della libreria include il codice per inizializzare il portafoglio, come evidenziato dal frammento seguente:

```
contract WalletLibrary is WalletEvents {

  ...

  // METHODS
```

```
  ...

  // constructor is given number of sigs required to do protected
  // "onlymanyowners" transactionsas well as the selection of addresses
  // capable of confirming them
  function initMultiowned(address[] _owners, uint _required) {
    m_numOwners = _owners.length + 1;
    m_owners[1] = uint(msg.sender);
    m_ownerIndex[uint(msg.sender)] = 1;
    for (uint i = 0; i < _owners.length; ++i)
    {
      m_owners[2 + i] = uint( owners[i]);
      m_ownerIndex[uint(_owners[i])] = 2 + i;
    }
    m_required = _required;
  }

  ...

  // constructor - just pass on the owner array to multiowned and
  // the limit to daylimit
  function initWallet(address[] _owners, uint _required, uint _daylimit) {
    initDaylimit(daylimit);
    initMultiowned( owners, _required);
  }
}
```

Nessuna delle funzioni di cui sopra specifica esplicitamente la propria visibilità, pertanto entrambe sono impostate come pubbliche (public). La funzione initWallet viene invocata dal costruttore del portafoglio e stabilisce i proprietari del portafoglio multisig, come evidenziato anche nella funzione initMultiowned. Poiché queste funzioni sono state erroneamente lasciate pubbliche, un utente malintenzionato è in grado di richiamarle attraverso contratti distribuiti, assegnando la proprietà all'indirizzo dell'utente malintenzionato al proprio indirizzo. Assumendo il controllo come proprietario, l'attaccante ha poi prelevato gli ether presenti nei portafogli.[1][42]6.3

5.5 Entropy Illusion

Ogni transazione nella blockchain di Ethereum rappresenta un'operazione di stato deterministica. Ciò implica che ciascuna transazione apporta un cambiamento calcolabile allo stato globale dell'ecosistema Ethereum, senza alcuna incertezza. Questa caratteristica ha una conseguenza fondamentale: non esiste alcuna sorgente di entropia o casualità all'interno di Ethereum.

La vulnerabilità

Supponiamo ora di voler realizzare un gioco d'azzardo decentralizzato. Questo richiede l'elemento dell'incertezza (cioè qualcosa su cui scommettere), che rende la creazione di un sistema di gioco d'azzardo sulla blockchain, intrinsecamente deterministica, una sfida significativa. È evidente che l'incertezza deve derivare da una fonte esterna alla blockchain. Questo è possibile in caso di scommesse tra giocatori (come ad esempio l'approccio commit-reveal).

Facciamo un esempio; Un errore frequente è l'utilizzo di variabili relative a blocchi futuri, cioè variabili contenenti informazioni su blocchi di transazioni il cui contenuto non è ancora noto, come ad esempio hash, timestamp, numeri di blocchi o limiti di gas. Il problema, con queste variabili, è che queste sono controllate dal miner che estrae il blocco e pertanto non sono intrinsecamente casuali. Ad esempio, consideriamo uno smart contract per la roulette con una logica che restituisce un numero nero se l'hash del blocco successivo termina con un numero pari. Un miner o un gruppo di miner potrebbe scommettere 1 milione di dollari sul nero. Se risolvono il blocco successivo e trovano che l'hash termina con un numero dispari, potrebbero tranquillamente non rendere pubblico il loro blocco e generare un nuovo blocco finché non troveranno una soluzione con un hash che si conclude con un numero pari.

Tecniche preventive

L'origine dell'entropia deve risiedere esternamente alla blockchain. Questo può essere realizzato, tra pari, attraverso sistemi come il commit-reveal o modificando il modello di affidabilità in un gruppo di partecipanti (come nel caso di RandDAO). Inoltre, è possibile ottenere questo risultato tramite un'entità centralizzata che agisce come un oracolo di casualità. Le variabili di blocco non dovrebbero essere impiegate per generare entropia, poiché potrebbero essere manomesse dai miner.

Esempi reali: contratti PRNG

Nel febbraio 2018, Arseny Reutov ha condiviso nel suo blog un'analisi condotta su 3.649 smart contract in tempo reale che facevano uso di generatori di numeri pseudocasuali (PRNG). Durante questa analisi, è emerso che 43 contratti presentavano delle vulnerabilità sfruttabili.[43][1]

5.6 External Contract Referencing

Uno dei vantaggi che l'ecosistema di Ethereum offre è la capacità di riutilizzare il codice e di interagire con i contratti già esistenti sulla rete. Di conseguenza, molti contratti fanno riferimento ad altri contratti esterni, spesso attraverso chiamate di messaggi esterni. Queste chiamate possono rendere più complesso rilevare e mettere in luce i possibili approcci che un malintenzionato potrebbe adottare.

La vulnerabilità

In Solidity, è possibile assegnare a un contratto qualsiasi indirizzo, indipendentemente dal fatto che il codice di quell'indirizzo rappresenti effettivamente il tipo di contratto previsto. Questa situazione può causare difficoltà, specialmente quando l'autore del contratto cerca di nascondere un codice dannoso. Per comprendere meglio, esaminiamo un esempio concreto.

Immaginiamo di avere un frammento di codice come "Rot13Encryption.sol", che implementa in modo basilare la crittografia ROT13.

Esempio 8. Rot13Encryption.sol

```
// encryption contract
contract Rot13Encryption {

    event Result(string convertedString);

    // rot13-encrypt a string
    function rot13Encrypt (stringtext) public     {
        uint256 length = bytes(text).length;
        for(var i = 0; i < length; i++)     {
            byte char = bytes(text)[i];
            // inline assembly to modify the string
            assembly {
                // get the first byte
                char := byte(0,char)
                // if the character is in [n,z], i.e. wrapping
                if and(gt(char,0x6D), lt(char,0x7B))
                // subtract from the ASCII number 'a',
                // the difference between character <char> and 'z'
                { char:= sub(0x60, sub(0x7A, char))     }
                if iszero(eq(char, 0x20)) // ignore spaces
                // add 13 to char
                {mstore8(add(add(text,0x20), mul(i,1)), add(char,13))}
            }
        }
        emit Result(text);
    }

    // rot13-decrypt a string
    function rot13Decrypt (string text) public     {
        uint256 length = bytes(text).length;
        for(var i = 0; i < length; i++)     {
            byte char = bytes(text)[i];
            assembly {
                char := byte(0,char)
                if and(gt(char,0x60), lt(char,0x6E))
                { char:= add(0x7B, sub(char,0x61))    }
                if iszero(eq(char, 0x20))
```

```
                    {mstore8(add(add(text,0x20), mul(i,1)), sub(char,13))}
                }
            }
            emit Result(text);
        }
}
```

Questo codice esegue una crittografia semplice, prendendo una stringa composta da lettere dalla 'a' alla 'z' e spostando ogni carattere di 13 posizioni verso destra (riavvolgendo da 'z' a 'a'), seguendo un meccanismo simile a un cifrario di Cesare. Ora esaminiamo il contratto successivo, il quale fa uso di questo codice per la crittografia:

```
import "Rot13Encryption.sol";

// encrypt your top-secret info
contract EncryptionContract {
    // library for encryption
    Rot13Encryption encryptionLibrary;

    // constructor - initialize the library
    constructor(Rot13Encryption _encryptionLibrary) {
        encryptionLibrary = _encryptionLibrary;
    }

    function encryptPrivateData(string privateInfo) {
        // potentially do some operations here
        encryptionLibrary.rot13Encrypt(privateInfo);
    }
}
```

Il problema evidente in questo contratto è che l'indirizzo "encryptionLibrary" non è né pubblico né costante. Di conseguenza, il creatore del contratto potrebbe inserire un indirizzo nel costruttore che punti a un contratto diverso:

```
// encryption contract
contract Rot26Encryption {

    event Result(string convertedString);

    // rot13-encrypt a string
    function rot13Encrypt(stringtext) public {
        uint256 length = bytes(text).length;
        for(var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            // inline assembly to modify the string
            assembly {
                // get the first byte
                char := byte(0,char)
                // if the character is in [n,z], i.e. wrapping
                if and(gt(char,0x6D), lt(char,0x7B))
```

```
                // subtract from the ASCII number 'a',
                // the difference between character <char> and 'z'
                { char := sub(0x60, sub(0x7A, char))    }
                // ignore spaces
                if iszero(eq(char, 0x20))
                // add 26 to char!
                {mstore8(add(add(text,0x20), mul(i,1)), add(char,26))}
            }
        }
        emit Result(text);
    }

    // rot13-decrypt a string
    function rot13Decrypt (string text) public    {
        uint256 length = bytes(text).length;
        for(var i = 0; i < length; i++)    {
            byte char = bytes(text)[i];
            assembly {
                char := byte(0,char)
                if and(gt(char,0x60), lt(char,0x6E))
                { char := add(0x7B, sub(char,0x61))    }
                if iszero(eq(char, 0x20))
                {mstore8(add(add(text,0x20), mul(i,1)), sub(char,26))}
            }
        }
        emit Result(text);
    }
}
```

Questo contratto adotta una variazione del codice ROT26, che in pratica non altera la posizione di alcun carattere, poiché l'alfabeto è composto da 26 lettere e uno spostamento di 26 posizioni equivale a nessuna modifica. -. In modo più chiaro, un attaccante avrebbe potuto collegare il contratto seguente per ottenere lo stesso effetto:

```
contract Print {
    event Print(string text);

    function rot13Encrypt(stringtext) public    {
        emit Print(text);
    }
 }
```

Se fosse fornito l'indirizzo di uno di questi contratti nel costruttore, la funzione "encryptPrivateData" si limiterebbe a generare un evento che rende visibili i dati privati non cifrati.

Sebbene in questo esempio un contratto simile a una libreria sia stato impostato nel costruttore, è comune che un utente privilegiato possa modificare gli indirizzi dei contratti di libreria.

Nel caso in cui un contratto collegato non contenga la funzione richiesta, verrà invece eseguita la funzione di fallback. Per esempio, considerando la linea "encryptionLibrary.rot13Encrypt()", se il contratto specificato da "encryptionLibrary" fosse:

```
contract Blank {
    event Print(string text);
    function() {
        emit Print("Here");
        // put malicious code here and it will run
    }
}
```

La funzione quindi produrrebbe un evento con il testo "Here". Di conseguenza, se gli utenti possono alterare le librerie del contratto, in linea di principio potrebbero ingannare altri utenti portandoli a eseguire del codice arbitrario senza rendersene conto.

Tecniche preventive

Come dimostrato in precedenza, i contratti sicuri possono essere implementati in modo tale da agire malevolmente. Un revisore potrebbe analizzare pubblicamente un contratto e suggerire al proprietario di incorporare comportamenti dannosi, con il risultato di un contratto che è stato sottoposto a ispezione pubblica, ma che contiene vulnerabilità o intenti dannosi.

Esistono numerose tecniche per prevenire tali scenari. Una di queste consiste nell'utilizzo della parola chiave "new" per creare contratti. Nel contesto dell'esempio precedente, il costruttore potrebbe essere formulato come:

```
constructor() {
    encryptionLibrary = new Rot13Encryption();
}
```

In questo modo, al momento della distribuzione, viene creata un'istanza del contratto di riferimento, e il distributore non può sostituire il contratto "Rot13Encryption" senza modificarlo.

Un'altra soluzione è rappresentata dall'uso di codifica degli indirizzi dei contratti esterni. In generale, è fondamentale esaminare attentamente il codice che richiama contratti esterni. Per gli sviluppatori, nella definizione di contratti esterni, può essere una buona pratica rendere pubblici gli indirizzi dei contratti, in modo da consentire agli utenti di esaminare agevolmente il codice a cui fa riferimento il contratto.

Al contrario, se un contratto possiede un indirizzo contrattuale variabile e privato, ciò può essere un segnale di comportamento malevolo. Nel caso in cui un utente possa modificare un indirizzo di contratto utilizzato per richiamare funzioni esterne, potrebbe risultare importante (specialmente all'interno di un contesto di sistema decentralizzato) implementare un meccanismo

di blocco temporale e/o di voto. Questi meccanismi permetterebbero agli utenti di monitorare le modifiche apportate al codice o di decidere di aderire o meno al nuovo indirizzo contrattuale.

Esempi reali: Reentrancy Honey Pot

Sulla rete principale sono stati recentemente introdotti numerosi "honey pot". Questi contratti cercano di ingannare gli hacker di Ethereum che tentano di sfruttarli, ma invece finiscono per perdere ether a causa del contratto stesso che cercano di sfruttare. Un esempio di questa tattica consiste nella sostituzione di un contratto legittimo con uno dannoso durante la fase di costruzione. Il codice di riferimento.[48]

```
pragma solidity ^0.4.19;

contract Private Bank
{
    mapping (address => uint) public balances;
    uint public MinDeposit = 1 ether;
    Log TransferLog;

    function Private_Bank(address _log)
    {
        TransferLog = Log(log);
    }

    function Deposit()
    public
    payable
    {
        if(msg.value >= MinDeposit)
        {
            balances[msg.sender]+=msg.value;
            TransferLog.AddMessage(msg.sender,msg.value,"Deposit");
        }
    }

    function CashOut(uint _am)
    {
        if(_am<=balances[msg.sender])
        {
            if(msg.sender.call.value(_am)())
            {
                balances[msg.sender]-=_am;
                TransferLog.AddMessage(msg.sender,_am,"CashOut");
            }
        }
    }

    function() public payable{}
```

```
}

contract Log
{
    struct Message
    {
        address Sender;
        string  Data;
        uint  Val;
        uint  Time;
    }

    Message[] public History;
    Message LastMsg;

    function AddMessage(address _adr, uint _val, string _data)
    public
    {
        LastMsg.Sender = _adr;
        LastMsg.Time = now;
        LastMsg.Val = _val;
        LastMsg.Data = _data;
        History.push(LastMsg);
    }
}
```

[1]

5.7 Short Address/Parameter Attack

Questo tipo di attacco non è diretto ai contratti Solidity stessi, ma piuttosto alle applicazioni di terze parti che potrebbero interagire con essi.

La vulnerabilità

Quando si passano dei parametri a uno smart contract, questi vengono codificati secondo le specifiche dell' ABI. È possibile inviare parametri codificati più corti della lunghezza prevista del parametro. Ad esempio, è possibile inviare un indirizzo con soli 38 caratteri esadecimali (19 byte) invece dei 40 caratteri esadecimali standard (20 byte). In tal caso, l'EVM aggiungerà degli zeri alla fine dei parametri codificati per raggiungere la lunghezza corretta.

Questa situazione diventa problematica quando le applicazioni di terze parti invalidano gli input. Un esempio evidente si verifica quando un exchange non verifica l'indirizzo di un token

ERC20 durante una richiesta di prelievo da parte di un utente.

Consideriamo l'interfaccia della funzione di trasferimento standard per un token ERC20, prendendo nota dell'ordine dei parametri:

```
function transfer(address to, uint tokens)
                                public returns (bool success);
```

Esaminiamo ora un caso in cui un exchange detiene una considerevole quantità di token e un utente intende ritirare la sua quota di 100 token. In questo scenario, l'utente fornirebbe il proprio indirizzo 0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddead e il numero di token 100. L'exchange codificherebbe questi parametri seguendo l'ordine specificato dalla funzione "transfer", ovvero prima l'indirizzo e poi i token. Il risultato della codifica sarebbe il seguente:

```
a9059cbb000000000000000000000000mortomortodea \
dmortomortomortomortomortomorto0000000000000
0000000000000000000000000000000000056 bc75e2d63100000
```

I primi 4 byte (a9059cbb) rappresentano il selettore della funzione "transfer", seguiti dai successivi 32 byte che rappresentano l'indirizzo e altri 32 byte che corrispondono al valore uint256 del numero di token. Importante notare che l'ottavo esadecimale (56bc75e2d63100000) rappresenta 100 token, con 18 cifre decimali, come specificato dal contratto.

Esaminiamo ora cosa potrebbe accadere se venisse inviato un indirizzo con 1 byte mancante (2 cifre esadecimali). Supponiamo che un utente malevolo invii 0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddeadde come indirizzo (mancano le ultime due cifre) e cerchi di prelevare gli stessi 100 token. Nel caso in cui l'exchange non convalidi questo input, la codifica risultante sarebbe:

```
a9059cbb000000000000000000000000mortomortodea \
dmortomortomortomortomortomorto00000000000000
0000000000000000000000000000000000056 bc75e2d6310000000
```

La differenza è sottile ma rilevante. Si può notare che è stato aggiunto "00" alla fine della codifica per correggere l'indirizzo abbreviato inviato. Quando questa codifica è inviata allo smart contract, i parametri "address" verranno letti come "0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde00" e il valore sarà letto come "56bc75e2d6310000000" (si osservino i due zeri aggiuntivi). Di conseguenza, il valore ora rappresenta 25600 token (essendo il valore stato moltiplicato per 256).

In questo esempio, se l'exchange detenesse un numero cos`ı elevato di token, l'utente preleverebbe erroneamente 25600 token all'indirizzo modificato, mentre l'exchange crede che l'utente stia effettuando un prelievo di soli 100 token. Naturalmente, l'attaccante in questo caso non possederebbe l'indirizzo modificato.Tuttavia, se l'attaccante fosse in grado di generare un indirizzo che termina con un 0 (cosa che può essere facilmente ottenuta tramite un attacco di forza bruta)

e utilizzasse questo indirizzo generato, potrebbe sottrarre token dall'exchange all'insaputa dello stesso.

Tecniche preventive

È essenziale convalidare tutti i parametri di input nelle applicazioni esterne prima di trasmetterli alla blockchain. Inoltre, è importante sottolineare che l'ordine dei parametri gioca un ruolo fondamentale in questo contesto. Poiché il riempimento con zeri si verifica solo alla fine della codifica, un meticoloso ordinamento dei parametri nello smart contract può contribuire a mitigare alcune forme di questo tipo di attacco.[1]

5.8 Unchecked CALL Return Values

Esistono diversi metodi per eseguire chiamate esterne in Solidity. Il trasferimento di ether a conti esterni è comunemente effettuato utilizzando il metodo "transfer". Tuttavia, è anche possibile utilizzare la funzione "send"e , per chiamate esterne più flessibili, è possibile impiegare il codice operativo CALL direttamente in Solidity. Le funzioni "call" e "send" restituiscono un valore booleano che indica se la chiamata è stata eseguita con successo o meno. Pertanto, è importante notare che queste funzioni presentano un avviso: se la chiamata esterna (inizializzata tramite "call" o "send") non riesce, la transazione che contiene non verrà annullata o ripristinata, Ma, le funzioni restituiranno semplicemente "false". Un errore comune è che gli sviluppatori possano erroneamente presumere che una transazione venga annullata in caso di fallimento di una chiamata esterna, senza una verifica del il valore restituito.

La vulnerabilità

Consideriamo il seguente esempio:

```
contract Lotto {

    bool public payedOut = false;
    address public winner;
    uint public winAmount;

    // ... extra functionality here

    function sendToWinner() public {
        require(!payedOut);
        winner.send(winAmount);
        payedOut = true;
    }

    function withdrawLeftOver() public {
        require(payedOut);
```

```
        msg.sender.send(this.balance);
    }
}
```

Questo codice rappresenta un contratto simile a una lotteria, in cui il vincitore ("winner") riceve una somma di ether ("winAmount"), lasciando di solito un piccolo importo residuo che può essere ritirato da chiunque.

La vulnerabilità si trova nella riga 11, dove viene utilizzato il metodo "send" senza che venga effettuato alcun controllo sulla risposta. In questo semplice esempio, una transazione da parte del vincitore ("winner") che fallisce, a causa dell'esaurimento di gas o perché il contratto destinatario esegue intenzionalmente una funzione di fallback, potrebbe portare alla modifica di "payedOut", impostandola su "true", indipendentemente dall'invio effettivo di ether. In questo scenario, chiunque potrebbe quindi ritirare l'importo residuo tramite la funzione "withdrawLeftOver".

Tecniche preventive

Quando possibile, è preferibile utilizzare la funzione "transfer" invece di "send", poiché "transfer" verrà annullata se la transazione esterna viene annullata. Nel caso in cui sia necessario utilizzare "send", è fondamentale verificare sempre il valore restituito per garantire la gestione corretta delle transazioni.

Tuttavia, una raccomandazione ancora più forte è quella di adottare un modello di prelievo. In questo approccio, ogni utente dovrebbe richiamare una funzione di prelievo separata che gestisce l'invio di ether al di fuori del contratto e affronta le conseguenze delle transazioni di invio non riuscite. L'idea alla base è di isolare logicamente la funzionalità di invio esterno dal resto del codice e di trasferire la responsabilità di una possibile transazione fallita all'utente finale che invoca la funzione di prelievo.

```
constructor() {
    encryptionLibrary = new Rot13Encryption();
}
```

In questo modo, al momento della distribuzione, viene creata un'istanza del contratto di riferimento, e il distributore non può sostituire il contratto "Rot13Encryption" senza modificarlo.

Un'altra soluzione è rappresentata dall'uso di codifica degli indirizzi dei contratti esterni. In generale, è fondamentale esaminare attentamente il codice che richiama contratti esterni. Per gli sviluppatori, nella definizione di contratti esterni, può essere una buona pratica rendere pubblici gli indirizzi dei contratti, in modo da consentire agli utenti di esaminare agevolmente il codice a cui fa riferimento il contratto.

Al contrario, se un contratto possiede un indirizzo contrattuale variabile e privato, ciò può

essere un segnale di comportamento malevolo. Nel caso in cui un utente possa modificare un indirizzo di contratto utilizzato per richiamare funzioni esterne, potrebbe risultare importante (specialmente all'interno di un contesto di sistema decentralizzato) implementare un meccanismo di blocco temporale e/o di voto. Questi meccanismi permetterebbero agli utenti di monitorare le modifiche apportate al codice o di decidere di aderire o meno al nuovo indirizzo contrattuale.

Esempi reali: Etherpot and King of the Ether

Etherpot rappresentava una lotteria basata su smart contract, non dissimile dal contratto di esempio menzionato in precedenza. Anche questo contratto aveva il problema di un valore di chiamata non controllato. Esaminiamo la funzione "cashin" in "lotto.sol": (snippet di codice).

Esempio 9. lotto.sol: snippet di codice

```
...
  function cash(uint roundIndex, uint subpotIndex){

        var subpotsCount = getSubpotsCount(roundIndex);

        if(subpotIndex>=subpotsCount)
            return;

        var decisionBlockNumber =
                    getDecisionBlockNumber(roundIndex, subpotIndex);

        if(decisionBlockNumber>block.number)
            return;

        if(rounds[roundIndex].isCashed[subpotIndex])
            return;
        //Subpots can only be cashed once.
        //This is to prevent double payouts
        var winner = calculateWinner(roundIndex, subpotIndex);
        var subpot = getSubpot(roundIndex);

        winner.send(subpot);

        rounds[roundIndex].isCashed[subpotIndex] = true;
        //Mark the round as cashed
}
...
```

Si noti che alla riga 21, il valore restituito dalla funzione "send" non viene controllato e, nella riga successiva, viene impostato un valore booleano che indica che i fondi sono stati inviati al vincitore. Questo errore potrebbe creare una situazione in cui il vincitore non riceve gli ether, ma lo stato del contratto indica erroneamente che il vincitore è stato già pagato.

Una situazione ancora più grave per questo errore è emersa nel caso del "King of the ether". In un post-mortem dettagliato di questo contratto, è stato descritto in modo eccellente come un errore non controllato potrebbe essere sfruttato per attaccare il contratto.[1]

5.9 Race Conditions/Front Running

La combinazione di chiamate esterne ad altri contratti e la natura multiutente della blockchain di Ethereum genera una serie di possibili insidie nei contratti in Solidity, dove gli utenti si contendono l'esecuzione del codice per raggiungere stati imprevisti. Un esempio di tale situazione è il reentrancy (già discusso in precedenza in questo capitolo). In questa sezione, esamineremo ulteriori tipologie di condizioni di competizione che possono sorgere sulla blockchain di Ethereum.

La vulnerabilità

Come nella maggior parte delle blockchain, le transazioni di Ethereum vengono aggregati in blocchi. Queste sono considerate valide solo quando un miner risolve il meccanismo di consenso. Il miner che risolve il blocco seleziona le transazioni dal pool da includere, spesso in base al prezzo del gas di ciascuna transazione. Qui emerge un possibile vettore di un attacco. Un utente malintenzionato può monitorare il pool di transazioni per individuare le transazioni che potrebbero contenere soluzioni a problemi, e quindi modificare o annullare le autorizzazioni del risolutore Oppure alterare lo stato di un contratto in modo dannoso per il risolutore. L'attaccante può quindi estrapolare i dati da questa transazione e creare una propria transazione con un valore di gasPrice più elevato, al fine di far sì che la sua transazione sia inclusa in un blocco prima di quella originale.

Esaminiamo come questo potrebbe funzionare con un semplice esempio. Prendi in considerazione il contratto illustrato in FindThisHash.sol.

```
contract FindThisHash {
  bytes32 constant public hash =
  0xb5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

  constructor() public payable {} // load with ether

  function solve(string solution) public {
    // If you can find the pre-image of the hash,
    require(hash == sha3(solution));// receive 1000 ether
    msg.sender.transfer(1000 ether);
  }
}
```

Supponiamo che questo contratto contenga 1.000 ether. Un utente in grado di individuare la variabile in chiaro corrispondente all'hash SHA-3 seguente:

0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a

può inviare la soluzione e recuperare i 1.000 ether. Immagina che un utente abbia compreso che la soluzione è "Ethereum!". Pertanto, invia la soluzione utilizzando "Ethereum!" come parametro nella funzione"solve". A questo punto, un ipotetico utente malintenzionato, che sta monitorando il pool delle transazioni per individuare qualsiasi soluzione proposta, non appena individua questa soluzione, ne verifica la validità e avvia una transazione equivalente con un valore di gasPrice molto più elevato rispetto alla transazione originale. Questo accade perché il miner che risolverà il blocco è probabilmente attratto dall'alto valore del gasPrice e darà la priorità all'attaccante, quindi estrarrà la loro transazione prima di quella dell'utente che ha trovato la soluzione originale. Di conseguenza, l'aggressore riuscirà ad ottenere i 1.000 ether, mentre l'utente che ha risolto il problema non riceverà nulla. È importante notare che in questo tipo di vulnerabilità, conosciuta come "front-running", i miner sono fortemente incentivati a condurre gli attacchi da soli (oppure possono essere corrotti per eseguire questi attacchi a commissioni elevate).

Tecniche preventive

Esistono due categorie di attori che possono eseguire attacchi di front-running: gli utenti, che modificano il gasPrice delle proprie transazioni, e i miner stessi, che possono riordinare le transazioni all'interno di un blocco a loro vantaggio. Un contratto vulnerabile alla prima categoria (utenti) è più problematico di uno vulnerabile alla seconda (miner), poiché i miner possono eseguire l'attacco solo quando risolvono un blocco, il che è improbabile per ogni singolo miner che mira a un blocco specifico. Di seguito, elencheremo alcune misure di mitigazione relative ad entrambe le categorie di aggressori.

Un approccio consiste nell'applicare un limite superiore al gasPrice. Ciò impedisce agli utenti di aumentare il gasPrice per ottenere priorità di elaborazione oltre il limite stabilito. Questa misura protegge solo la prima categoria di aggressori (utenti arbitrari). I miner in questo scenario possono comunque attaccare il contratto, poiché possono ordinare le transazioni nel loro blocco a loro discrezione, indipendentemente dal gasPrice.

Un metodo più efficace impiega uno schema di tipo commit-reveal. In questo schema, gli utenti inviano transazioni contenenti informazioni nascoste (tipicamente un hash). Successivamente, dopo che la transazione è stata inclusa in un blocco, l'utente invia una transazione di rivelazione per rivelare i dati precedentemente nascosti (fase di rivelazione). Questo metodo impedisce sia ai miner che agli utenti di eseguire front-running, poiché non possono conoscere il contenuto della transazione in anticipo. Tuttavia, questo approccio non può celare il valore della transazione, che spesso risulta essere l'informazione sensibile da nascondere.

Un esempio è il smart contract ENS (Ethereum Name Service), che consentiva agli utenti di inviare transazioni con dati impegnati, inclusa la quantità di ether che erano disposti a spendere. Durante la fase di rivelazione, agli utenti veniva rimborsata la differenza tra l'importo inviato nella transazione e l'importo che avevano dichiarato inizialmente.

Esempi reali: ERC20 e Bancor

Lo standard ERC20 è ampiamente riconosciuto per la creazione di token su Ethereum. Tuttavia, questo standard presenta una rilevante vulnerabilità legata alla funzione "approve".
Nel dettaglio, lo standard definisce la funzione "approve" come segue:

```
function approve(address _spender, uint256 _value)
                                    returns (bool success)
```

Questa funzione consente agli utenti di autorizzare terzi a trasferire token per loro conto. La vulnerabilità di front-running si verifica in un contesto in cui un utente, diciamo Alice, concede l'autorizzazione al suo amico Bob di spendere 100 token. Successivamente, Alice decide di revocare questa autorizzazione e crea una transazione per ridurre l'assegnazione di Bob a 50 token. Bob, che monitora attentamente la blockchain, rileva questa transazione e agisce tempestivamente, eseguendo una sua transazione per spendere i 100 token. Bob assegna un gasPrice più alto alla sua transazione rispetto a quello di Alice, assicurandosi che la sua transazione venga confermata prima. In alcune implementazioni di "approve", Bob potrebbe trasferire i suoi 100 token e poi, quando la transazione di Alice viene confermata, modificare nuovamente l'autorizzazione a 50 token, ottenendo cos`ı l'accesso a 150 token in totale.

Un caso concreto di questa vulnerabilità è emerso nell'implementazione iniziale di Bancor.[4][1]

5.10 Denial of Service (DoS)

Questa categoria è piuttosto ampia, ma comprende principalmente attacchi attraverso i quali gli utenti possono rendere un contratto inutilizzabile per un certo periodo di tempo o, in alcuni casi, in modo permanente. Questo può comportare il blocco del ether all'interno di alcuni contratti per sempre, come accadde nell'attacco reale del Parity Multisig Wallet.

La vulnerabilità

Esistono vari modi in cui un contratto può diventare non raggiungibile. Qui di seguito alcuni modelli di codifica Solidity meno conosciuti che possono portare a vulnerabilità di tipo DoS:

Ciclo attraverso mappature o array manipolati esternamente

Questo modello si presenta spesso quando un proprietario intende distribuire token agli investitori tramite una funzione simile a quella illustrata in questo esempio di contratto:

```
contract DistributeTokens {
    address public owner; // gets set somewhere
    address [] investors; // array of investors
```

```
    uint[] investorTokens; // the amount of tokens each investor gets

    // ... extra functionality, including transfertoken()

    function invest() public payable    {
        investors.push(msg.sender);
        investorTokens.push(msg.value * 5); // 5 times the wei sent
        }

    function distribute() public {
        require(msg.sender == owner); // only owner
        for(uint i = 0; i < investors.length; i++)    {
            // here transferToken(to,amount) transfers "amount" of
            // tokens to the address "to"
            transferToken(investors[i],investorTokens[i]);
        }
    }
}
```

Si osservi che il ciclo in questo contratto opera su un array che potrebbe essere manipolato in modo da ridimensionarlo. Un utente malintenzionato potrebbe creare numerosi account utente, facendo aumentare le dimensioni dell'array "investor". In teoria, ciò potrebbe essere fatto in modo tale che il gas richiesto per eseguire il ciclo "for" superi il limite di gas del blocco, rendendo praticamente inutilizzabile la funzione di distribuzione.

Operazioni del'owner

Un altro modello comune riguarda i casi in cui i proprietari detengono privilegi specifici nei contratti e devono compiere predeterminate azioni affinché il contratto possa avanzare. Un esempio potrebbe essere rappresentato da un contratto per un'offerta iniziale di monete , in cui è richiesto che il proprietario esegua la funzione "finalize" per consentire ai token di essere trasferibili. Ad esempio:

```
bool public isFinalized = false;
address public owner; // gets set somewhere

function finalize() public {
    require(msg.sender == owner);
    isFinalized == true;
}

// ... extra ICO functionality

// overloaded transfer function
function transfer(address _to, uint _value) returns (bool) {
    require(isFinalized);
    super.transfer(_to,_value)
}
```

. . .

In queste situazioni, se l'utente con privilegi speciali perde le sue chiavi private o diventa inattivo, l'intero contratto dei token può diventare inutilizzabile. Ad esempio, se il proprietario non è in grado di eseguire la chiamata a "finalize", i token non potranno essere trasferiti e l'intero sistema dei token potrebbe dipendere da un singolo indirizzo.

Stato progressivo in base alle chiamate esterne

A volte, i contratti vengono sviluppati in modo tale che il passaggio a uno stato successivo dipenda dall'invio di ether ad un indirizzo o dall'attesa di un input proveniente da una fonte esterna. Tali schemi possono generare vulnerabilità agli attacchi di tipo Denial-of-Service (DoS) quando la chiamata esterna non riesce o viene ostacolata da fattori esterni. Ad esempio, nel contesto dell'invio di ether, un utente potrebbe creare un contratto che rifiuta di accettare ether. Se, per avanzare a uno stato successivo, il contratto richiede che l'ether venga ritirato , in questo modo il contratto non sarà mai in grado di raggiungere il nuovo stato. Ciò avviene perché l'ether non può essere ritirato o inviato al contratto dell'utente che ha impostato il divieto di accettazione dell'ether.

Tecniche preventive

Nel primo esempio, è importante evitare che i contratti dipendano da strutture di dati alterabili da utenti esterni. Un modello consigliato è l'approccio di prelievo, in cui ciascun investitore richiama autonomamente una funzione di ritiro per ottenere i propri token.

Nel secondo esempio, dove è richiesta l'azione di un utente privilegiato per modificare lo stato, possono essere implementati meccanismi di sicurezza per fronteggiare eventuali incapacità del proprietario.

Una soluzione potrebbe essere la trasformazione del proprietario in un contratto multisig. In alternativa, l'utilizzo di un blocco temporale potrebbe essere adottato: nell'esempio proposto, il requisito alla riga 13 potrebbe includere una clausola temporale come require(msg.sender == owner —— now ¿ unlockTime), che permetterebbe a qualsiasi utente di finalizzare l'azione dopo un periodo specifico indicato da unlockTime. Questo approccio potrebbe essere applicato anche al terzo esempio.

Nel caso in cui le chiamate esterne debbano portare a uno stato successivo, è consigliabile considerare la possibilità di fallimento e, se necessario, integrare modifiche future dello stato nel caso in cui la chiamata prevista non venga mai effettuata.

Esempi reali: GovernMental

GovernMental rappresentava un vecchio schema Ponzi, uno schema Ponzi è un modello economico di vendita, ideato da Charles Ponzi, che promette forti guadagni ai primi investitori, a discapito di nuovi "investitori", che ha accumulato una notevole quantità di ether,raggiungendo 1.100 ether. Purtroppo, questo schema è risultato vulnerabile alle stesse debolezze di tipo DoS discusse in questa sezione. Il contratto richiedeva la cancellazione di una mappatura di notevole dimensione per poter far in modo di prelevare l'ether accumulato. Il costo in termini di gas per tale operazione superava il limite di gas per il blocco in questione, rendendo impossibile il ritiro dei 1.100 ether. L'indirizzo del contratto in questione è

0 xF45717552f12Ef7cb65e95476F217Ea008167Ae3

Attraverso la transazione

0 x0 d 80 d 67202 bd 9 cb 6773 df 8 dd 2020 e 7190 a 1 b 0793 e 8 ec 4 fc 105257 e 8128 f 0506 b

, è evidente che i 1.100 ether sono stati prelevati tramite una transazione che ha utilizzato 2,5 milioni di gas, un valore possibile grazie all'aumento del limite di gas del blocco.[14][1]

5.11 Block Timestamp Manipulation

I timestamp dei blocchi hanno trovato applicazioni in diversi contesti, come la generazione di numeri casuali5.5,per il blocco dei fondi ad intervalli temporali e per l'implementazione di condizioni che variano lo stato in base al tempo. Va notato che i miner hanno la capacità di apportare piccole modifiche ai timestamp dei blocchi, e questa possibilità può rivelarsi rischiosa quando i timestamp vengono impiegati in modo errato all'interno degli smart contract.

La vulnerabilità

"block.timestamp" o più comunemente denominato "now", è soggetto a manipolazioni da parte dei miner qualora ne abbiano l'incentivo. A titolo di esempio, consideriamo un semplice gioco illustrato nel codice di roulette.sol, il che potrebbe risultare vulnerabile all'exploit da parte dei miner.

Esempio 11. roulette.sol

```
contract Roulette {
    uint public pastBlockTime; // forces one bet per block

    constructor() public payable {} // initially fund contract

    // fallback function used to make a bet
    function () public payable {
        require(msg.value == 10 ether); // must send 10 ether to play
        require(now != pastBlockTime); // only 1 transaction per block
```

```
        pastBlockTime = now;
        if(now % 15 == 0) { // winner
            msg.sender.transfer(this.balance);
        }
    }
}
```

Questo contratto opera come una semplice lotteria in cui ogni transazione per ciascun blocco può scommettere 10 ether per avere la possibilità di vincere l'intero saldo del contratto. L'idea alla base è che le ultime due cifre di block.timestamp siano uniformemente distribuite, creando così una possibilità su 15 di vincere questa lotteria.

Tuttavia, come è noto, i miner hanno la capacità di modificare il timestamp a loro vantaggio. In questa situazione specifica, se il contratto contiene una quantità significativa di ether, un miner che risolve un blocco potrebbe avere l'incentivo di selezionare un timestamp in modo che block.timestamp in modulo 15 sia uguale a 0. Questo gli permetterebbe di ottenere non solo le ricompense del blocco, ma anche l'Ether bloccato nel contratto. Tuttavia, va notato che questo modello è vulnerabile agli attacchi front-running poiché solo una persona è autorizzata a scommettere per blocco. 5.9

Nella realtà però, questo tipo di attacco incontra delle problematiche, infatti i timestamp dei blocchi crescono in modo monotono, quindi i miner non possono selezionare timestamp arbitrari per i blocchi. Inoltre, i miner sono limitati nell'impostare tempi di blocco troppo distanti nel futuro poiché i blocchi con timestamp futuri sono generalmente rifiutati dalla rete.3.4

Esempi reali: GovernMental

GovernMental, presentava una vulnerabilità legata al timestamp. Nel contratto, il pagamento veniva effettuato al giocatore che fosse stato l'ultimo a partecipare a un turno di gioco. Di conseguenza, un miner che agiva anche da giocatore poteva manipolare il timestamp, inserendo una data futura, per far apparire di essere stato l'ultimo giocatore ad unirsi per oltre un minuto, anche se ciò non fosse.[2][1]

5.12 Constructors with Care

I constructor sono funzioni speciali che spesso svolgono attività critiche e privilegiate durante l'inizializzazione dei contratti. Prima della versione 0.4.22 di Solidity, i costruttori venivano definiti con lo stesso nome del contratto in cui erano contenuti. In queste circostanze, se il nome del contratto veniva modificato durante lo sviluppo e il nome del costruttore non veniva aggiornato di conseguenza, questo poteva trasformarsi in una normale funzione richiamabile. È facile immaginare come ciò potesse portare (ed effettivamente ha portato) a interessanti situazioni di hacking contrattuale.

La vulnerabilità

Se il nome del contratto viene modificato o è presente un errore di battitura nel nome del costruttore tale da non corrispondere al nome del contratto, il costruttore si comporterà come una normale funzione. Ciò può portare a conseguenze disastrose, soprattutto se il costruttore esegue operazioni privilegiate. Consideriamo il seguente contratto:

```
contract OwnerWallet {
    address public owner;

    // constructor
    function ownerWallet(address _owner) public {
        owner = _owner;
    }

    // Fallback. Collect ether.
    function () payable {}

    function withdraw() public {
        require(msg.sender == owner);
        msg.sender.transfer(this.balance);
    }
}
```

Questo contratto accumula ether e consente solo al proprietario di prelevarlo, richiamando la funzione withdraw. Il problema sorge poiché il costruttore non ha lo stesso nome del contratto: la prima lettera è diversa. Di conseguenza, qualsiasi utente può chiamare la funzione ownerWallet, impostarsi come proprietario e quindi accedere a tutto l'ether nel contratto, richiamando la funzione withdraw.

Tecniche preventive

Questo problema è stato risolto nella versione 0.4.22 del compilatore Solidity. In questa versione è stata introdotta la parola chiave "constructor" per specificare il costruttore, eliminando cos`ı l'obbligo che il nome della funzione coincidesse con il nome del contratto. Si raccomanda di utilizzare questa nuova parola chiave per dichiarare i costruttori al fine di evitare possibili problemi legati alla denominazione.

Esempi reali: Rubixi

Rubixi era un altro schema piramidale/Ponzi che presentava la stessa vulnerabilità. Inizialmente noto come DynamicPyramid, il nome del contratto è stato cambiato prima della distribuzione a Rubixi. Tuttavia, il nome del costruttore non è stato modificato, permettendo a qualsiasi utente di diventare il creatore.In definitiva, questa vulnerabilità ha consentito agli utenti di gareggiare per il ruolo di creatore al fine di ottenere commissioni dallo schema piramidale.[2][1]

5.13 Tx.Origin Authentication

Solidity include una variabile globale, tx.origin, che attraversa l'intero stack di chiamate e contiene l'indirizzo dell'account che inizialmente ha avviato la chiamata o la transazione. Utilizzare questa variabile per l'autenticazione all'interno di uno smart contract espone il contratto a vulnerabilità di tipo phishing.

La vulnerabilità

I contratti che fanno uso della variabile tx.origin sono generalmente a rischio di attacchi di phishing, in cui gli utenti vengono portati ad eseguire azioni autenticate all'interno di un contratto vulnerabile.

Esaminiamo un esempio concreto, il contratto Phishable.sol.

Esempio 13. Phishable.sol

```
    address public owner;

    constructor (address _owner) {
        owner = _owner;
    }

    function () public payable {} // collect ether

    function withdrawAll(address _recipient) public {
        require(tx.origin == owner);
        _recipient.transfer(this.balance);
    }
}
```

Si noti che alla riga 11 il contratto autorizza la funzione withdrawAll utilizzando tx.origin. Questa situazione rende possibile per un utente malintenzionato creare un contratto di attacco come segue:

```
import "Phishable.sol";

contract AttackContract {

    Phishable phishableContract;
    address attacker; // The attacker's address to receive funds

    constructor (Phishable _phishableContract, address _attackerAddress) {
        phishableContract = _phishableContract;
        attacker = _attackerAddress;
```

```
    }

    function () payable {
        phishableContract.withdrawAll(attacker);
    }
}
```

Se la vittima invia una transazione con gas sufficiente all'indirizzo dell' AttackContract, verrà innescata la funzione fallback, che a sua volta richiamerà la funzione withdrawAll del contratto Phishable con il parametro attacker. Questo comporterà il prelievo di tutti i fondi dal contratto Phishable e il trasferimento all'indirizzo dell'attacker. Questo avviene perché l'indirizzo che ha inizializzato per primo la chiamata è quello della vittima (cioè il titolare del contratto Phishable). Pertanto, tx.origin sarà uguale all'owner e supererà la condizione di verifica alla linea 11 del contratto Phishable.

Tecniche preventive

L'utilizzo di tx.origin per l'autenticazione nei contratti intelligenti è sconsigliato. Tuttavia, ciò non implica che la variabile tx.origin non debba mai essere utilizzata. Esistono casi d'uso legittimi in cui questa variabile può essere utilizzata nei contratti intelligenti. Per esempio, se si desidera evitare che i contratti esterni richiamino il contratto corrente, è possibile implementare una verifica con require nella forma require(tx.origin == msg.sender). Questo approccio impedisce l'uso di contratti intermedi per richiamare il contratto corrente.[1]

Capitolo 6

Analisi automatizzata per gli Smart Contractt

In questo capitolo, esamineremo l'importanza degli strumenti di analisi automatica nel contesto degli smart contract. Vedremo come questi strumenti possono essere un alleato importante per identificare e mitigare potenziali problemi nel codice sorgente. Esploreremo diversi metodi di analisi presentando casi pratici in cui errori nei contratti hanno avuto conseguenze significative. Inoltre, in questo capitolo, presenteremo lo strumento di analisi automatica da me progettato per la verifica di smart contract in Solidity. Esamineremo il funzionamento di questo strumento e valuteremo la sua efficacia nel rilevare potenziali vulnerabilità.

6.1 Unit Test Overflow/Underflow

Un Unit Test è una pratica essenziale nel campo dello sviluppo del software, che serve a verificare il corretto funzionamento di singole unità di codice, come funzioni, metodi o classi, in modo isolato dal resto dell'applicazione. Il principale obiettivo di un test di unità è individuare errori o bug nel codice nelle prime fasi di uno sviluppo, consentendo ai programmatori di correggerli tempestivamente. In questa sezione, esamineremo un contratto estratto da un'applicazione reale che ha causato notevoli perdite in passato. In particolare, analizzeremo un attacco del 2018 in cui un gruppo di token ERC20 è risultato vulnerabile a un overflow del buffer.
Come abbiamo già esaminato nel capitolo precedente (vedi sezione X), ogni tipo di variabile ha una capacità massima per la lunghezza dei valori che può contenere. Violare i limiti di valore di una variabile può causare comportamenti imprevisti nell'applicazione. Ad esempio, un overflow potrebbe generare un errore di accesso a elementi oltre i limiti consentiti o causare un wrap-around del numero, portandolo al massimo o al minimo valore consentito.

Nel linguaggio Solidity, quando si supera il valore massimo consentito per un tipo "uint", il risultato ritorna a un numero che rientra nei suoi limiti. Ad esempio, se abbiamo una variabile che può contenere solo due cifre e raggiungiamo il numero 99, incrementando ancora otterremo il valore 00. Al contrario, se partiamo da 00 e sottraiamo 1, otterremo 99.
Gli sviluppatori si aspettano che ci si aspetta che le operazioni sui numeri si comportino in base alle regole matematiche tradizionali, quando invece seguono regole specifiche definite dal linguaggio.

Compreso cos'è un overflow e un underflow, esamineremo uno smart contract che ha portato, a dover sospendere tutti i trasferimenti dei token ERC20. Inizieremo analizzando il codice originale utilizzato nell'attacco e poi procederemo a riscrivere il contratto per proteggerlo da tali attacchi.
I token coinvolti in questo attacco facevano uso di una funzione di invio non sicura che non era protetta dagli overflow dei numeri interi. Questa vulnerabilità è stata successivamente copiata in molti token diversi e, obbligando i broker ad interrompere tutti i trasferimenti fino alla risoluzione del problema.

Il nostro processo di esame seguirà questi passaggi:

- Inizieremo estraendo il codice sorgente e analizzeremo la funzione vulnerabile.
- Successivamente, esamineremo il payload effettivo utilizzato nell'attacco originale su Etherscan per comprenderne il funzionamento.
- Infine, proporremo una soluzione per risolvere il problema e testeremo la nostra implementazione.

Di seguito è riportata la funzione del contratto ERC20 vulnerabile, insieme a un collegamento per visualizzare il codice su Etherscan. Etherscan è un servizio online per l'esplorazione e l'analisi della blockchain di Ethereum. Gli utenti possono utilizzare Etherscan per esplorare transazioni, indirizzi e smart contract, non appena questi vengono distribuiti e pubblicati.[15]

```
function batchTransfer(address[] _receivers, uint256 _value)
                                        public whenNotPaused returns (bool) {
        uint cnt = _receivers.length;
        uint256 amount = uint256(cnt) * _value;
        require(cnt > 0 && cnt <= 20);
        require(_value > 0 && balances[msg.sender] >= amount);


        balances[msg.sender] = balances[msg.sender].sub(amount);
        for (uint i = 0; i < cnt; i++)        {
        balances[_receivers[i]] = balances[_receivers[i]].add(value);
              Transfer(msg.sender, _receivers[i], _value);
        }
    return true;
}
```

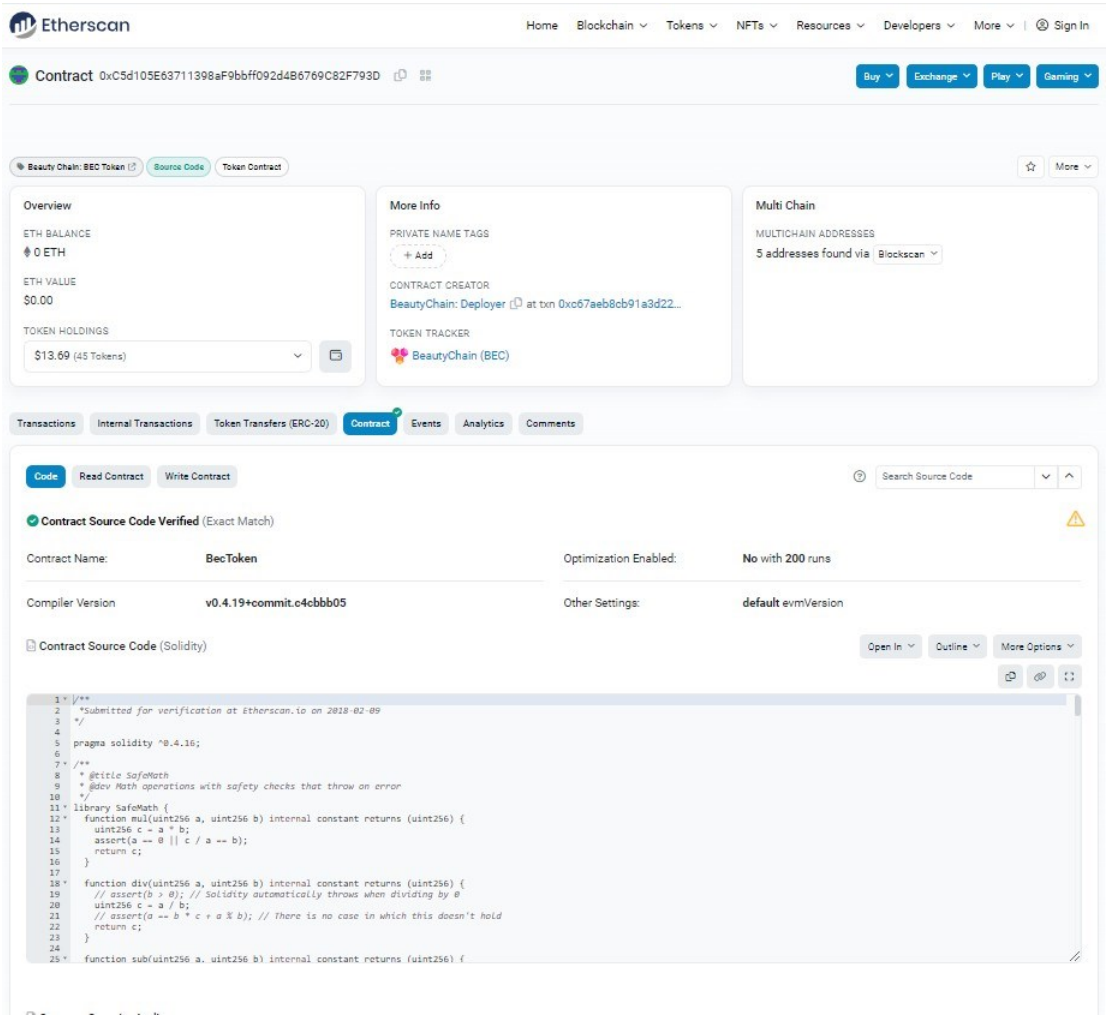

Figura 32: Screenshot di Etherscan del contratto ERC20.

Questa funzione effettua un controllo del saldo alla riga 5 basato sull'importo calcolato nella riga 3, che presenta una vulnerabilità di overflow.
L'importo viene calcolato moltiplicando la lunghezza dell'array per il valore inviato. Poiché non ci sono controlli per evitare che questa operazione matematica produca un valore inferiore al nostro saldo, è possibile forzare "amount" su 0 utilizzando un valore molto grande come "_value". Quando i saldi vengono aggiornati sulla riga 10, non viene utilizzato "amount" su 0, ma il valore iniziale "_value" inviato alla funzione. In questo caso, non c'è alcuna moltiplicazione, quindi non si verifica un overflow, il saldo viene semplicemente aggiornato.

Revisione della transazione di attacco reale

Ora diamo un'occhiata alla transazione reale che ha causato questo attacco di overflow.

Di seguito è riportata la transazione derivante dall'attacco oltre ad un collegamento per

visualizzare personalmente la transazione su Etherscan.[16]

Function : batchTransfer(address[] _receivers, uint256 _value)

MethodID : 0x83f12fec

[0]: 0000000000000000000000000000000000000000000000000000000000000040

[1]: 8000000000000000000000000000000000000000000000000000000000000000

[2]: 0000000000000000000000000000000000000000000000000000000000000002

[3]: 000000000000000000000000 b4d30cac5124b46c2df0cf3e3e1be05f42119033

[4]: 000000000000000000000000 e823ffe018727585eaf5bc769fa80472f76c3d7

Esaminiamo più da vicino i valori delle transazioni e come sono stati generati. Questo ci aiuterà a comprendere meglio l'attacco.
I dati all'interno di una transazione possono essere suddivisi in questo modo:

- Un MethodID di 4 byte.
- Cinque variabili da 32-byte.

Il MethodID di 4 byte che precede i parametri della funzione, rappresenta i primi 4 byte di un hash SHA-3, della dichiarazione del metodo "batchTransfer", escludendo i nomi delle variabili e gli spazi. Possiamo verificare questo valore utilizzando le funzioni di utilità dalla libreria Web3 come mostrato nell'immagine.

```
$ npm install web3
$ node
> const web3 = require('web3')
> web3.utils.sha3("batchTransfer(address[],uint256)").substring(0,10)
'0x83f12fec'
```

Figura 33: Screenshot del procediemento SHA-3 descritto

Le variabili che seguono il MethodID sono qui di seguito definite:

0 Offset all'array recievers, lunghezza: 40Hex o 64 byte

1 Questo è il _value effettivo inviato che, se moltiplicato, provoca un overflow.

2 Questa è la dimensione dell'array _recievers inviato al trasferimento, in questo caso 2 indirizzi

3 Questo è il primo indirizzo dell'array _recievers.

4 Questo è il secondo indirizzo dall'array _recievers.

L'azione compiuta dall' attacco è quella di prendere un valore molto grande dalla variabile [1] e moltiplicarlo per la lunghezza dell'array, che è pari a 2. Ciò ha creato una condizione di overflow che ha portato il valore a diventare 0 permettendo all'attaccante di superare i controlli e completarlo.

Tenderly

Uno strumento che ho utilizzato ampiamente nel mio percorso è Tenderly, una piattaforma online open source che consente di simulare con precisione un attacco, replicando la situazione esatta nello stato del blocco della transazione in esame.
Facciamo l'esempio utilizzando i dati analizzati nella sezione sopra, presi direttamente da Etherscan:[16]

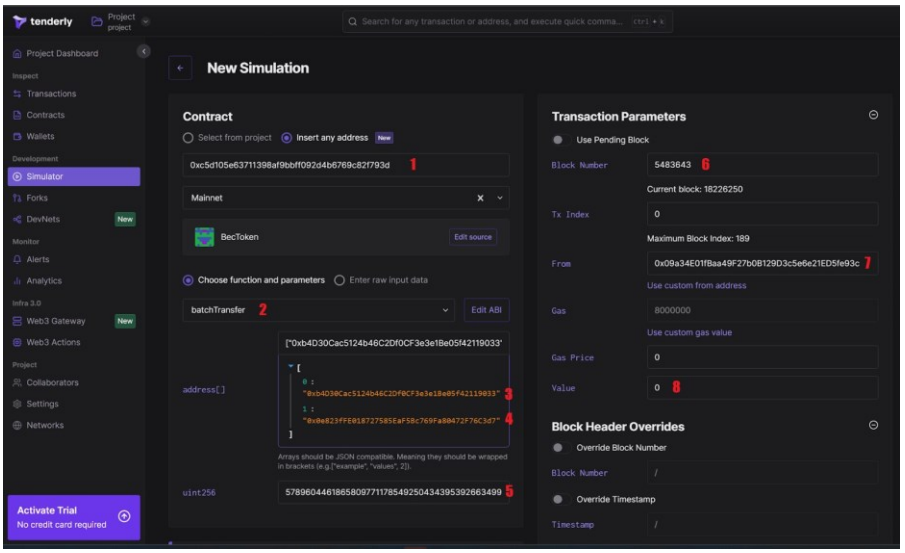

Figura 34: Dimostrazione dei parametri numerati da riportare da Etherscan in Tenderly

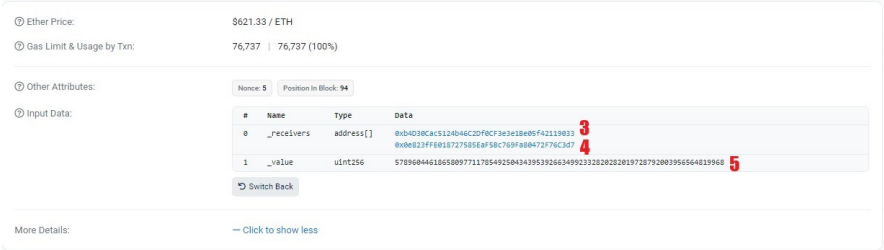

Figura 35: Dimostrazione dei parametri numerati da riportare da Etherscan in Tenderly

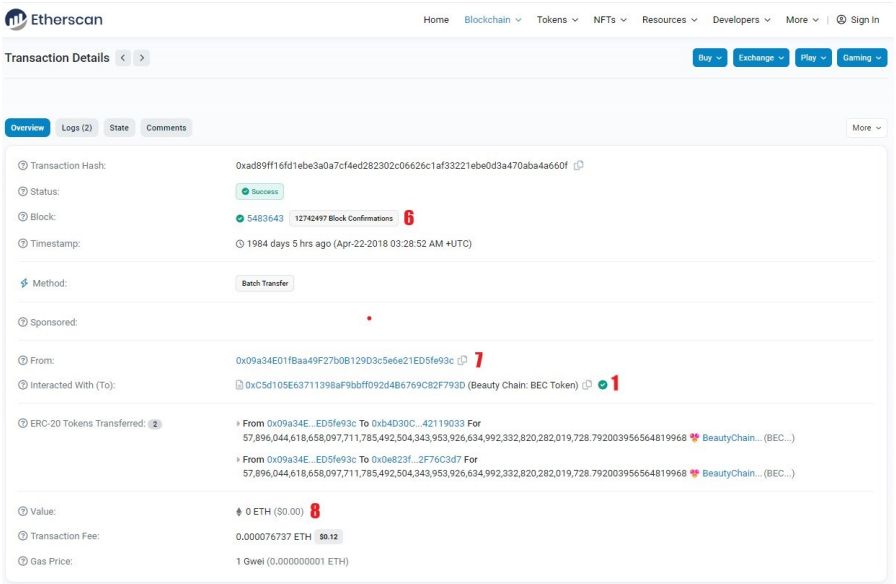

Figura 36: Dimostrazione dei parametri numerati da riportare da Etherscan in Tenderly

Come si può vedere dalle immagini i valori utilizzati per l'attacco sono stati reperiti dalla transazione registrata su Etherscan. Il risultato della simulazione è il seguente:

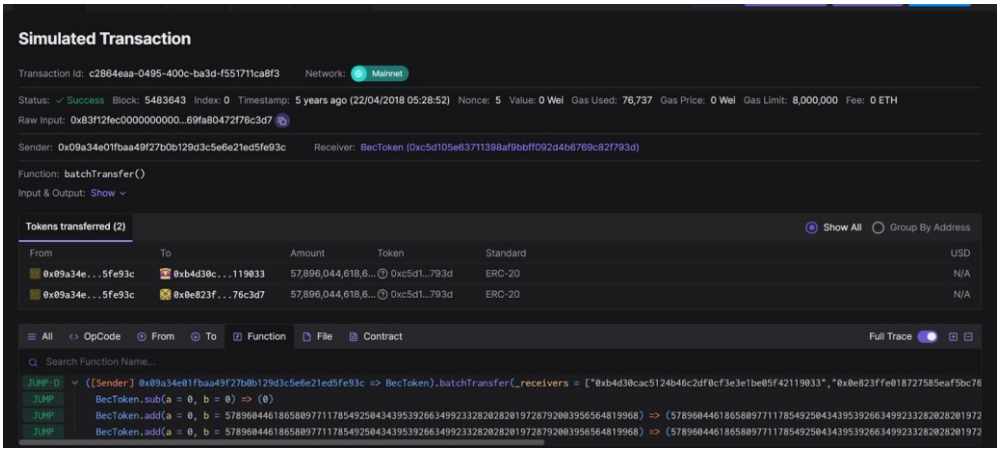

Figura 37: Schermata di Tenderly dove viene visualizzato il risultato

Dove nella sezione Tokens transferred possiamo vedere il quantitativo di tokens prelevati dall'attacco. Con Tenderly è inoltre possibile correggere il codice ed eseguire le stesse condizioni per verificare la risoluzione del problema.

In riferimento al contratto vulnerabile, proponiamo ora una possibile soluzione alla vulnerabilità. Nella sicurezza delle applicazioni, è fondamentale evitare di implementare manualmente le proprie librerie di sicurezza. Invece, si dovrebbero utilizzare delle librerie open source ben consolidate per i progetti di sviluppo. Ethereum non fa eccezione a questa regola ed è dotato di alcune librerie open source come quella di OpenZeppelin, che gestiscono una vasta gamma di aspetti, dai calcoli matematici sicuri all'autenticazione basata sui ruoli. Di seguito il link github alla libreria:[24]

per correggere il codice basterà importare la libreria in questo modo:

```
import https://github.com/OpenZeppelin/.../SafeMath.sol;
```

Dobbiamo poi aggiornare tutte le funzioni matematiche all'interno del nostro contratto e testarlo nuovamente. Con un rapido sguardo ci si convince che le istruzioni da modificare sono tre e devono essere aggiornate nel seguente modo.

```
uint256 amount = SafeMath.mul(uint256(cnt), _value);

balances[msg.sender] = SafeMath.sub(balances[msg.sender], amount);

balances[_receivers[i]] = SafeMath.add(balances[_receivers[i]], _value);
```

Rispettivamente riga 3, 8 e 10.

6.2 Unit Test Reentrancy

In questa sezione, esamineremo un modello di attacco di Reentrancy. Questi modelli errati possono dar luogo ad attacchi che consentono a un utente malintenzionato di svuotare il contratto dei suoi fondi con relativa facilità. L'errato ordine delle operazioni permette di evitare dichiarazioni obbligatorie che controllano se il saldo di un utente è sufficientemente alto per effettuare una transazione. Questo può essere sfruttato per eludere schemi logici errati e svuotare i fondi di un contratto.

Gli attacchi di reentrancy consentono a un utente malintenzionato di creare un ciclo tra un contratto di destinazione e un suo contratto di proprietà. Invece di provenire da un normale utente, la richiesta deriva dal contratto dell'attaccante, che non permette il completamento dell'esecuzione del contratto di destinazione fino a quando non vengono eseguite le azioni maliziose previste dall'attaccante. Solitamente, queste azioni comprendono lo svuotamento graduale dei fondi da parte dell'aggressore.

Per effettuare questo ciclo viene sfruttata una funzionalità del linguaggio Solidity ovvero la funzione di Fallback.Una funzione di fallback è una funzione predefinita in un contratto che viene chiamata quando non viene specificata nessun'altra funzione.

Figura 38: Schema riassuntivo Reentrancy

Nell'immagine sopra, vediamo il contratto target e il contratto dell'attaccante affiancati. L'attacco inizia con la chiamata alla funzione di withdraw. In seguito, la funzione di fallback viene invocata attraverso la transazione di withdraw, riportando il flusso all'inizio della funzione. Questo crea un ciclo tra withdraw e fallback fino a quando il contratto non dispone più di 1 Ether.

Caso reale il DAO Hack

L'attacco DAO è stato uno degli eventi più noti nella storia delle blockchain. Il DAO (Decentralized Autonomous Organization) era un fondo di venture capital che univa gli investitori di Ether per finanziare progetti in modo simile a una piattaforma di crowdfunding. Inizialmente, il progetto riusc`ı a raccogliere 12,7 milioni di Ether, che all'epoca corrispondevano a circa 150 milioni di dollari.
Il contratto DAO includeva una funzione chiamata "SplitDao," pensata per consentire agli utenti di prelevare fondi da un "child DAO" in caso di disaccordo su una decisione di maggioranza nell'impiego dei fondi. Tuttavia, all'interno di questa funzione "split," fu scoperta una vulnerabilità di tipo Reentrancy, che consent`ı all'attaccante di sottrarre 3,6 milioni di Ether. Questo era un importo considerevole, ma il problema principale fu la decisione della comunità Ethereum di annullare la transazione e restituire i fondi agli utenti. La decisione andò contro il principio di immutabilità delle blockchain, ma fu considerata necessaria data l'immaturità della rete all'epoca.
Questa è stata l'unica occasione in cui la rete Ethereum ha violato il principio di immutabilità della blockchain e ha annullato le transazioni. La decisione generò una divisione ideologica significativa nella comunità, che si tradusse in un hard fork della rete. La biforcazione ha dato origine a Ethereum Classic ed Ethereum, due catene blockchain separate. Ethereum Classic conserva la perdita di fondi originaria, mentre Ethereum ha mantenuto la transazione annullata, ed è la versione più utilizzata della rete oggi.
Di seguito possiamo vedere una versione ridotta della funzione SplitDAO originale che conteneva la vulnerabilità:

```
function splitDAO(
    uint _proposalID,
    address _newCurator
    noEther onlyTokenholders returns (bool _success)) {

    //Righe rimosse
    Transfer(msg.sender, 0, balances[msg.sender]);
```

```
        withdrawRewardFor(msg.sender);

    totalSupply -= balances[msg.sender];
    balances[msg.sender] = 0;
    paidOut[msg.sender] = 0;
    return true;
}
```

Se si osservano le righe 7-11, si può notare una violazione del modello ”Controllo -¿ Effetti -¿ Interazioni” che serve a prevenire attacchi di Reentrancy.

Nelle righe 7-8, il contratto effettua una chiamata withdrawal. Tuttavia, i withdrawals vengono aggiornati solo nelle righe 10-11 successivamente a questi prelievi. Questo significa che se un aggressore richiamasse la funzione splitDao mentre l’interazione è avvenuta sulla linea 8, sarebbe in grado di svuotare il contratto. I withdrawals non vengono mai aggiornati fino a quando il codice dell’aggressore non ha completato la sua esecuzione.

Una Reentrancy è relativamente semplice da correggere, ma altrettanto facile da commettere. È un errore che può verificarsi poiché la logica vulnerabile sembra corretta nell’ambito del mondo reale. Il codice vulnerabile funzionerebbe correttamente se non interagisse con un contratto dannoso. Tuttavia, non ci si aspetta che il contratto dell’attaccante sia il destinatario del prelievo, introducendo così un elemento inaspettato nella logica. Questo è il motivo per cui è essenziale riscrivere il codice utilizzando un modello di sicurezza quando si lavora con DApp e Web 3.0. [10][27][34]

Ora procediamo alla dimostrazione dello schema corretto di codifica cambiando l’ordine delle operazioni. L’immagine seguente mostra un codice vulnerabile ad un Reentrancy Attack e lo stesso codice corretto nella parte inferiore:

Come si può vedere dall’immagine qui sopra le righe 4-5 sono state invertite in modo da aggiornare prima il bilancio e poi effettuare il prelievo cos`ı da impedire la Reentrancy (Il codice qui sopra è semplificato e non è da intendere come sicuro in termine assoluti ma solo in termini di Reentrancy)

6.3 Unit Test Delegate Call Attack Vectors

Le Delegate Call vengono utilizzate per invocare funzionalità di un altro contratto, ma le modifiche influenzano solo il contesto del contratto chiamante. In sostanza, questa operazione è simile ad importare una libreria quando si sviluppano progetti complessi, consentendo quindi di utilizzare la funzionalità del contratto chiamato come parte integrante del proprio progetto. Questo concetto può essere mal inteso e condurre a vulnerabilità se non viene eseguito un controllo appropriato di ciò che accade in memoria.

```
function withdraw(uint withdrawAmount) public returns (uint) {
        require(withdrawAmount <= balances[msg.sender]);

        msg.sender.call.value(withdrawAmount)("");
        balances[msg.sender] -= withdrawAmount;

        return balances[msg.sender];
}
```

```
function withdraw(uint withdrawAmount) public returns (uint) {
        require(withdrawAmount <= balances[msg.sender]);

        balances[msg.sender] -= withdrawAmount;
        msg.sender.call.value(withdrawAmount)("");

        return balances[msg.sender];
}
```

Figura 39: Schema che mostra uno schema di risoluzione per Una Reentrancy

Esaminiamo ora in modo più dettagliato come le variabili vengono gestite all'interno delle delegate call e l'importanza di gestirle correttamente per evitare sovrascritture errate delle posizioni di memoria. Quando modifichiamo il valore di una variabile attraverso una delegate call, il valore della variabile nel contratto chiamante viene modificato. Questo può essere pericoloso e portare a risultati indesiderati, come vedremo nell'attacco al Parity Wallet alla fine di questa sezione.

Ora, esaminiamo un semplice esempio di cosa accade in memoria quando le variabili vengono gestite in modo errato con delegatecall.

Codice di esempio di archiviazione di DelegateCall:

```
pragma solidity 0.6.6;

  contract LogicContract {
      uint public a;

    function set(uint256 val) public {
      a = val;
```

```
    }
  }

  contract CallingContract  {
    uint 256 public b=5;
    uint256 public a = 5;
    address logic_pointer = address(new LogicContract());

    function setA(uint val) public {
       logicpointer.delegatecall
                  (abi.encodeWithSignature("set(uint256)", val));
     }
  }
```

Questo semplice esempio mostra un possibile errore di logica nella gestione della chiamata, in quanto da una prima valutazione si potrebbe pensare che, richiamando la funzione SetA, verrà modificata la variabile uint256 "a", ma non è così. Esprimiamo meglio il problema con un'immagine che mostra lo stack dei due contratti, mostrando come vengono inizializzate le memorie al momento del deploy del contratto.

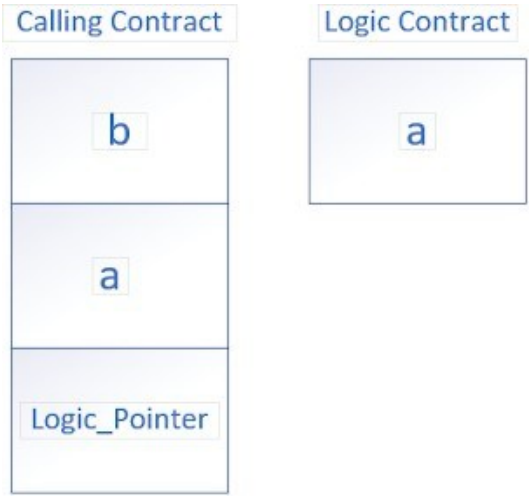

Figura 40: Schema che il comportamento dello stack in una Delegate Call

Quando viene chiamata una delegatecall il contratto chiamato sovrappone la sua memoria con il contratto chiamante. In questo caso il LogicContract ha solo una variabile "a" che verrà sovrapposta con il primo elemento nello stack del Calling Contract. In questo modo quando verrà invocata la funzione setA verrà modificato il valore di b.

Parity Wallet Attack

Nel mondo Web3 quando si parla di Delegate call Attack si fa riferimento ad uno degli attacchi più noti, l'hacking del Parity Wallet. Che ha causato perdite multimilionarie. Per

comprendere meglio come funziona uno di questi attacchi, esamineremo in dettaglio l'incidente di Parity Wallet.

Il contratto Parity vulnerabile a cui facciamo riferimento è visibile su Etherscan:[17]

Il Parity Wallet era un portafoglio multi-firma "molto leggero" che sfruttava le funzionalità di una libreria principale. L'utilizzo delle libreria rappresentava un modo per risparmiare sui costi, poiché i portafogli venivano implementati più volte sulla blockchain e le tariffe per l'implementazione dei contratti dipendevano dalle dimensioni delle istruzioni utilizzate nel contratto. Meno istruzioni in un portafoglio si traduce in costi di transazione complessivamente inferiori. Distribuendo la funzionalità principale all'interno di una libreria richiamabile, il codice comportava solo una tariffa iniziale per accedere alla codebase più grande. Ogni contratto aggiuntivo che veniva distribuito aveva un costo molto ridotto grazie ad istruzioni più concise. Questo approccio offriva vantaggi sia in termini di risparmio sui costi che di facilità di aggiornamento.
Tuttavia, il progetto Parity Wallet presentava alcune problematiche. Errori di autorizzazione che consentivano chiamate dirette alle funzioni, e una Delegate call che permetteva agli aggressori di interagire nel contesto del contratto chiamante.
Nello specifico ciò che ha permesso l'attacco è stato: una libreria che permetteva una Delegate Call in una funzione di FallBack, funzioni che non controllavano lo stato di inizializzazione corrente del wallet e alcune funzioni pubbliche che non richiedevano l'autorizzazione.
In questo attacco l'utente malintenzionato ha ottenuto il controllo della libreria tramite una funzione di inizializzazione pubblica. Una volta ottenuto il controllo della biblioteca tramite questa, l'aggressore è riuscito a inviare due transazioni. La prima è stata quella di acquisire la proprietà del contratto:[18]

Come si vede dai dati di input della transazione visibili su Etherscan è stata effettuata una chiamata alla funzione initWallet che ha sovrascritto i proprietari del contratto con l'indirizzo dell'aggressore contenuto nella variabile [4].

```
Function: initWallet
            (address[] _owners, uint256 _required, uint256 _daylimit)

MethodID:   0xe46dcfeb

[0]:   0000000000000000000000000000000000000000000000000000000000000060

[1]:   0000000000000000000000000000000000000000000000000000000000000000

[2]:   000000000000000000000000000000000000000000000116779808c03e4140000

[3]:   0000000000000000000000000000000000000000000000000000000000000001

[4]:   000000000000000000000000b3764761e297d6f121e79c32a65829cd1ddb4d32
```

I valori della transazione sono strutturati come spiegato nella sezione precedente. I seguenti sono i 5 parametri che seguono il MethodID di questa transazione sono definiti come segue:

0 Offset al valore della lunghezza dell'array dei proprietari

1 Quanti proprietari sono necessari

2 Limite di spesa giornaliero del contratto

3 Lunghezza array del proprietario

4 L'indirizzo dell'attaccante è l'unico valore presente nell'array

È stata poi inviata una seconda transazione :[19]
dove il valore _value contenuto in [1] è stato fornito all'indirizzo _to in [0]. Qui di seguito riporto i dati della funzione:

```
Function: execute (address _to, uint256 _value, bytes _data) ***

MethodID: 0xb61d27f6

[0]: 000000000000000000000000b3764761e297d6f121e79c32a65829cd1ddb4d32

[1]: 0000000000000000000000000000000000000000000000116779808 c03e4140000

[2]: 0000000000000000000000000000000000000000000000000000000000000060

[3]: 0000000000000000000000000000000000000000000000000000000000000000

[4]: 0000000000000000000000000000000000000000000000000000000000000000
```

La Funzione di Fallback che ha permesso l'attacco

All'interno del Parity Wallet era presente una funzione di fallback, che faceva uso di una Delegate call all'interno della libreria del portafoglio. Le funzioni di fallback vengono attivate quando viene effettuata una chiamata ad un contratto senza specificare il nome di una funzione durante un invio di valore. Sfruttando questa funzionalità, l'utente malintenzionato è riuscito a utilizzare la Delegate Call, senza specificare il nome della funzione ma includendolo nel "msg.data" con l'indirizzo dell'obbiettivo e i valori mostrati nelle transazioni precedenti.

Codice vulnerabile del Parity Wallet : [17]
Analizzando da riga 431 del codice sorgente dal link precedente, notiamo che la funzione di fallback si espone a tutte le funzioni pubbliche del portafoglio permettendo a chiunque possa invocarla di inviare dati tramite una Delegate Call nel contesto del contratto chiamante, come mostrato nella riga 436.

```
// gets called when no other function matches
        function () payable {
                        // just being sent some cash?
                        if (msg.value > 0)
                                Deposit (msg.sender, msg.value);
```

```
else if (msg.data.length > 0)
    _walletLibrary.delegatecall(msg.data);
```

Si noti che alla riga 435, la logica del codice stabilisce che se ci sono dati presenti nella transazione con una lunghezza maggiore di 0, viene effettuata una Delegate Call che richiama la libreria del portafoglio all'interno del contesto del contratto chiamante.
Per riassumere, per eseguire questo attacco è stato necessario effettuare una chiamata all'indirizzo del contratto non specificando il nome della funzione, attivando cos`ı la funzione di fallback e inserendo come payload l'msg.data mostrato in precedenza.
Questo attacco ha provocato perdite per 31 milioni di dollari agli utenti del Parity Wallet.

6.4 Analisi Automatica

Il metodo di analisi del codice mostrato nelle sezioni precedenti dimostra quanto possa essere complesso ispezionare singolarmente le funzioni di un progetto composto da diverse parti e centinaia di righe di codice. Per questa ragione, gli strumenti di analisi automatica rappresentano un prezioso alleato in questo processo.
Come è evidenziato dal paper "What are the Actual Flaws in Important Smart Contracts"[29] prodotto dal team di auditing Trail of Bits, esiste probabilmente un grande potenziale profitto nel fare un uso più efficace delle analisi, statiche e dinamiche automatiche per rilevare i peggiori problemi negli smart contract. Infatti sebbene utili, i test unitari da soli sembrano avere, nella migliore delle ipotesi, un impatto marginale per garantire la sicurezza.
Lo studio si concentra sull'analisi dettagliata dei risultati di audit su 23 smart contract. Gli audit sono stati eseguiti dal team a da altri partner del settore, utilizzando sia strumenti open-source che proprietari, nonché analisi manuale eseguita da esperti.
L'analisi si basa su un totale di 246 scoperte complessive. Le tabelle 1 e 2 riassumono le informazioni su queste scoperte. Ogni difetto è classificato in base alla sua gravità "severity", considerando l'impatto potenziale dell'exploit come segue:

- **Alta gravità:** se influisce su un gran numero di utenti o ha serie implicazioni legali e finanziarie;
- **Media gravità**: se influisce sulle informazioni individuali degli utenti o ha possibili implicazioni legali e un impatto finanziario moderato;
- **Bassa gravità:** se il rischio è relativamente basso o non è un rischio che il cliente ha indicato essere importante;
- **Informativo:** se il problema non rappresenta un rischio immediato ma è rilevante per le migliori pratiche di sicurezza;

Un'altra importante caratteristica di ciascuna scoperta è quanto sia difficile da sfruttare "difficulty:

- **Bassa difficoltà:** per i difetti comunemente sfruttati dove esistono strumenti pubblici o l'exploit può essere facilmente automatizzato;
- **Media difficoltà:** per difetti che richiedono una conoscenza approfondita di un sistema complesso;
- **Alta difficoltà:** per difetti in cui un attaccante deve avere un accesso privilegiato interno al sistema o deve scoprire altre vulnerabilità per sfruttarle.

Un'altra colonna mostra attenzione alle scoperte classificate come **"Hight-low"**, cioè alta gravità e bassa difficoltà. Queste offrono agli attaccanti un modo facile per infliggere danni potenzialmente molto gravi.[29]

			Severity					Difficulty			
Category	%	High-Low	High	Med.	Low	Info.	Und.	High	Med.	Low	Und.
data validation	36%	11%	21%	36%	24%	13%	6%	27%	16%	55%	2%
access controls	10%	25%	42%	25%	12%	21%	0%	33%	12%	54%	0%
race condition	7%	0%	41%	41%	6%	12%	0%	100%	0%	0%	0%
numerics	5%	23%	31%	23%	38%	8%	0%	31%	8%	62%	0%
undefined behavior	5%	23%	31%	15%	31%	8%	15%	15%	8%	77%	0%
patching	7%	11%	17%	11%	39%	28%	6%	6%	11%	61%	22%
denial of service	4%	10%	20%	30%	30%	20%	0%	50%	0%	40%	10%
authentication	2%	25%	50%	25%	25%	0%	0%	50%	0%	50%	0%
reentrancy	2%	0%	50%	25%	25%	0%	0%	50%	25%	0%	25%
error reporting	3%	0%	29%	14%	0%	57%	0%	43%	29%	29%	0%
configuration	2%	0%	40%	0%	20%	20%	20%	60%	20%	20%	0%
logic	1%	0%	33%	33%	33%	0%	0%	100%	0%	0%	0%
data exposure	1%	0%	33%	33%	0%	33%	0%	33%	33%	33%	0%
timing	2%	25%	25%	0%	75%	0%	0%	75%	0%	25%	0%
coding-bug	2%	0%	0%	67%	33%	0%	0%	17%	0%	83%	0%
front-running	2%	0%	0%	80%	0%	20%	0%	100%	0%	0%	0%
auditing and logging	4%	0%	0%	0%	33%	44%	22%	33%	0%	56%	11%
missing-logic	1%	0%	0%	0%	67%	33%	0%	0%	0%	100%	0%
cryptography	0%	0%	0%	0%	100%	0%	0%	100%	0%	0%	0%
documentation	2%	0%	0%	0%	25%	50%	25%	0%	0%	75%	25%
API inconsistency	1%	0%	0%	0%	0%	100%	0%	0%	0%	100%	0%
code-quality	1%	0%	0%	0%	0%	100%	0%	0%	0%	100%	0%

Severity and difficulty distributions for finding categories. The second column shows what percent of all findings that category represents; the remaining columns are percentages *within-category*.

Figura 41: Tabella che riporta i risultati ottenuti nell'elaborato [29]

La maggior parte di queste valutazioni è stata effettuata con strumenti di analisi statica e dinamica, oltre all'analisi manuale del codice, ma la fonte principale delle scoperte è stata l'analisi manuale. In particolare, è stata utilizzata una versione dell'analizzatore statico Slither,

Category	% Dynamic	% Static	Category	% Dynamic	% Static
data validation	57%	22%	logic	0%	0%
access controls	50%	4%	data exposure	0%	0%
race condition	6%	59%	timing	50%	25%
numerics	46%	69%	coding-bug	67%	50%
undefined behavior	0%	31%	front-running	0%	0%
patching	17%	33%	auditing and logging	0%	38%
denial of service	40%	0%	missing-logic	67%	0%
authentication	25%	0%	cryptography	0%	100%
reentrancy	75%	100%	documentation	0%	0%
error reporting	29%	14%	API inconsistency	0%	0%
configuration	0%	0%	code-quality	0%	67%

Optimistic percentages of each category detectable by automated methods.

Figura 42: Tabella che riporta i risultati ottenuti nell'elaborato [29]

che includeva una serie di rilevatori non disponibili nella versione pubblica. In alcuni casi, è stato utilizzato il testing delle proprietà con Echidna e l'analisi simboliche con Manticore per rilevare alcuni problemi. Solo due audit non hanno utilizzato strumenti automatizzati. Sedici degli audit hanno utilizzato Slither, sedici hanno utilizzato Manticore e tredici hanno utilizzato Echidna. Quando Slither è stato utilizzato, è stato impiegato in modo più esteso rispetto a Manticore o Echidna, che erano tipicamente limitati a alcune proprietà di grande interesse. [29]

6.5 Analisi Statica - Slither

Il testing degli smart contract rappresenta un processo essenziale per garantire l'affidabilità, la sicurezza e la corretta funzionalità dei contratti implementati su piattaforme blockchain distribuite. Lo scopo principale del testing è verificare che gli smart contract non solo operino secondo le aspettative, ma che siano anche privi di vulnerabilità o errori che potrebbero compromettere la loro integrità.
Sebbene ci siano stati moltissimi sforzi per costruire metodi migliori, esistono due rami principali per l'analisi del codice: **Dinamico e Statico**. Come si potrebbe dedurre dal nome, è facile intuire le caratteristiche di entrambi i metodi.
L'analisi dinamica del codice analizza un programma in esecuzione, mentre quella statica lo analizza quando non lo è.
L'analisi dinamica può sembrare la meno importante e diffusa da parte degli sviluppatori o dei programmatori, in realtà rappresenta un elemento ormai fondamentale del processo di produzione del software. Infatti fa parte delle tecniche di analisi dinamiche il "debugging". Questo processo implica la ricerca e la correzione degli errori in un programma per computer.

Uno dei principali vantaggi dell'analisi dinamica rispetto a quella statica è la sua capacità di rilevare gli errori con un alto grado di precisione. Ciò è dovuto alla sua focalizzazione sul processo in esecuzione e sui risultati del programma. Tuttavia, è importante notare che ci sono alcune categorie di errori che l'analisi dinamica non è in grado di individuare in modo efficace. Ecco perché è essenziale integrare anche l'analisi statica nel processo di ricerca.

Slither:A Static Analysis Framework For Smart Contracts Questa sezione introduce Slither[20], un framework di analisi statica open-source. Slither è stato progettato per fornire informazioni dettagliate sul codice dello smart contract e ha la flessibilità necessaria per supportare molte applicazioni. Attualmente, il framework è utilizzato per i seguenti scopi:

- Rilevazione automatica delle vulnerabilità: è possibile individuare una vasta gamma di bug nei contratti senza l'intervento dell'utente.
- Rilevazione automatica delle ottimizzazioni: Slither individua ottimizzazioni del codice che il compilatore non riesce a rilevare.
- Comprensione del codice: Slither riassume e visualizza le informazioni dei contratti per agevolare lo studio della base di codice.
- Revisione assistita del codice: attraverso la sua API, un utente può interagire con Slither.

Slither verifica i contratti utilizzando l'analisi statica in una procedura multistadio. Prende come input iniziale l'Abstract Syntax Tree (AST) di Solidity generato dal compilatore Solidity a partire dal codice sorgente del contratto. Nella prima fase, Slither recupera informazioni importanti come il grafo delle eredità del contratto, il grafo del flusso di controllo (CFG) e l'elenco delle espressioni. Successivamente, Slither trasforma l'intero codice del contratto in SlithIR, il suo linguaggio di rappresentazione interno. SlithIR utilizza l'analisi statica a valutazione singola (SSA) per facilitare il calcolo di una varietà di analisi del codice. Nella terza fase, di analisi effettiva del codice, Slither calcola un insieme di procedure predefinite che forniscono informazioni migliorate agli altri moduli.[20]

Come analizzare un contratto con Slither

Il modo più semplice per controllare un contratto è eseguire "slither [target]". Il target può essere specificato in diversi modi:

- Copiando in locale il sorgente del contratto. Esempio: slither SecureContract.sol
- Specificando la directory del progetto. Esempio: slither /percorso/del/progetto/Secure-Project
- Inserendo l'indirizzo del contratto Mainnet.
  Esempio: 0xf34960d9d60be18cC1D5Afc1A6F012A723a28811

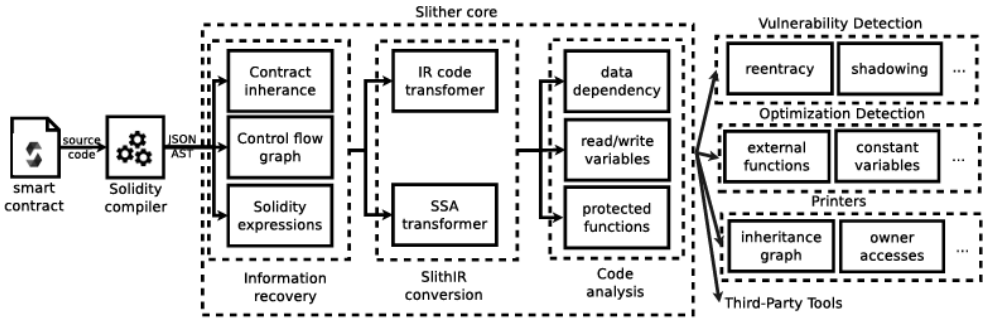

Fig. 1: Slither overview

Figura 43: Schema riassuntivo dell'architettura di Slither
[20]

Slither supporta ben 15 reti differenti tra cui citiamo: Ethereum, Kovan, Rinkeby, Goerli, ecc.. Facciamo ora un esempio di come è possibile analizzare un contratto vulnerabile ad attacchi di Reentrancy. Il contratto in questione è reperibile a questo link : [23]

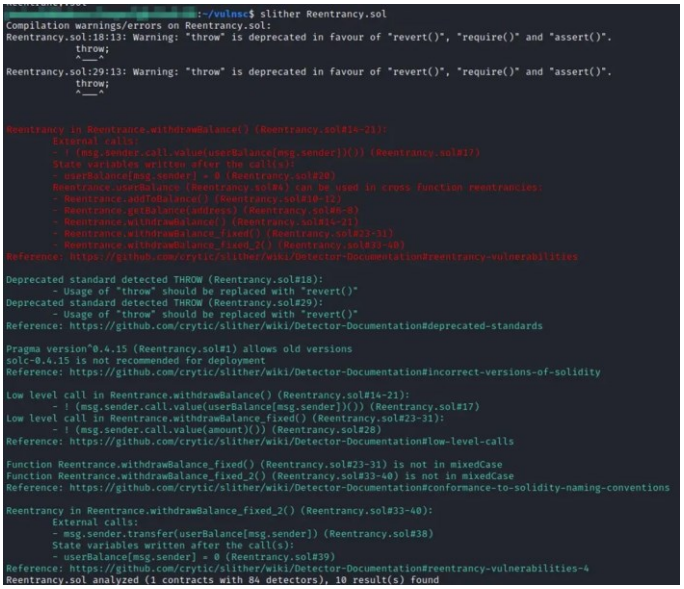

Figura 44: Screenshot funzionamento di Slither

L'output colorato evidenzia i risultati dell'analisi più utili. Slither fornisce anche una descrizione dettagliata della vulnerabilità, come funzionano, quali funzioni utilizzano e alcuni riferimenti utili.
I risultati dell'output possono essere filtrati
Ecco alcuni parametri che possono essere utilizzati per filtrare i risultati:

dipendenze: –exclude-dependencies
ottimizzazione: –exclude-optimization
informativo: –exclude-informational
risultati a basso impatto: –exclude-low

Il Detector è uno dei metodi più utilizzati in Slither e viene utilizzato quando si cerca un tipo specifico di vulnerabilità. Esistono 83 detector di vulnerabilità. Un Elenco dettagliato è ispezionabile in questa repository GitHub [26] o digitando slither -h. Per utilizzare uno specifico detector bisogna eseguire ”slither –detect [nome del detector]”.

Un’altra funzionalità utile di Slither sono le stampe, che possono aiutare a estrarre rapidamente informazioni cruciali sul contratto. Ciò può essere utile nell’analisi manuale, nei rapporti di audit, ecc.
Per controllare l’elenco completo delle possibili stampe che si possono effettuare basta eseguire ”slither –list-printers”.

Num	Printer	What it Does
1	call-graph	Export the call-graph of the contracts to a dot file
2	cfg	Export the CFG of each functions
3	constructor-calls	Print the constructors executed
4	contract-summary	Print a summary of the contracts
5	data-dependency	Print the data dependencies of the variables
6	declaration	Prototype showing the source code declaration, implementation and references of the contracts objects
7	dominator	Export the dominator tree of each functions
8	echidna	Export Echidna guiding information
9	evm	Print the evm instructions of nodes in functions
10	function-id	Print the keccak256 signature of the functions
11	function-summary	Print a summary of the functions
12	human-summary	Print a human-readable summary of the contracts
13	inheritance	Print the inheritance relations between contracts
14	inheritance-graph	Export the inheritance graph of each contract to a dot file
15	modifiers	Print the modifiers called by each function
16	pausable	Print functions that do not use whenNotPaused
17	require	Print the require and assert calls of each function
18	slithir	Print the slithIR representation of the functions
19	slithir-ssa	Print the slithIR representation of the functions
20	variable-order	Print the storage order of the state variables
21	vars-and-auth	Print the state variables written and the authorization of the functions

Figura 45: Screenshot funzionamento di Slither

Ecco alcuni esempi delle stampe: Eseguendo il comando “slither SecureContract.sol –print contract-summary, function-summary, modifiers” il tool mostrerà tutte informazioni su ciascuna funzione del contratto, la loro visibilità, i modificatori, le chiamate interne ed esterne.46

Inoltre, è presente un metodo per costruire grafici per visualizzare le interazioni delle funzioni all’interno del contratto. Basta eseguire “slither SecureContract.sol –print call-graph” e lo strumento genererà un file che può essere aperto.47

```
+ Contract Reentrance (Most derived contract)
  - From Reentrance
    - addToBalance() (public)
    - getBalance(address) (public)
    - withdrawBalance() (public)
    - withdrawBalance_fixed() (public)
    - withdrawBalance_fixed_2() (public)


Contract Reentrance
Contract vars: ['userBalance']
Inheritance:: []

+-----------------------------+-------------+-----------+-------------------------------+-----------------+-----------------+
|          Function           | Visibility  | Modifiers |             Read              |      Write      | Internal Calls  |
+-----------------------------+-------------+-----------+-------------------------------+-----------------+-----------------+
|    getBalance(address)      |   public    |    []     |       ['userBalance']         |       []        |       []        |
|       addToBalance()        |   public    |    []     |  ['msg.sender', 'msg.value']  | ['userBalance'] |       []        |
|                             |             |           |       ['userBalance']         |                 |                 |
|     withdrawBalance()       |   public    |    []     | ['msg.sender', 'userBalance'] | ['userBalance'] |       []        |
|  withdrawBalance_fixed()    |   public    |    []     | ['msg.sender', 'userBalance'] | ['userBalance'] |       []        |
| withdrawBalance_fixed_2()   |   public    |    []     | ['msg.sender', 'userBalance'] | ['userBalance'] |       []        |
+-----------------------------+-------------+-----------+-------------------------------+-----------------+-----------------+

+------------+-------------+------+-------+-----------------+-----------------+
| Modifiers  | Visibility  | Read | Write | Internal Calls  | External Calls  |
+------------+-------------+------+-------+-----------------+-----------------+
+------------+-------------+------+-------+-----------------+-----------------+


Contract Reentrance
+-----------------------------+-----------+
|          Function           | Modifiers |
+-----------------------------+-----------+
|         getBalance          |    []     |
|        addToBalance         |    []     |
|      withdrawBalance        |    []     |
|   withdrawBalance_fixed     |    []     |
| withdrawBalance_fixed_2     |    []     |
+-----------------------------+-----------+
Reentrancy.sol analyzed (1 contracts)
```

Figura 46: Screenshot funzionamento di Slither

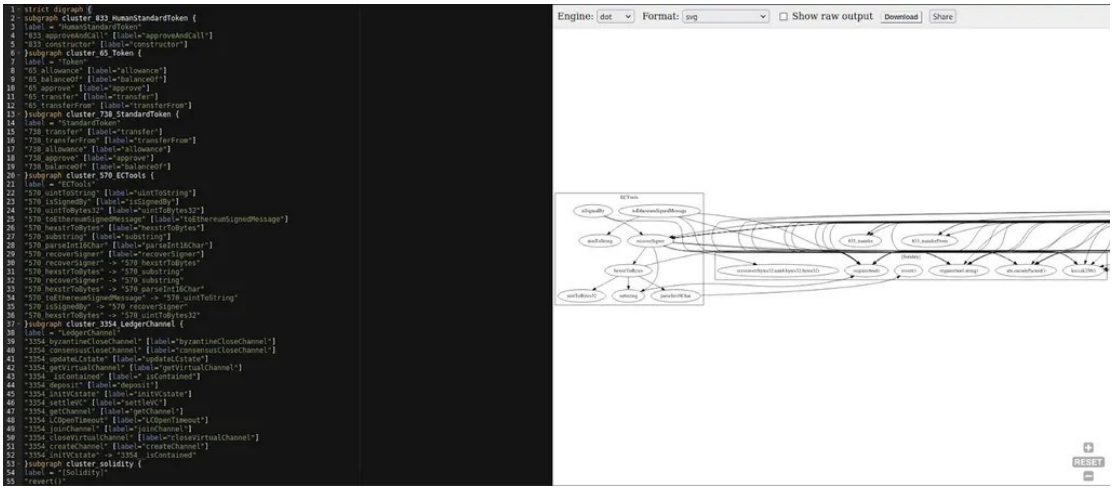

Figura 47: Screenshot funzionamento di Slither

Personalmente per questo utilizzo trovo più pratico i set di tool fornito dall'estensione per VS Code Solidity Visual Developer [36]che permette di visualizzare il grafico direttamente dall'IDE.

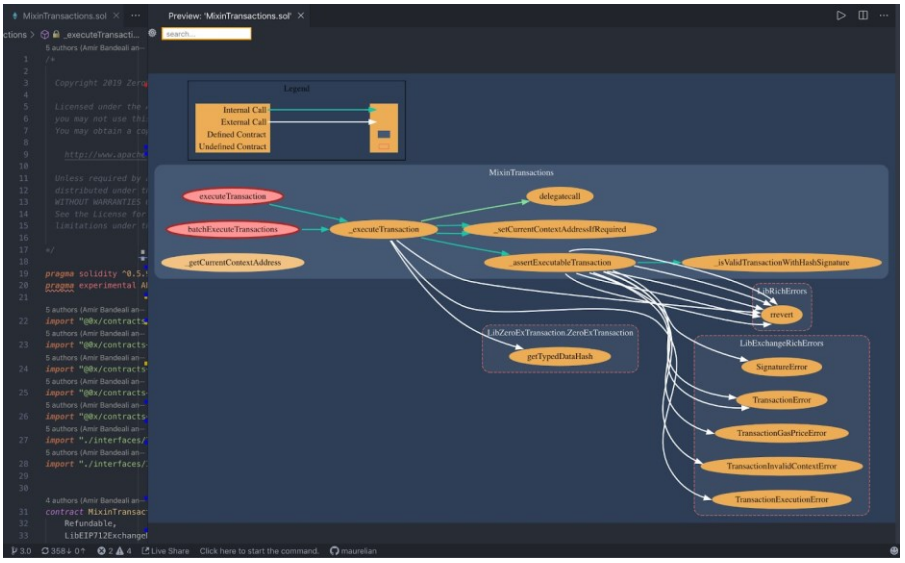

Figura 48: Screenshot funzionamento di Solidity Visual Developer

Riassumendo Slither, è un framework di analisi statica. E' veloce, robusto, accurato e fornisce informazioni dettagliate sugli smart contract. Il framework si basa su SlithIR, una rappresentazione intermedia progettata per l'analisi statica pratica del codice Solidity. Slither è l'unico strumento che può essere utilizzato contemporaneamente per trovare bug, suggerire ottimizzazioni del codice, aumentare la comprensione del codice di uno specifico contratto. È stato progettato per essere facilmente estendibile e servire come base su cui possono essere implementati strumenti di terze parti.[20]

6.6 Analisi Dinamica - Echidna

Gli smart contract per la blockchain di Ethereum, solitamente scritti nel linguaggio Solidity, facilitano e verificano transazioni finanziarie di alto valore, oltre a tracciare beni fisici e proprietà intellettuale. Pertanto, è essenziale che questi programmi siano corretti e sicuri, il che non è sempre il caso. Recenti lavori di rilevazione e catalogazione delle lacune nei contratti critici [29] hanno stabilito che il fuzzing, utilizzando proprietà definite dall'utente, potrebbe rilevare fino al 63% delle lacune più gravi e sfruttabili nei contratti. Questo parte suggerisce una forte necessità di un fuzzing di alta qualità e di facile utilizzo per gli sviluppatori di smart contract e gli auditor. Echidna è un fuzzer open-source per smart contract di Ethereum. Invece di basarsi su un insieme fisso di oracoli di bug predefiniti per rilevare vulnerabilità durante l'analisi, supporta tre tipi di proprietà:

- proprietà definite dall'utente per il testing basato su proprietà;

- verifica delle asserzioni;
- stima dell'uso di gas;

Attualmente, Echidna può testare sia smart contract scritti in Solidity che Vyper e supporta la maggior parte dei framework di sviluppo contrattuale, tra cui Truffle e Embark.
Il vantaggio di utilizzare Echidna risiede nel fatto che sia un tool facile da usare e configurare, che effettua un'ottima copertura del contratto o dello stato della blockchain e ha un'ottima velocità nel produrre risultati. Il fuzzing degli smart contract introduce alcune sfide insolite per lo sviluppo di un fuzzer. In primo luogo, viene richiesto un grande sforzo di ingegneria per rappresentare la semantica dell'esecuzione della blockchain. Che è una sfida diversa dall'esecuzione di binari nativi. In secondo luogo, si presenta la necessità di generare una sequenza di transazioni essendo i contratti di Ethereum calcolati tramite esse invece che con buffer di byte arbitrari.[28]

Fuzzing

Capito ora l'importanza di utilizzare strumenti di analisi automatica esaminiamo ora di questi metodi. Il fuzz testing, spesso noto come fuzzing , è una tecnica di test automatizzato del software utilizzata per identificare difetti dell'applicazione, rischi per la sicurezza e vulnerabilità. Utilizzando uno strumento di fuzzing, l'obiettivo è introdurre input errati, impropri o inattesi nell'applicazione e tenere traccia del modo in cui il sistema risponde ad essi, ad esempio, eccezioni, perdita di informazioni o arresti anomali dell'applicazione. I fuzzer per software tradizionale sono noti per essere strumenti efficienti per trovare bug.

Architettura Echidna

Questa Figura mostra l'architettura di Echidna divisa in due fasi: la pre-elaborazione e la campagna di fuzzing. Lo strumento inizia con un insieme di contratti forniti, più le proprietà integrate. Come primo step, Echidna sfrutta Slither, il framework di analisi statica presentato nelle precedenti sezionecitazione, per compilare e analizzare i contratti al fine di identificare costanti utili e funzioni che gestiscono direttamente Ether (ETH). Nel secondo step, inizia la campagna di fuzzing. Questo processo iterativo genera transazioni casuali utilizzando l'ABI fornita dal contratto. Quando viene rilevata una violazione di proprietà, viene automaticamente minimizzato un controesempio per segnalare la sequenza di transazioni più piccola e semplice che attiva il fallimento. Echidna può produrre un insieme di transazioni che massimizzano la copertura su tutti i contratti in modo facoltativo.[28]

Un elemento chiave del design di Echidna è rendere sostenibile il miglioramento continuo della funzionalità. Echidna dispone di un ampio set di test che verifica non solo la corretta esecuzione ma rileva anche errori introdotti deliberatamente per garantire che le funzionalità esistenti non vengano degradate, e utilizza le funzionalità del linguaggio Haskell per massimizzare l'astrazione e l'applicabilità del codice a nuovi approcci di testing.
La regolazione dei parametri con Echidna fornisce un gran numero di configurazioni che controllano vari aspetti del testing. Attualmente ci sono più di 30 impostazioni, controllate fornendo a Echidna un file di configurazione YAML. Tuttavia, per evitare di appesantire gli utenti con la complessità e rendere l'esperienza più fluida possibile, a queste impostazioni sono assegnati valori predefiniti attentamente scelti. L'impostazione con l'impatto più significativo è la scelta

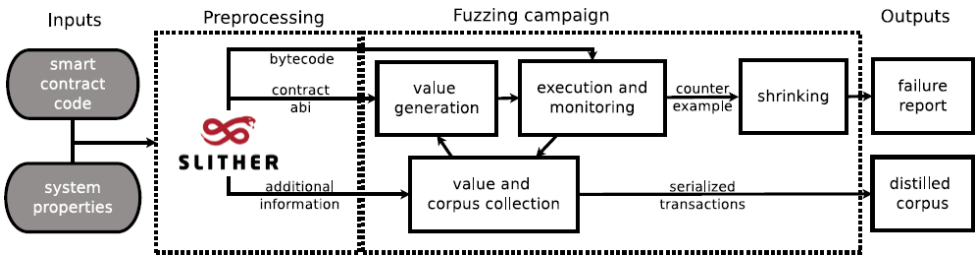

Figura 49: Schema riassuntivo dell'architettura di Echidna
[28]

del numero dei test generati.[28]

```
contract TEST {
  bool flag0; bool flag1;

  function set0(int val) public returns (bool) {
    if (val % 100 == 23)          { flag0 = true; } }

  function set1(int val) public returns (bool) {
    if (val % 10  == 5 && flag0) { flag1 = true; } }

  function echidna_flag() public returns (bool) {
    return(!flag1); }
}
```

(a) A contract with an echidna property.

```
Echidna 1.4.0.1
Tests found: 1
Unique instructions: 242
Unique codehashes: 1
Tests
echidna_flag: FAILED!

Call sequence:
1.set0(23)
2.set1(5)

Campaign complete, C-c or esc to exit
```

(b) A screenshot of the UI with the result of a fuzzing campaign

Figura 50: Esempio Echidna
[28]

Facciamo ora qualche esempio del suo funzionamento:
Consideriamo ora il seguente contratto vulnerabile ispirato volutamente dall'Ethernaut's Fal1Out challenge.

```
Fal1Out challenge
// ICO.sol
contract ICO {
    bool is_paused;
    address owner = msg.sender;

    functionICO() public {
        owner = msg.sender;
    }

    modifier isOwner() {
        require(owner == msg.sender);
        _;
```

```
    }

    function resume() public isOwner {
      is_paused = false;
    }
}
```

La funzione curriculum() può essere richiamata solo dal "owner" del contratto. Scriviamo un fuzz test per vedere se Echidna riesce a trovare un modo per rompere il contratto e impostare la variabile is_paused su true:

```
import "./ICO.sol";

// ICOFuzzTest.sol
contract ICOFuzzTest is ICO {

    constructor() public {
        is_paused= true;
        owner = address(0x0);
    }

    function echidna_test_pause() public view returns(bool)      {
        return is_paused == true;
    }
}
```

Il codice di inizializzazione va inserito nel costruttore.
Nel codice del contratto da analizzare, is_paused è impostato su true e il proprietario è impostato su 0x0.
Successivamente, scriviamo il test che definisce che is_paused dovrebbe sempre restituire true.
Per rompere questa proprietà Echidna utilizza in modo casuale i seguenti indirizzi chiamante: 0x10 , 0x20 e 0x30. E' importante ricordare che tutti i test fuzz devono iniziare con echidna_ affinché il tool li possa rilevare.
Per avviare il test fuzz, bisogna eseguire il seguente comando "echidna-test ICOFuzzTest.sol –contract ICOFuzzTest" dopo poco Echidna restituirà il seguente risultato.

```
Echidna 2.0.2
Tests found: 1
Seed: -7228999843977232232
Unique instructions: 118
Unique codehashes: 1
Corpus size: 1
Tests
echidna_test_pause: FAILED! with ReturnFalse

Call sequence:
1.IC0()
2.resume()
```

Figura 51: Screenshot Echidna

Il test è fallito poiché Echidna ha trovato un modo per violare l'invariante attraverso le seguenti chiamate:

1. IC0()
2. resume()

La funzione IC0() non è un costruttore valido in quanto è scritta con il numero '0' anziché la lettera 'O'. Inoltre, con Solidity ¿ 4.21, è necessario definire il costruttore utilizzando la parola chiave "constructor" anziché utilizzare il nome del contratto. Poiché IC0() è una funzione normale e non dispone di un modifier di accesso, chiunque può chiamarla per acquisire la proprietà del contratto. Dopo aver acquisito la proprietà, è possibile richiamare la funzione "curriculum" per impostare "is_paused" su "true".

Per quanto semplice possa sembrare l'esempio precedente, dimostra chiaramente come Echidna funziona: generando automaticamente transazioni arbitrarie (con input casuale) per testare l'invariante.

Facciamo un'altro esempio:

```
pragma solidity 0.7.0;

contract Pausable {
    bool is_paused;

    function paused() public {
        is_paused = true;
    }

    function resume() public {
        is_paused = false;
    }
}

contract Token is Pausable {
    mapping(address => uint256) public balances;

    constructor() {
        is_paused = true;
    }

    function transfer(address to, uint256 value) public {
        require(!is_paused);

        uint256 initial_balance_from = balances[msg.sender];
        uint256 initial_balance_to = balances[to];
```

```
        balance s[msg.sender] -= value;
        balances[to] += value;

        assert(balances[msg.sender] <= initial_balance_from);
        assert(balances[to] >= initial_balance_to);
    }
}
```

Il codice sopra utilizzato in Solidity è in versione 0.7.0. Per Solidity versione 0.8.0 e successive, il compilatore ha controlli integrati per underflow/overflow, ma per le versioni precedenti è necessario gestire queste situazioni.
Nel precedente esempio abbiamo utilizzato la modalità di test ”property”. Echidna supporta altre modalità di test come “assertion” e “dapptest”. In questo esempio, eseguiamo Echidna in modalità di test “assertion”. Echidna cerca così le asserzioni di Solidity nel codice da testare e cerca di falsificarle generando transazioni e input casuali.
Per avviare il fuzz test in modalità “assertion” eseguire: “echidna-test Token.sol –test-mode assertion –contract Token” in poco tempo verrà restituito il seguente risultato.

```
Echidna 2.0.2
Tests found: 5
Seed: 1867304744675579423
Unique instructions: 397
Unique codehashes: 1
Corpus size: 3
Tests
assertion in transfer(address,uint256): FAILED! with ErrorUnrecognizedOpcode

Call sequence:
1.resume()
2.transfer(0x0,1)

AssertionFailed(..): PASSED!

assertion in paused(): PASSED!

assertion in balances(address): PASSED!

assertion in resume(): PASSED!
```

Figura 52: Screenshot Echidna

Ha verificato le asserzioni in ogni funzione e ha trovato un test che non è riuscito nella funzione di trasferimento.
Il test è fallito dopo aver eseguito le seguenti chiamate:

1. resume()
2. transfer(0x0, 1)

Echidna ha prima chiamato curriculum per impostare is_paused = false, il che è necessario per accedere alla funzione di trasferimento. Nel secondo passaggio, è stata effettuata la chiamata a trasferimento con i parametri 0x0 per l’indirizzo e 1 per il valore. La riga del contratto che ha

causato il fallimento del test è questa:

```
balances[msg.sender] -= value;
```

Il saldo per msg.sender era 0 ed è stato ridotto di 1. Ciò ha provocato un underflow, che non è stato gestito poiché stiamo utilizzando Solidity versione 0.7.0. Di conseguenza, il saldo di msg.sender è stato impostato al numero massimo di uint256. Questo comportamento viola la prima affermazione:

```
assert(balances[msg.sender] <= initial_balance_from);
```

poiché ora il nuovo saldo è molto superiore a quello iniziale.

Echidna è un efficace, facile da usare e veloce fuzzer per smart contract sulla blockchain Ethereum. Fornisce un'esperienza di fuzzing potente già pronta all'uso con una minima configurazione o preparazione, ma consente una notevole personalizzazione. Echidna supporta il controllo delle asserzioni, il controllo personalizzato delle proprietà e la stima dell'uso massimo di gas. Utilizza un set di funzionalità principali basato sull'esperienza acquisita tramite audit di sicurezza dei contratti. I parametri di generazione di test predefiniti di Echidna sono stati calibrati utilizzando l'esperienza nel mondo reale degli audit commerciali e mediante esperimenti su benchmark e analisi delle mutazioni svolta dal team Trail of Bits.[28]

6.7 L'insufficienza dei test Unitari

Come abbiamo visto ad inizio capitolo è buona pratica ottenere una copertura dei test unitari di tutto il contratto, ma questo da solo non è sufficiente. La qualità dei test può essere notevolmente migliorata utilizzando metriche di copertura più avanzate, come il punteggio del test di mutazione, che si dimostra più efficace rispetto alle metriche più comuni come la copertura delle istruzioni o dei rami, ma ancora non è sufficiente.
Un essere umano può scrivere solo una quantità limitata di test unitari, coprendo solo un piccolo numero di input possibili all'interno dell'enorme spazio di tutte le possibili combinazioni.
Come dimostrato dall'articolo "What are the Actual Flaws in Important Smart Contracts (and How Can We Find Them)?"[29] presentato nelle precedenti sezioni non esiste una correlazione statisticamente significativa tra la qualità dei test unitari e il numero di vulnerabilità gravi rilevate nei codici sorgente.
Sebbene i test unitari siano utili, sembrano avere solo un impatto marginale sulla garanzia di sicurezza. Quello che voglio presentare in questa sezione è il mio "property-based tester" ispirato dal progetto Echidna[28].Andremo quindi ad analizzare le tecnologie e le scelte progettuali effettuate di questo Fuzzer.

Un property-based tester generalizza i test unitari manualmente scritti. Il tester è un fuzzer che effettua una verifica automatica sulla base di proprietà o invarianti, definite in forma di dichiarazioni matematiche o logiche, che per il programma in analisi dovrebbero essere sempre soddisfatte. Per entrare nel dettaglio facciamo un esempio del funzionamento: consideriamo un

programma che permette ad un utente A di inviare un importo x di token a un altro utente B. Innanzitutto, lo sviluppatore scrive alcune funzioni chiamate invarianti che dovrebbero sempre restituire true se il programma è stato implementato correttamente. L'invariante definisce che l'invio di token avvenga solo quando il saldo iniziale di A risulta superiore o uguale all'importo richiesto x. A questo punto il fuzzer, prende come input sia il software da controllare che l'invariante, generando migliaia di test casuali e verificando ad ogni test che l'invariante sia rispettata. È preferibile scrivere inizialmente l'invariante in linguaggio naturale (ad esempio, in inglese) e successivamente tradurla accuratamente in codice.

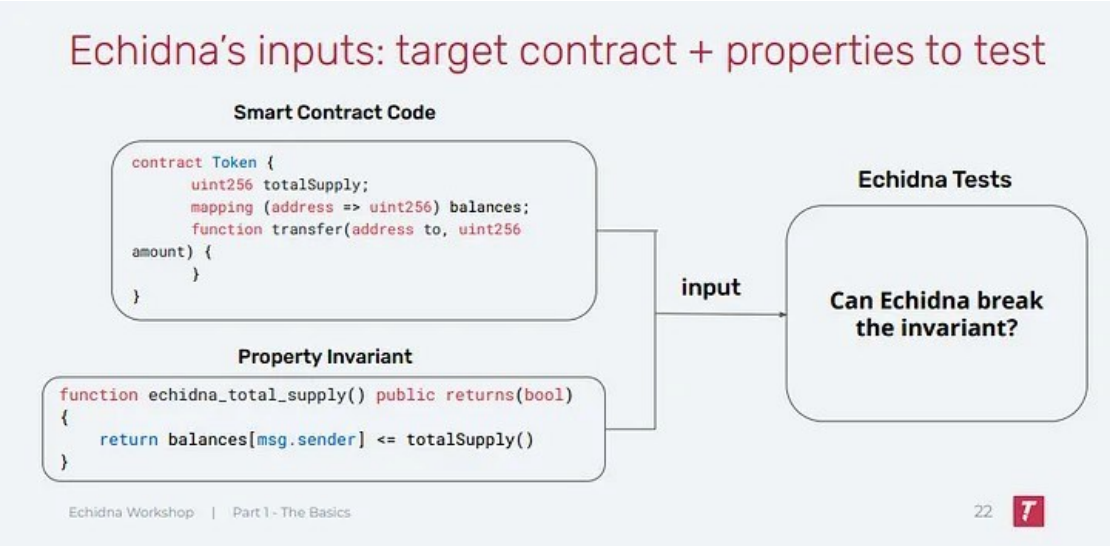

Figura 53: Schema riassuntivo Echidna

Dopo aver definito tutti gli invarianti, si avvia il fuzzer, che prende sia il software da testare che gli invarianti come input. Il fuzzer genera migliaia di test in modo casuale e verifica ciascun invariante rispetto a tutti questi test. Gli input per i test non sono del tutto casuali, alcuni di essi possono essere scelti uniformemente in un ampio intervallo, ma spesso il fuzzer opta per concentrarsi su valori estremi, come 0 o il valore massimo di un tipo di dato unsigned integer. In questo modo, vengono automaticamente testati diversi casi limite che potrebbero spesso causare overflow o altri problemi. Questo approccio può essere paragonato ai casi di test creati manualmente dagli esseri umani, che tendono a concentrarsi maggiormente su "percorsi felici" e meno su situazioni limite insolite. Questo rappresenta uno dei vantaggi significativi del fuzzing rispetto ai test unitari, poiché molti bug si verificano proprio nei casi limite, che non sono sempre intuitivi per gli sviluppatori.

Nel contesto del mio progetto, i test generati consistono in sequenze di transazioni inviate al contratto. Dopo ogni singola transazione, tutti gli invarianti vengono verificati e alla fine della sequenza, che ha una lunghezza massima stabilita dall'utente, vengono nuovamente verificate tutte le invarianti. Se alla fine della verifica tutte le proprietà rispondono positivamente, lo stato della blockchain locale viene ripristinato e un'altra sequenza di transazioni verrà inviata al nodo locale.

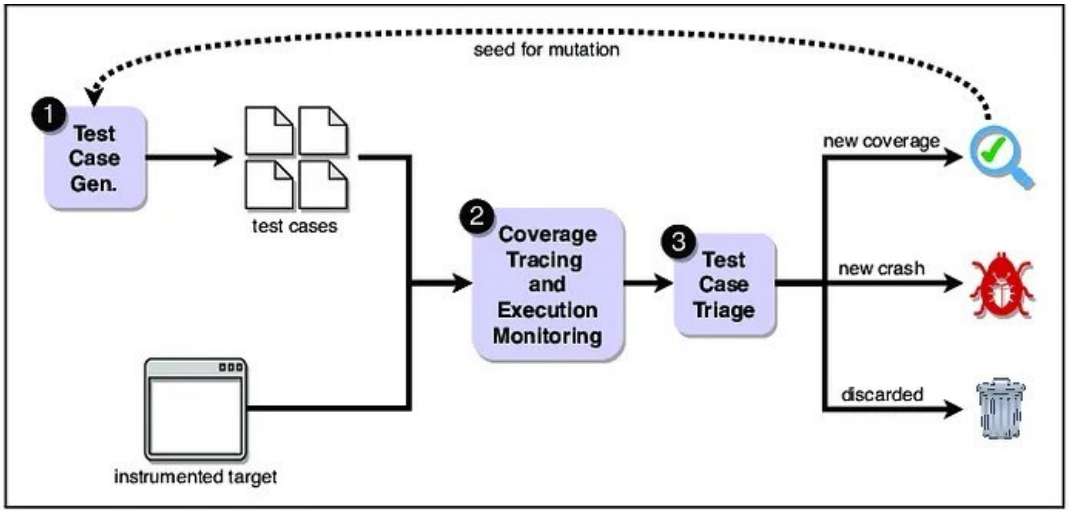

Figura 54: Schema riassuntivo Echidna

Questo ciclo verrà ripetuto molte volte finché non verrà rilevato un riscontro negativo negli invarianti, o fino alla generazione di un numero di test sufficiente a dimostrare la qualità del codice.

Se si scopre che alcuni invarianti sono fragili, c'è una fase finale chiamata "shrinking" o restringimento. Questa fase è molto utile per comprendere l'origine del bug. L'obiettivo del restringimento è quello di trovare il caso di test più semplice che causa la rottura dell'invariante. Ad esempio, se la rottura è causata da un intero senza segno, il restringimento troverà l'intero più piccolo responsabile dell'attivazione. Un altro esempio potrebbe essere trovare la sequenza con il numero minore di transazioni responsabili della falsificazione dell'invariante, nel caso in cui siano necessarie diverse chiamate.

6.8 Property-Based Tester

In questa sezione andrò a presentare il Property-Based Tester sviluppato in Python per rilevare vulnerabilità negli smart contract. Il motore di mutazione che genera i casi di test è basato sulla libreria Hypothesis in Python. Per quanto riguarda l'utilizzo della rete di test la mia scelta si è concentrata su Anvil, testenet selezionata dal framework Foundry, invece della più comune Ganache. Questa scelta ha portato un miglioramento in media nella velocità del 40%.

La libreria Hypothesis

La libreria Hypothesis ci consente di descrivere come appaiono gli input invece di specificare cosa sono. Quindi genera molti casi di test con input corrispondenti a quella descrizione. Il framework si arresta quando incontra un test case fallito e genera un errore specificando quale

input ha bloccato il codice.

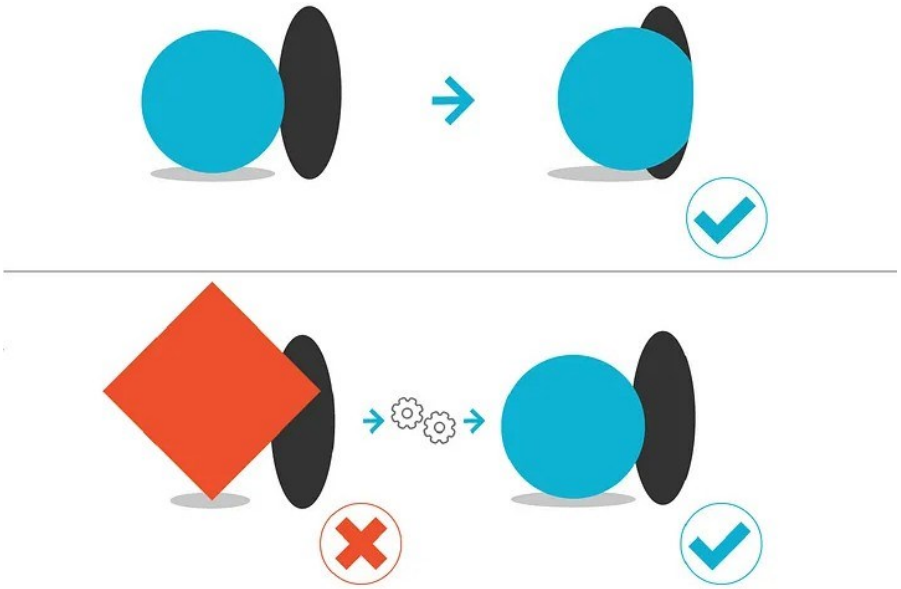

Figura 55: Grafica che descrive il funzionamento di Hypothesis

Facciamo ora un semplice esempio per comprendere meglio il funzionamento. Prendiamo in esame una funzione “divide” e una semplice funzione di test ”test divide”. In questo semplice esempio il codice del test case viene eseguito correttamente e sembra che la nostra funzione lavori correttamente.

```
def divid e(x,y):
    return x / y
def test_divide():
    divid e(1, 2)
```

Ora riscriviamo la funzione “test_divide” con il framework Hypothesis chiamandola “test_divide_hypothesis” :

```
from hypothesis import given, settings
from hypothesis import strategies as st

@given(st.floats(),st.floats())
def test_divide_hypothesis(x,y):
    divid e(x, y)
```

La libreria genererà più input per testare il codice e in tal caso riporterà un “ZeroDivisionError” per gli input (x = 0.0, y = 0.0). Questo significa che la funzione “divide” non gestisce questo caso correttamente.

Per utilizzare la libreria Hypothesis è necessario:

- importarle, come si vede le codice qui sopra;
- Utilizzare il decoratore @given davanti alla funzione di test;
- Descrivere i dati di input con strategie. Hypothesis supporta i tipi di dati primitivi e fornisce anche la possibilità di crearne dei complessi.

Altri decoratori utili sono @reproduce_failure che riproduce un caso di test che fallisce sempre e @setting che permette di attribuire al decoratore @given alcuni esempi più espliciti. Maggiori informazioni sulla libreria Hypothesis possono essere trovate sulla documentazione ufficiale.[22]

Iniziamo ora con una breve descrizione del processo di installazione e configurazione del mio tool. Per poter iniziare è richiesta l'installazione dei seguenti prerequisiti:

```
hypothesis==6.53.0
web3==6.0.0
solc-select==0.2.1
crytic-compile==0.2.4
typer==0.7.0
slither-analyzer==0.9.0
art==5.9.0
argparse==1.4.0
datetime==5.1.0
```

Il tutto è possibile tramite il comando "pip install -r requirements.txt" successivamente è richiesto l'installazione e la configurazione della versione in uso del compilatore solc con i comandi "solc-select install 0.8.19" "solc-select use 0.8.19". Come ultimo passaggio bisogna scaricare e attivare la testnet. Può essere fatto copiando la repository ufficiale dal github ufficiale:[21] o tramite il comando "curl -L https://foundry.paradigm.xyz — bash". Per permettere al tool di funzionare è necessario far partire la rete eseguendo "foundryup" e digitando "Anvil" si dovrebbe avere una schermata di questo tipo.

```
$ anvil

                             _   _
                            (_) | |
      __  _    _ __   __   __  _  | |
     / _` | | '_ \  \ \ / / | | | |
    | (_| | | | | |  \ V /  | | | |
     \__,_|_|_|_|_|   \_/   |_|_|_|

    0.1.0 (8d507b4 2023-08-05T00:20:34.048397801Z)
    https://github.com/foundry-rs/foundry

Available Accounts
==================

(0) "0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266" (10000 ETH)
(1) "0x70997970C51812dc3A010C7d01b50e0d17dc79C8" (10000 ETH)
...

Private Keys
==================

(0) 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80
(1) 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d
...

Listening on 127.0.0.1:8545
```

Deploy

Il deploy del contratto sulla rete Anvil avviene grazie alla libreria web3 tramite il seguente codice:

```
from web3 import Web3, HTTPProvider, Account
from web3._utils.method_formatters import BlockNotFound,
                                         TransactionNotFound
from AnvilSetUp import fixture_anvil
from abi import get_abi_and_bytecode, get_abi_by_name


def deploy_contract(w3, anvil, contract_names, test_file_name):
    """Deploy del contratto sulla test net local di Anvil."""
    abis, bytecodes = get_abi_and_bytecode(test_file_name)
    targets = []
    deployed_abis = []
    for contract in contract_names:
        abi = abis[contract]
        bytecode = bytecodes[contract]
        signed_txn = w3.eth.account.sign_transaction(
```

```
            dict(
                nonce=w3.eth.get_transaction_count(anvil.eth_address),
                maxFeePerGas=20000000000,
                maxPriorityFeePerGas=1,
                gas=15000000,
                to=b"",
                data="0x" + bytecode,
                chainId=1,
            ),
            anvil.eth_privkey,
        )
        tx_hash =
            w3.eth.send_raw_transaction(signed_txn.rawTransaction)
        time.sleep(
            0.5
        )
    # per evitare errori di connessione durante la distribuzione
    # dei contratti sul nodo locale
        address =
            w3.eth.get_transaction_receipt(tx_hash)["contractAddress"]
        target = w3.eth.contract(address, abi=abi)

        if "setUp" in target.functions:
         # Memorizza con quale contratto ha effettuato il deploy
         # La rimuove dalle future funzioni da fuzzare
          func = target.functions["setUp"]
          func().transact({"from": w3.eth.default_account.address})

         # Il fuzzer prende solo in considerazioni
         # funzioni di invarianti denominate setUp
           for info in target.abi:
             if info["type"] == "function"
                              and info["stateMutability"] == "view":
               for ret in info["outputs"]:
                 internal_type = ret["internalType"]
                 if internal_type.startswith("contract"):
                   deployed = target.functions[info["name"]]().call(
                     {"from": w3.eth.default_account.address}
                         )

                  # TODO Verificare la problematica degli omonimi
                  contract_name = internal_type.split(" ")[1]
                  deployed_abi = get_abi_by_name(contract_name, test_file_name)
                  targets.append(
                      w3.eth.contract(abi=deployed_abi, address=deployed)
                  )

            targets.append(target)
```

```
    return targets
```

Il tool prende in input un file in estensione .sol dove all'interno deve essere presente il contratto da analizzare che deve avere lo stesso nome passato come parametro e il contratto InvariantTest che dovrà importare il contratto da analizzare e contenere le invarianti.
Per spiegare come funziona lo strumento iniziamo ora con un primo esempio:

```
contractInvariantBreaker {
bool public flag0 = true;
bool public flag1 = true;

function set0(int256 val) public returns (bool) {
if (val % 100 == 0) flag0 = false;
return flag0;
}

function set1(int256 val) public returns (bool) {
if (val % 10 == 0 && !flag0) flag1 = false;
return flag1;
}
}

contract InvariantTest {
InvariantBreaker public inv;

function setUp() public {
inv = new InvariantBreaker();
}

function invariant_neverFalse() public returns (bool)      {
return inv.flag1();
}
}
```

Questo codice contiene 2 contratti: il primo, InvariantBreaker, è il contratto che vogliamo verificare. Il secondo, InvariantTest, è il contratto di supporto responsabile della configurazione delle proprietà che vogliamo verificare. Il setup di test: viene utilizzato prima per distribuire il nostro contratto tramite la funzione setUp(). In questo modo l'invariante che vogliamo controllare dopo ogni transazione verrà descritta nella funzione invariant_neverFalse().

I parametri di configurazione predefiniti sono sei e sono contenuti e modificabili direttamente dal file config.yaml

- shrinking: true In modalità True il tool cercherà di trovare il controesempio più semplice se un invariante viene violato;

- swarm_testing: true In modalità True il tool farà un'indagine più approfondita del codice per la scoperta di ulteriori bug;
- coverage_guidance: true In modalità True il tool farà un'esplorazione più approfondita del codice;
- constants_mining: true In modalità True il tool farà delle rilevazioni di costanti nel codice;
- favor_long_sequence: true In modalità True il tool genererà con maggiore frequenza delle sequenze di transazione molto lunghe;
- anvil_port: 8545 La porta usata dalla testnet di Anvil

Vi sono inoltre altri due parametri che devono essere esplicitati tramite riga di comando quando viene lanciato il programma. Questi sono "–run" Il numero di sequenze di transazioni da generare durante il fuzzing, e "–sequence" il numero massimo di transazioni che una singola sequenza può contenere. Questi due parametri sono i più importanti. Impostandoli rispettivamente a 1000 e 100 il fuzzer proverà a generare 1000 sequenze contenenti fino a 100 transazioni ciascuna. Tra ogni sequenza, lo stato della blockchain viene ripristinato al suo stato iniziale, ovvero quello definito dopo l'esecuzione di setUp() all'inizio della campagna fuzzing. Per i casi più difficili e per avere maggiore fiducia nella sicurezza, è utile aumentare questi due parametri. Gli autori di Echidna hanno scoperto che un parametro seq_len (dimensione massima di una sequenza di transazioni) fino a 300 è ottimale, mentre fuzz_runs (numero di sequenze di transazioni) potrebbe essere aumentato di uno o due ordini di grandezza, ma in questo modo aumenterà anche il tempo di elaborazione dello stesso fattore e potrebbe non essere pratico. Il passaggio di parametri è stato gestito tramite la libreria argparse nel seguente modo:

```
def get_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "-r",
        "--run",
        dest="run",
        type=int,
        help="Inserire il numero ... fuzzing",
    )
    parser.add_argument(
        "-s",
        "--sequence",
        dest="sequence",
        type=int,
        help="Inserire il numero massimo ... contenere",
    )
    parser.add_argument(
        "-c",
        "--contract",
        dest="contract",
        type=str,
        help="Inserire il contratto da analizzare",
    )
```

```
    options = parser.parse_args()
    if not options.run:
        parser.error("[-] Specificare il ... transazioni.")
    if not options.sequence:
        parser.error("[-] Specificare lil ... transazioni.")
    if not options.contract:
        parser.error("[-] Specificare il ...analizzare.")
    return options.run, options.sequence, options.contract
```

Lanciamo ora il nostro primo test sul semplice contratto mostrato sopra con il comando "python3 lysmata.py -r 1000 -s 100 -c tests/invariant breaker.sol".

Figura 56: Grafica Lysmata Property-Based Tester

Il fuzzer dovrebbe essere in grado di trovare un controesempio decisivo in meno di 1 minuto. Si dovrebbe ottenere un output simile a:

```
Falsifying example: composite test(
    ops=[
     (<Function set1>, 0),
     (<Function set1>, 0),
     (<Function set1>, 0),
     (<Function set1>, 0),
     (<Function set1>, 0),
     (<Function set1>, 0),
     (<Function set1>, 0),
     (<Function set1>, 0),
     (<Function set1>, 0),
     (<Function set1>, 0),
     (<Function set0>, 0),
     (<Function set1>, 0)],
)
Invariante rotta !!!
```

Si può osservare dalle seguenti righe "(¡Function set0¿, 0), (¡Function set1¿, 0)]", che il fuzzer chiama set0(0) e poi set1(0). Andando a verificare nel contratto InvariantBreaker questa è la sequenza che serve per impostare la variabile di stato flag1 su false , infrangendo quindi la proprietà che volevamo testare!

Il processo di contrazione di Hypothesis non è molto performante quando si tratta di test complicati come elenchi di transazioni, e lo strumento non sarà in grado di trovare l'esempio minimo in questo caso:

```
(<Function set0>, 0),
(<Function set1>, 0)
```

La libreria descrizione di Hypothesis è stata implementata nel seguente modo:

```
@settings(
        max_examples=fuzz_runs,
        phases=phases_tuple,
        deadline=None,
        suppress_health_check=list(HealthCheck),
    )
    @given(ops=operations_lists_strategy())
    def composite_test(ops):
        seqCoverage = set()
        nonlocal snapshotID
        nonlocal num_examples
        nonlocal current_max

        def update_coverage_frequency(seqCov):
            nonlocal current_max
            if coverage_guidance:
                CounterCoverage.update(seqCov)
                covered_paths = [CounterCoverage[ID] for ID in seqCov]
                if(
                    len(covered_paths) > 0
                ):
        # per evitare rari errori di instabilita' con hypothesis
                    if 1 in covered_paths:  # nuova path scoperta
                      selected_IDs = [
                       ID for ID,
                       count in CounterCoverage.items() if count == 1
                       ]
                      for ID in selected_IDs:
                        dico_first_seen[ID] = num_examples
                        current_max = num_examples
                        target(num_examples)
                    else:
                      selected_values =
                            [dico_first_seen[ID] for ID in seqCov]
                      max_value = max(selected_values)
                      if current_max == max_value:
                       target(num_examples)
                      else:
                       target(max_value)
```

Il decoratore @setting prende in input le configurazione dei parametri precedentemente impostati, mentra la strategia è implementata tramite il decoratore @given "operations_list_strategy()". La funzione "composite_test(ops)" svolge un ruolo principale nella strategia, prendendo come unico parametro "ops" che è una lista di valori che vogliamo esaminare.

Il tool è anche in grado di restituire un controesempio minimizzato corretto. Anche in questo caso il restringimento potrebbe fallire. Eseguiamo dunque il seguente comando "python3 RBSM_lysmata.py -r 1000 -s 100 -c tests/invariant_breaker.sol".

```
Esempio di falsificazione:
state = StateFulFuzzer()
state.reset_evm_state()
state.invariant_neverFalse()
state.set0(arg=0)
state.invariant_neverFalse()
state.set1(arg=0)
state.invariant_neverFalse()
state.teardown()
Invariante rotto
```

Come si può vedere, il tool rileva che sono necessarie solo due transazioni: set0(arg=0) seguita da set1(arg=0).

Swarm Testing

Lo "Swarm Testing" è un approccio alla qualità del software che prende spunto dall'organizzazione delle api in uno sciame. Poiché in base al teorema "no free lunch" sull'ottimizzazione, viene dimostrato che non esiste una configurazione ottimale adatta a risolvere ogni problema, nello Swarm Testing vengono coinvolti una grande quantità di test, invece di utilizzarne uno, o un piccolo campionario. L'idea della tecnica dello Swarm è quella di variare alcuni parametri del fuzzer in diversi passaggi del processo, per poter trovare bug più in profondità. Tali parametri potrebbero essere la lunghezza delle transazioni, il sottoinsieme di funzioni chiamate durante una sequenza di transazioni, ecc. Ecco come l'ho implementato nel codice:

```
@st.composite
 def operations_list_strategy(draw):
     # Generate a random subset of operations
      if swarm_testing:
          min_size_sampled =
            len(operations)-draw(st.integers(0,len(operations)-1))
          unique_operations =
            st.sets(st.sampled_from(operations),
            min_size=min_size_sampled)
      else:
          unique_operations = st.just(operations)
      selected_operations = draw(unique_operations)
      if favor_long_sequence:
```

```
            min_seq_len_sampled =
              seq_len-draw(st.integers(0,seq_len-1))
        else:
            min_seq_len_sampled = 1
        selected_ops =
          st.lists(st.one_of(selected_operations),
          min_size=min_seq_len_sampled, max_size=seq_len)
        return draw(selected_ops)
```

Quando viene abilitata la modalità "swarm_testing", viene selezionato solo un sottoinsieme di funzioni, chiamate "operations", che viene selezionato dalla libreria "unique_operations" e che determina la costruzione della sequenza per la transazioni. Questo approccio consente al fuzzer di rilevare bug che altrimenti potrebbero passare inosservati. Ad esempio si, può verificarsi il caso in cui una funzione debba essere chiamata più volte consecutive senza essere interrotta.

La riga del codice in cui viene definita la variabile "min_size_sampled" richiede una precisazione. La sua definizione è stata scelta in modo specifico per favorire valori interi vicini a zero, agevolando il processo di restringimento. Tuttavia, allo scopo di migliorare l'esplorazione dello spazio di ricerca, è preferibile poter campionare da tutte le funzioni disponibili per costruire una possibile sequenza di transazioni. Ecco perché non è stata definita direttamente come

```
min\_size\_sampled = draw(st.integers(0,len(operations)-1))
```

ma come

```
min\_seq\_len_sampled = seq\_len-draw(st.integers(0,seq\_len-1))
```

Se si modificasse questa riga Hypothesis non sarebbe quasi mai in grado di campionare tutte le sequenze disponibili in un contratto composto da molte funzioni. In linea con questa logica, è anche vantaggioso mantenere il valore predefinito di "favor_long_sequence" impostato su "true", poiché incoraggia la generazione di lunghe sequenze di transazioni per l'esplorazione di possibili bug più in profondità.

Facciamo un esempio dimostrativo eseguiamo "python3 lysmata.py -r 1000 -s 100 -c tests/-swarm_test.sol" che verifica il seguente contratto:

```
contract InvariantBreaker {
    uint counter = 0;
    bool public flag1 = true;

    function reset(int256 val) public {
        counter = 0;
    }

    function set(int256 val) public returns(bool) {
        counter = counter + 1;
        if (counter == 50) {
```

```
            flag1 = false;
        }
        return flag1;
    }
}

contract InvariantTest {
    InvariantBreaker public inv;

    function setUp() public {
        inv = new InvariantBreaker();
    }

    function invariant_neverFalse() public returns (bool)      {
        return inv.flag1();
    }
}
```

Il tool riuscirà a rompere l'invariante producendo il seguente risultato :

```
Esempio di falsificazione: composite_test(
    ops=[( <Function set>,0),
     ( <Function  set> , 0),
     ( <Function  set> , 0),
     ( <Function  set> , 0),
     (<Function set>,0),
        ...
        ...
        ...
        ...
     (< Set di funzioni  > , 0),
     ( < Set di funzioni  >, 0),
     ( < Set di funzioni  >, 0),
     ( < Set  di funzioni >, 0),
     ( < Set di funzioni  >, 0),
     ( < Set di funzioni  >, 0),
     ( < Set di funzioni  >, 0),
     ( < Set di funzioni  >, 0),
     ( < Set di funzioni > set > , 0)],
)
Invariante rotta !!!
```

Questo esempio dimostra quanto detto, perchè se si modifica il file config.yaml per impostare "swarm_testing" su false, il tool non sarà quasi mai in grado di interrompere l'invariante in "solo" 1000 sequenze di transazioni. Questo perché il fuzzer sceglie casualmente quale funzione chiamare in una sequenza di transazioni, ed è estremamente improbabile scegliere solo la funzione set() per 50 volte di seguito senza mai chiamare reset() per caso.

Constants mining

In linea con l'approccio del fuzzer Echidna, è stato integrato l'utilizzo dell'analisi statica per estrarre delle costanti codificate all'interno dei contratti in esame, in modo da indirizzare il fuzzer a campionare tra quelle costanti. In effetti, è comune pratica sbagliata di programmazione che vengano codificate delle costanti utilizzate nelle uguaglianze all'interno delle istruzioni if. L'utilizzo di questa scansione permette di intraprendere certi percorsi più facilmente. Ad esempio un fuzzer difficilmente raggiungerebbe un ramo del genere: if (x == 43685820975004) se il valore di x venisse generato in modo casuale in solo 1000 tentativi.
E' stato adottato "Slither", presentato nelle sezioni precedenti, per estrarre le costanti codificate direttamente dal codice sorgente. Queste costanti vengono successivamente incorporate nelle strategie di campionamento delle variabili di Hypothesis. È possibile testare questa funzionalità eseguendo: "python3 lysmata.py -r 1000 -s 100 -c tests/constants_test.sol"

che testerà il seguente contratto:

```
contract InvariantBreaker {
    bool public flag0 = true;

    function set0(string memory val) public returns (bool) {
        if (keccak256(abi.encodePacked(val)) ==
                    keccak256("StringaMoltoSignificativa"))
            flag0 = false;
        return flag0;
    }
}

contract InvariantTest {
    InvariantBreaker public inv;

    function setUp() public {
        inv = new InvariantBreaker();
    }

    function invariant_neverFalse() public returns(bool)      {
        return inv.flag0();
    }
}
```

Come puoi vedere, il fuzzer deve essere in grado di generare una stringa molto specifica "StringaMoltoSignificativa" per poter rompere l'invariante. Questo può essere facilmente verificabile modificando su false l'impostazione "Constants mining" dal file Yaml. Cos`ı facendo il fuzzer difficilmente troverà il percorso per rompere l'invariante.

Coverage-guided fuzzing

Per implementare la guida alla copertura, ho monitorato ogni transazione utilizzando le

chiamate "debug_traceTransaction" al nodo di test e ho estratto l'elenco dei program counter raggiunti durante ciascuna transazione. Questa attività è stata accettata dai test in quanto aggiunge solo un sovraccarico temporale del 10% durante l'esecuzione del codice. Quando viene raggiunto per la prima volta un nuovo program counter durante una sequenza di transazioni, si assegna una ricompensa molto elevata a tale sequenza per incoraggiare il fuzzer a esplorare percorsi simili. Questo processo avviene mediante il trasferimento della ricompensa alla funzione target dalla libreria delle hypothesis, che abilita test mirati basati sulle proprietà.

L'opzione "cover_guidance" è attivata per impostazione predefinita nel Yaml ed è di solito utile quando è necessaria un'esplorazione approfondita.

6.9 Conclusioni

In questo lavoro ho proposto il mio property-based tester scritto in Python, un framework di fuzzing testato per rilevare 5 tipi di vulnerabilità sugli smart contract utilizzando contratti vulnerabili selezionati dalla repository GitHub [25]. Il nostro esperimento è stato effettuato con contratti estratti da challenge, audit e da esempi del mondo reale tra cui, DAO e il Parity Wallet bug, ha mostrato che la strategia di generazione di input tramite la definizione di invarianti sia efficace per rilevare vulnerabilità con una buona precisione e a un livello pari a quanto evidenziato dallo studio eseguito Trail of Bits[29]. Il confronto con Echidna[28], che implementa la medesima tecnica, ha mostrato un tasso di precisione simile nella ricerca di vulnerabilità nelle classi esaminate, ma con una significativa lentezza nel confronto e una peggiore user-experience[28]. Per quanto riguarda possibili futuri sviluppi del tool, può essere interessante integrare una maggiore versatilità nelle reti. Infatti ora il tool ne accetta solamente una. Inoltre, una maggiore integrazione con i framework di sviluppo tipo Truffle o Brownie aumenterebbe la possibilità di copertura nei progetti. Inoltre è fondamentale un processo di ottimizzazione del processo di esecuzione in modo da migliorare le prestazioni del fuzzer. Potrebbe rivelarsi inoltre di grande utilità una migliore implementazione della libreria Hypothesis integrando un dataset di input noti per permettere al fuzzer di generare nuovi input da essi, anziché iniziare da zero per ogni generazione. Questo permetterebbe la creazione di input più complessi in un numero minore di tentativi.

Nell'utilizzo all'interno dell'azienda si è dedotto che la pessima user-experience data dallo strumento e da attribuire anche alla lentezza dello strumento e dalla poca interazione con il terminale Infatti come si deduce dal paper "Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts"[28], gli strumenti facili da eseguire e che producono risultati rapidi sono più probabilmente integrati dagli ingegneri durante il loro processo di sviluppo. La velocità dunque rende anche più pratico l'uso di uno strumento e la sua integrazione continua.
Queste problematiche attuali possono essere compensate dalla versatilità e implementabilità di Python con future estensioni. Concludo dicendo che questo elaborato e questo progetto mi sono serviti per comprendere meglio le tecnologie del Web3 e l'auditing di smart contract con l'impiego di strumenti di analisi automatica.

Appendice A

Glossario

Questo glossario contiene e spiega molti dei termini utilizzati in relazione a Ethereum, alla blockchain e alle criptovalute.

Account
Un oggetto contenente un indirizzo, un saldo, un nonce. Un account può essere un account di un contratto o un account di proprietà esterna (EOA).

Blocco
Una collezione di informazioni,che vengono richieste nell'intestazione, riguardo alle transazioni contenute in esso, e un insieme di altre intestazioni note come "ommers". I blocchi vengono aggiunti alla rete Ethereum dai minar.

Blocco Genesi
Il primo blocco in una blockchain, utilizzato per inizializzare una particolare rete e la sua relativa criptovaluta.

Blockchain
In Ethereum, una sequenza di blocchi convalidati dal sistema di Proof-of-work, ciascuno collegato al suo predecessore fino al blocco genesi. Questo varia dal protocollo Bitcoin non avendo un limite di dimensione nel blocco,ma limiti di gas variabili.

Bytecode
Un set di istruzioni astratto, progettato per un'esecuzione efficiente da parte di un interprete software o di una macchina virtuale. A differenza del codice sorgente leggibile dall'uomo, il bytecode è espresso in formato numerico.

Chiave privata
Vedi "chiave segreta"

Chiave pubblica
Un numero derivato da una chiave privata tramite una funzione unidirezionale, non reversibile, che può essere condiviso pubblicamente e utilizzato da chiunque per verificare una firma digitale fatta con la corrispondente chiave privata.

Chiave segreta (ovvero chiave privata)
Il numero segreto che consente agli utenti di Ethereum di dimostrare la titolarità di un account o di un contratto, producendo una firma digitale (vedi "chiave pubblica", "indirizzo", "ECDSA").

Compilazione

Conversione del codice scritto in un linguaggio di programmazione di alto livello (ad es. Solidity) in un linguaggio di livello inferiore (ad es. bytecode EVM).

Consenso
Quando numerosi nodi, di solito la maggior parte di quelli presenti sulla rete, hanno gli stessi blocchi nella loro blockchain locale validata riconosciuta come la migliore. Da non confondere con le regole di consenso.

Conto contratto
Un conto contenente un codice che viene eseguito ogni volta che riceve una transazione da un altro conto (EOA o contratto).

DAO
Organizzazione Autonoma Decentralizzata. Una società o altra organizzazione che opera senza una gestione gerarchica. Potrebbe anche fare riferimento a un contratto denominato "The DAO" lanciato il 30 aprile 2016, che è stato poi violato nel giugno 2016. Questo alla fine ha attivato un hard fork, nome in codice DAO, al blocco n. 1.192.000, che ha annullato il contratto DAO e causato la divisione di Ethereum ed Ethereum Classic in due sistemi concorrenti.

DApp
Applicazione decentralizzata. E' un contratto intelligente con un'interfaccia utente web. Più in generale, una DApp è un'applicazione Web costruita su servizi di infrastruttura peer-to-peer aperti, decentralizzati.

Deed
Standard di token non fungibili (NFT), introdotto dalla proposta ERC721. A differenza dei token ERC20, gli NFT dimostrano la proprietà e non sono intercambiabili, sebbene non siano riconosciuti come documenti legali in nessuna giurisdizione, almeno non attualmente (vedi anche "NFT").

Difficoltà
Un'impostazione a livello di rete che controlla la quantità di calcolo necessaria per produrre un proof-of-work.

ECDSA
Algoritmo di firma digitale a curva ellittica. Un algoritmo crittografico utilizzato da Ethereum per garantire che i fondi possano essere spesi solo dai loro proprietari.

Entropia
Nel contesto della crittografia, ci si riferisce alla mancanza di prevedibilità o livello di casualità. Quando si generano informazioni segrete, come le chiavi private, gli algoritmi di solito si affidano a una fonte di elevata entropia per garantire che l'output sia imprevedibile.

EOA
Account di proprietà esterna. Un account creato da persone che compongono la rete Ethereum.

Ether
La criptovaluta nativa utilizzata dall'ecosistema Ethereum, che copre i costi del gas durante l'esecuzione di contratti intelligenti.

Event
Consente l'uso delle funzionalità di registrazione EVM. Le DApp possono ascoltare gli eventi e utilizzarli per attivare i callback JavaScript nell'interfaccia utente.

EVM
Macchina virtuale Ethereum. Una macchina virtuale basata su stack che esegue bytecode. In Ethereum, il modello di esecuzione specifica come viene alterato lo stato del sistema dato, con una serie di istruzioni bytecode e una piccola tupla di dati. Questo è specificato attraverso un modello formale di una macchina a stati virtuale.

Fallback function
Una funzione predefinita chiamata in assenza di dati o un nome di funzione dichiarato.

Faucet
Un servizio che eroga fondi sotto forma di test ether gratuito che può essere utilizzato su una testnet.

Firma digitale
Una breve stringa di dati che un utente produce per un documento, utilizzando una chiave privata in modo tale che chiunque abbia la corrispondente chiave pubblica, possa verificare che il documento è stato "firmato" dal proprietario di quella chiave privata e che non sia stato modificato dopo la firma.

Fork
Un cambiamento nel protocollo che causa la creazione di una catena alternativa o una divergenza temporale in due potenziali percorsi di blocco durante il mining.

Ganache
Una blockchain Ethereum personale che può essere utilizzata per eseguire test, comandi e ispezionare lo stato controllando il funzionamento della catena.

Gas
Un carburante virtuale utilizzato in Ethereum per eseguire contratti intelligenti. L'EVM utilizza un meccanismo contabile per misurare il consumo di gas e limitare il consumo di risorse di calcolo (vedi "Turing completo").

Hash
Un'impronta digitale di lunghezza fissa di input di dimensione variabile, prodotta da una funzione hash.

IDE
Ambiente di sviluppo integrato. Un'interfaccia utente che in genere combina un editor di codice, un compilatore, un runtime e un debugger.

Immutabile
Una volta distribuito il codice di un contratto, o di una libreria, diventa immutabile. Le pratiche di sviluppo software standard si basano sulla possibilità di correggere eventuali bug e

di aggiungere nuove funzionalità. Questo rappresenta una sfida per lo sviluppo di contratti intelligenti.

Indirizzo
Più in generale, questo rappresenta un EOA o un contratto che può ricevere, indirizzo di destinazione, o inviare, indirizzo di origine, transazioni sulla blockchain. Più tecnicamente, sono i 160 bit più a destra di un hash Keccak di una chiave pubblica ECDSA.

IPFS
File system interplanetario: un protocollo, una rete e un progetto open source progettato per creare un metodo peer-to-peer indirizzabile al contenuto per l'archiviazione e la condivisione di ipermedia in un file system distribuito.

Keccak-256
Funzione hash crittografica utilizzata in Ethereum. Keccak-256 è stato standardizzato come SHA-3. La funzione di hash Keccak prende un input di lunghezza variabile e produce un output di lunghezza fissa (256 bit per Keccak-256).

Keystore file
Un file con codifica JSON che contiene una singola chiave privata, generata in modo casuale, crittografata da una passphrase per una maggiore sicurezza.

LevelDB
Un archivio chiave-valore su disco open source, implementato come una libreria leggera e monouso, con collegamenti a molte piattaforme.

Library
Un tipo speciale di contratto che non ha funzioni a pagamento, nessuna funzione di fallback e nessuna memorizzazione dei dati. Pertanto non ha la possibilità di può ricevere, o detenere ether o archiviare dati.

Lightweight client
Un client Ethereum che non memorizza una copia locale della blockchain o convalida blocchi e transazioni. Offre le funzioni di un portafoglio e può creare e trasmettere transazioni.

Limite gas
La quantità massima di gas che una transazione o un blocco può consumare. Linguaggio assembly EVM Una forma leggibile dall'uomo di bytecode EVM.

Merkle Patricia Tree
Una struttura dati utilizzata in Ethereum per archiviare in modo efficiente coppie chiave-valore.

Messaggio
Una transazione interna che non viene mai serializzata e inviata solo all'interno dell'EVM.

Message call
L'atto di passare un messaggio da un account a un altro. Se l'account di destinazione è associato al codice EVM, la VM verrà avviata con lo stato di quell'oggetto e il messaggio sul quale si agirà.

Miner
Un nodo di rete che trova proof-of-work valide per nuovi blocchi, mediante hashing ripetuto.

NFT
Un token non fungibile, noto anche come ”atto”. Questo è uno standard token introdotto dalla proposta ERC721. Gli NFT possono essere monitorati e scambiati, ma ogni token è unico e distinto. Non sono intercambiabili come i token ERC20. Gli NFT possono rappresentare la proprietà di risorse digitali o fisiche.

Nodo
Un client software che partecipa alla rete.

Nonce
In crittografia, un valore che può essere utilizzato solo una volta. Esistono due tipi di nonce utilizzati in Ethereum: un account nonce è un contatore di transazioni in ciascun account, che viene utilizzato per prevenire attacchi di riproduzione; un nonce proof-of-work è il valore casuale in un blocco utilizzato per soddisfare la prova del lavoro.

Ommer
“Un blocco figlio di un antenato che non ha un padre”. Quando un minatore trova un blocco valido, un altro minatore potrebbe aver pubblicato prima di lui un blocco concorrente che viene aggiunto in cima alla blockchain. A differenza di Bitcoin, i blocchi “orfani” in Ethereum possono essere inclusi da più recenti come ommer e ricevere una ricompensa parziale per blocco.

Proof of stake (PoS)
Un metodo con cui un protocollo blockchain di criptovaluta mira a ottenere un consenso distribuito. PoS chiede agli utenti di dimostrare la proprietà di una certa quantità di criptovaluta, la loro ”quota” nella rete, per poter partecipare alla convalida delle transazioni.

Proof of work (PoW)
Un pezzo di dati, la prova, che richiede un calcolo significativo per essere trovato. In Ethereum, i minatori devono trovare una soluzione numerica all’algoritmo Ethash che soddisfi un obiettivo di difficoltà a livello di rete.

Python
Un linguaggio di programmazione smart contract procedurale (imperativo) con sintassi simile a Python.

Receipt
Dati restituiti da un client Ethereum per rappresentare il risultato di una particolare transazione, incluso l’hash della transazione, il suo numero di blocco, la quantità di gas utilizzata e, in caso di implementazione di uno smart contract, l’indirizzo del contratto.

Re-entrancy attack
Un attacco che consiste in un contratto dell’attaccante che richiama una funzione del contratto della vittima in modo tale che, durante l’esecuzione, la vittima chiami nuovamente il contratto dell’attaccante in modo ricorsivo. Ciò può comportare, ad esempio, il furto di fondi saltando parti del contratto della vittima che aggiornano i saldi o contano gli importi prelevati.

Regole di consenso
Le regole di convalida del blocco che i nodi seguono per rimanere in consenso con gli altri nodi. Da non confondere con il consenso.

Ricompensa

Una quantità di ether inclusa in ogni nuovo blocco come ricompensa dalla rete al minatore che ha trovato la soluzione proof-of-work.

Rete Ethereum
Una rete peer-to-peer che propaga transazioni e blocchi a ogni nodo Ethereum partecipante alla rete.

Satoshi Nakamoto
Il nome utilizzato dalla persona o dalle persone che hanno progettato Bitcoin, creando la sua implementazione di riferimento originale e sono state le prime a risolvere il problema della doppia spesa per la valuta digitale. La loro vera identità rimane sconosciuta.

SHA
Secure Hash Algorithm. Una famiglia di funzioni hash crittografiche pubblicate dal National Institute of Standards and Technology (NIST).

Singleton
Un termine utilizzato nella computer science per descrivere un oggetto di cui può esistere solo una singola istanza.

Smart contract
Un programma che viene eseguito sull'infrastruttura informatica di Ethereum.

Solidity
Un linguaggio di programmazione procedurale (imperativo) con sintassi simile a JavaScript, C++ o Java. Il linguaggio più popolare e più utilizzato per gli smart contract Ethereum. Creato dal Dr. Gavin Wood.

Swarm
Una rete di archiviazione decentralizzata (P2P), utilizzata insieme a Web3 e Whisper per creare DApp.

Testnet
Abbreviazione di "test network", è una rete utilizzata per simulare il comportamento della rete Ethereum principale.

Transazione
Dati impegnati nella blockchain di Ethereum firmati da un account di origine, indirizzati a un indirizzo specifico. La transazione contiene metadati come il limite del gas per quella transazione.

Transazione di creazione del contratto
Una transazione speciale, con l'indirizzo zero come destinatario, che serve per registrare un contratto e registrarlo sulla blockchain di Ethereum, vedi anche "indirizzo zero".

Transazione interna (anche "messaggio")
Una transazione inviata da un conto contratto a un altro conto contratto o a un EOA.

Turing completo
Un concetto che prende il nome dal matematico e informatico inglese Alan Turing. È un sistema di regole di manipolazione dei dati, come il set di istruzioni di un computer, un linguaggio di programmazione o un automa a stati. È detto "Turing completo" o "computazionalmente

universale" se può essere utilizzato per simulare qualsiasi macchina di Turing.

Viper
Un linguaggio di programmazione di alto livello, simile a Serpent, con sintassi simile a Python. Destinato ad avvicinarsi a un linguaggio puramente funzionale. Creato da Vitalik Buterin.

Wallet
Software che detiene chiavi segrete. Utilizzato per accedere e controllare gli account Ethereum e interagire con i contratti intelligenti. Le chiavi non devono essere archiviate in un wallet e possono invece essere recuperate dall'archivio offline, ad esempio un foglio di carta, per una maggiore sicurezza.

Web3
La terza versione del web. Proposto per la prima volta dal Dr. Gavin Wood, Web3 rappresenta una nuova visione e attenzione per le applicazioni web: dalle applicazioni gestite e possedute centralmente, alle applicazioni costruite su protocolli decentralizzati.

Wei
La più piccola denominazione di etere. 10^{18} wei = 1 etere.

Whisper
Un servizio di messaggistica decentralizzato (P2P). Viene utilizzato insieme a Web3 e Swarm per creare DApp.

Zero address
Uno speciale indirizzo Ethereum, composto interamente da zeri, specificato come indirizzo di destinazione di una transazione per la creazione di un contratto.

"Program testing can be used to show the presence of bugs,

but never to show their absence!"

– Edsger Dijkstra

Bibliografia

[1] Andreas M Antonopoulos e Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[2] Tanya Bahrynovska. *History of Ethereum Security Vulnerabilities, Hacks, and Their Fixes*. 2017. URL: https://applicature.com/blog/blockchain-technology/history-of-ethereum-security-vulnerabilities-hacks-and-their-fixes.

[3] Eric Banisadr. *How 800kEvaporatedf romthePoWHCoinPonziSchemeOvernight*. 2018. URL: https://medium.com/@ebanisadr/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530.

[4] Ivan Botayy. *Implementing Ethereum trading front-runs on the Bancor exchange in Python*. 2017. URL: https://hackernoon.com/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798.

[5] Vitalik Buterin. *The Meaning of Decentralization*. 2017. URL: https://medium.com/@VitalikButerin/the-meaning-of-decentralization-a0c92b76a274.

[6] Vitalik Buterin et al. "A next-generation smart contract and decentralized application platform". In: *white paper* 3.37 (2014), pp. 2–1.

[7] Chainalysis. *The 2023 Crypto Crime Report*. 2023. URL: https://go.chainalysis.com/2023-crypto-crime-report.html.

[8] Brian Curran. *What is Nakamoto Consensus? Complete Beginner's Guide*. 2018. URL: https://blockonomi.com/nakamoto-consensus/.

[9] Wei Dai. "b-money". In: *URL http://www. weidai. com/bmoney. txt* (1998).

[10] Phil Daian. *Analysis of the DAO exploit*. 2016. URL: https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[11] Chris Dannen. *Introducing Ethereum and solidity*. Vol. 1. Springer, 2017.

[12] Cynthia Dwork e Moni Naor. "Pricing via processing or combatting junk mail". In: *Annual international cryptology conference*. Springer. 1992, pp. 139–147.

[13] ethereum.org. *Introduction to Web3*. 2023. URL: https://ethereum.org/en/web3/.

[14] Etherik. *GovernMental's 1100 ETH jackpot payout is stuck because it uses too much gas*. 2016. URL: https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/.

[15] etherscan.io. URL: https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d#code.

[16] etherscan.io. URL: https://etherscan.io/tx/0xad89ff16fd1ebe3a0a7cf4ed282302c06626c1af33221ebe0d3

[17] etherscan.io. URL: https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4#code.

[18] etherscan.io. URL: https://etherscan.io/tx/0x9dbf0326a03a2a3719c27be4fa69aacc9857fd231a8d9dcaede

[19] etherscan.io. URL: https://etherscan.io/tx/0xeef10fc5170f669b86c4cd0444882a96087221325f8bf2f55d6

[20] Josselin Feist, Gustavo Grieco e Alex Groce. "Slither: a static analysis framework for smart contracts". In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE. 2019, pp. 8–15.

[21] github.com. *Anvil*. 2023. URL: https://github.com/foundry-rs/foundry/tree/master/crates/anvil.

[22] github.com. *Hypothesis readdocs*. 2023. URL: https://hypothesis.readthedocs.io/en/latest/index.html.

[23] github.com. *not-so-smart-contracts*. 2023. URL: https://github.com/crytic/not-so-smart-contracts.

[24] github.com. *openzeppelin-contracts-SafeMath.sol*. 2023. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol.

[25] github.com. *Security Reviews*. 2023. URL: https://github.com/trailofbits/publications#security-reviews.

[26] github.com. *Slither Detectors LIst*. 2023. URL: https://github.com/crytic/slither#detectors.

[27] github.com. *The DAO*. 2023. URL: https://github.com/blockchainsllc/DAO/blob/v1.0/README.md.

[28] Gustavo Grieco et al. "Echidna: effective, usable, and fast fuzzing for smart contracts". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020, pp. 557–560.

[29] Alex Groce et al. "What are the actual flaws in important smart contracts (and how can we find them)?" In: *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer. 2020, pp. 634–653.

[30] Stuart Haber e W Scott Stornetta. *How to time-stamp a digital document*. Springer, 1991.

[31] Daniel Idowu. *Learn Solidity: What is the Solidity compiler?* 2022. URL: https://www.alchemy.com/overviews/solidity-compiler.

[32] Seoung Kyun Kim et al. "Measuring ethereum network peers". In: *Proceedings of the Internet Measurement Conference 2018*. 2018, pp. 91–104.

[33] Leslie Lamport. "The weak Byzantine generals problem". In: *Journal of the ACM (JACM)* 30.3 (1983), pp. 668–676.

[34] Matthew Leising. *OUT OF THE ETHER*. Wiley, 2021.

[35] Ralph C Merkle. "A certified digital signature". In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 218–238.

[36] tintinweb microsoft. *Solidity Visual Developer*. 2023. URL: https://marketplace.visualstudio.com/items?itemName=tintinweb.solidity-visual-auditor&ssr=false#overview.

[37] Mittelstufe. *What Is A Sybil Attack in Blockchain?* 2022. URL: https://www.coincarp.com/de/learn/what-is-a-sybil-attack-in-blockchain/.

[38] Ujan Mukhopadhyay et al. “A brief survey of cryptocurrency systems”. In: *2016 14th annual conference on privacy, security and trust (PST)*. IEEE. 2016, pp. 745–752.

[39] Satoshi Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: *Decentralized business review* (2008), p. 21260.

[40] C Namecoin. *Namecoin*. 2010.

[41] Nathaniel Popper. *DIGITAL GOLD*. Harper, 2015.

[42] Haseeb Qureshi. *A hacker stole* 31*Mof Ether howithappened, andwhatitmeansforEthereum*. 2017. URL: https://www.freecodecamp.org/news/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce.

[43] Arseniy Reutov. *Predicting Random Numbers in Ethereum Smart Contracts*. 2018. URL: https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620.

[44] Meni Rosenfeld et al. “Overview of colored coins”. In: *White paper, bitcoil. co. il* 41 (2012), p. 94.

[45] Nick Szabo. “Smart contracts: building blocks for digital markets”. In: *EXTROPY: The Journal of Transhumanist Thought,(16)* 18.2 (1996), p. 28.

[46] Alchemy Team. *Web3 Development Report (Q4 2022)*. 2022. URL: https://www.alchemy.com/blog/web3-developer-report-q4-2022.

[47] TradingView. *Crypto Total Market Cap*. 2023. URL: https://it.tradingview.com/symbols/TOTAL/.

[48] Currency Tycoon. *Tricked by a honeypot contract or beaten by another hacker. What happened?* 2017. URL: https://www.reddit.com/r/ethdev/comments/7x5rwr/tricked_by_a_honeypot_contract_or_beaten_by/.

[49] Peter Vessenes. *Tweet di Peter Vessenes*. 2016. URL: https://twitter.com/vessenes/status/733030888028528640.

[50] Brady Werkheiser. *Ethereum Statistics*. 2019. URL: https://www.alchemy.com/overviews/ethereum-statistics.

[51] J. R. Willet. *MasterCoin Complete Specification*. 2012.

[52] Andreas M. Antonopoulos Dr. Gavin Wood. *ethereumbook*. 2019. URL: https://cypherpunks-core.github.io/ethereumbook/.

[53] Gavin Wood et al. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

Progetto sviluppato presso la Shielder S.r.l.

https://www.shielder.com/