UNIVERSITÀ
DI PARMA

Dipartimento di Scienze Matematiche, Fisiche e Informatiche

Corso di Laurea Triennale in Informatica

Costruzione di Control-Flow Graph completi per bytecode EVM

Candidato:
Saverio Mattia Merenda

Relatore:
Prof. Vincenzo Arceri

Matricola:
330503

Anno Accademico 2023–2024

0% chiacchiere, 200% lavoro.

A me medesimo.

Indice

Elenco delle figure

Elenco degli algoritmi

Elenco delle tabelle

Capitolo 1

Introduzione

Ethereum è una piattaforma blockchain che consente la creazione di un ambiente globale decentralizzato, in cui gli utenti possono sviluppare strumenti digitali sicuri utilizzando la valuta nativa *ether* (ETH) per transazioni e servizi computazionali [3].

La rete Ethereum è costituita da nodi interconnessi che seguono regole definite nel protocollo Ethereum, supportando una vasta gamma di comunità, applicazioni, organizzazioni e attività digitali accessibili a chiunque abbia una connessione internet.

La caratteristica principale di Ethereum è la capacità di eseguire smart contract tramite la Ethereum Virtual Machine (EVM), consentendo funzionalità avanzate al di là delle transazioni finanziarie di base (e.g., Bitcoin [4]). Gli smart contract sono programmi immutabili memorizzati sulla blockchain che eseguono azioni predefinite senza l'intervento umano.

Tuttavia, garantire la sicurezza e l'affidabilità degli smart contract è essenziale, poiché non possono essere modificati una volta caricati sulla blockchain. La presenza di bug o vulnerabilità potrebbe causare gravi conseguenze, come perdite di fondi o azioni indesiderate, proprio come è accaduto nel 2016 con l'attacco a The DAO.

Per garantire la qualità degli smart contract, vengono sviluppate tecniche di analisi e verifica del codice. L'analisi statica, ad esempio, identifica potenziali problemi di sicurezza senza eseguire effettivamente il codice. Tuttavia, nel contesto di Ethereum, la costruzione di un Control-Flow Graph (CFG) affidabile è complesso a causa delle destinazioni dinamiche dei salti (Jump Orfane) durante l'esecuzione del EVM bytecode.

In questo elaborato viene presentato EVMLiSA[1], un software in grado di costruire un CFG completo e affidabile di smart contract eseguibili su EVM.

[1] `https://github.com/lisa-analyzer/evm-lisa`

Capitolo 2

Background

Nel seguente capitolo, esploreremo il complesso mondo della tecnologia blockchain, ripercorrendo la sua storia e comprendendo il suo funzionamento. Successivamente, esamineremo l'ecosistema di Ethereum, analizzando il suo funzionamento e lo confronteremo con altre piattaforme blockchain. Dedicheremo particolare attenzione alla Ethereum Virtual Machine (EVM) e ai vantaggi offerti, con un focus sugli smart contract e le loro implicazioni. Infine, concluderemo esaminando le criticità associate a questi ultimi.

2.1 Blockchain

Il concetto di blockchain è stato introdotto per la prima volta nel 2008 da Satoshi Nakamoto con il whitepaper di Bitcoin [4] e successivamente implementato nel 2009 come tecnologia alla base di Bitcoin.

È importante sottolineare che Bitcoin e blockchain sono due entità separate: Bitcoin rappresenta solo una delle molteplici applicazioni costruite sulla tecnologia blockchain.

La blockchain è un registro digitale, decentralizzato e distribuito su una rete, strutturato come una catena di blocchi responsabili dell'archiviazione dei dati. Questi dati possono comprendere informazioni di valore economico o intere applicazioni digitali. È possibile aggiungere nuovi blocchi di informazioni, ma non è possibile modificare o rimuovere i blocchi precedentemente aggiunti alla catena.

In questo ambiente, la crittografia e i protocolli di consenso garantiscono sicurezza e immutabilità. Il risultato è un sistema aperto, neutrale, affidabile e sicuro, dove la nostra capacità di utilizzare e fidarci del sistema non dipende dalle intenzioni di nessun individuo o istituzione.

La blockchain può essere considerata parte della famiglia delle *Distributed Ledger Technology* (DLT): tutti i partecipanti alla rete hanno accesso al regi-

stro distribuito e al suo registro immutabile delle transazioni. Questo registro condiviso consente di registrare le transazioni una sola volta, eliminando la duplicazione degli sforzi tipica delle reti aziendali tradizionali [5].

2.1.1 Funzionamento

La blockchain è composta da una serie di blocchi, aggiunti uno dopo l'altro in modo sequenziale. Ogni blocco contiene una prova matematica, resa possibile dall'utilizzo della crittografia, che ne assicura la sequenzialità tramite un hash del blocco precedente.

Sebbene il concetto di catena di blocchi sia comune a quasi tutti i sistemi blockchain, la struttura e il contenuto dei blocchi possono variare a seconda dello scopo della blockchain stessa. Ad esempio, i blocchi della blockchain di Bitcoin possono essere diversi da quelli della blockchain di Ethereum.

I blocchi sono interconnessi tramite funzioni di hash crittografiche (e.g., SHA-256[1]), creando un legame matematico tra di loro. Questo processo fornisce un modo conveniente per esprimere l'intero contenuto della blockchain in una singola stringa di lunghezza definita.

Ogni blocco contiene informazioni specifiche (e.g., le transazioni) e l'hash del blocco precedente (Figura 2.1). Modificare qualsiasi informazione in un blocco altererebbe l'hash del blocco e, di conseguenza, tutti gli hash successivi, rendendo immediatamente evidente l'alterazione.

Ogni nuovo blocco rafforza la verifica dei blocchi precedenti, rendendo la blockchain a prova di manomissione e fornendo un registro affidabile delle transazioni su cui gli utenti possono fare affidamento [5].

[1]Una funzione di hash, come ad esempio SHA-256, trasforma un input di lunghezza arbitraria in un output di lunghezza fissa, chiamato *hash*, attraverso un processo che coinvolge operazioni matematiche complesse e non invertibili. Il risultato è un valore univoco che rappresenta l'input originale e che è praticamente impossibile da invertire per determinare l'input originale da un dato hash.

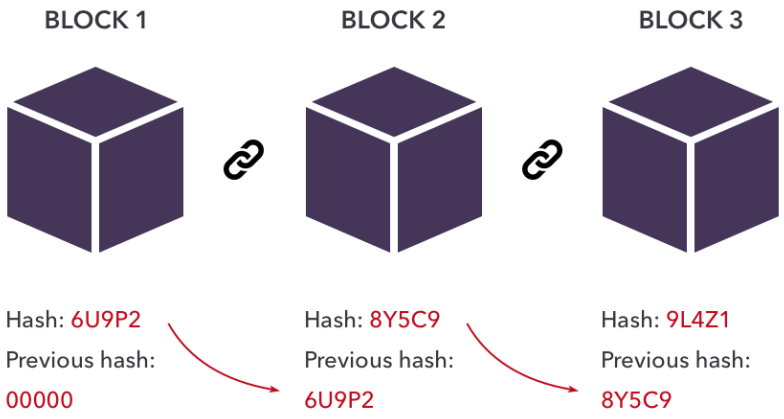


Figura 2.1: Funzionamento della blockchain.

2.1.2 Consenso

Il consenso è il processo mediante il quale i nodi della rete raggiungono un accordo su ciò che è avvenuto all'interno della blockchain. Questo accordo rappresenta l'unica verità sullo stato attuale della blockchain e viene ottenuto attraverso un accordo generale tra i partecipanti della rete. Il consenso non è un evento istantaneo, ma un processo continuo che coinvolge vari partecipanti con ruoli e responsabilità specifici. Si basa su principi matematici ed economici per incentivare tutti i membri a raggiungere un accordo su un unico dato [5].

Nella pratica, il consenso richiede che i nuovi blocchi siano approvati dalla maggioranza dei nodi della rete, i quali sono distribuiti su una vasta rete di computer sparsi in tutto il mondo. I due meccanismi di consenso più diffusi sono il *Proof-of-Work* (PoW) e il *Proof-of-Stake* (PoS).

- Proof-of-Work (PoW): è il sistema di consenso utilizzato principalmente da Bitcoin. Qui, i nodi della rete, noti come *miner*, competono per risolvere complessi problemi crittografici. Il primo nodo che risolve con successo il problema crittografico viene premiato con una quantità specifica di criptovaluta. La difficoltà di generare l'hash viene modificata man mano che la rete si espande, in modo che i nuovi blocchi vengano creati e approvati a un ritmo costante al variare della potenza di calcolo della rete. La difficoltà di generare blocchi nella blockchain di Bitcoin, ad esempio, viene regolata modificando il numero di zeri con cui devono iniziare, garantendo che un nuovo hash venga trovato solo una volta ogni dieci minuti circa dall'intera rete. Questi problemi sono progettati per richiedere una grande quantità di potenza di calcolo e l'energia utilizzata in questo processo è sia una caratteristica distintiva che un punto di vulnerabilità di PoW. Se da un lato rende la blockchain sicura, poiché

sarebbe estremamente costoso e poco pratico per un singolo attaccante possedere la maggior parte della potenza di calcolo della rete, dall'altro richiede una notevole quantità di energia elettrica [2].

- Proof-of-Stake (PoS): è un meccanismo di consenso in cui i nodi della rete, chiamati *validatori*, competono per essere selezionati per convalidare il prossimo blocco. La probabilità di essere selezionati è proporzionale alla quantità di criptovaluta che il nodo ha depositato nella blockchain. Quando un validatore viene selezionato, "mette in gioco" (stake) i propri fondi (token). Se il validatore non adempie ai propri doveri o tenta di commettere una frode, rischia di perdere i suoi fondi. Il consenso PoS non richiede una quantità significativa di energia elettrica come PoW, ma presenta un problema di monopolio della ricchezza, poiché coloro che possiedono più token hanno maggiori probabilità di essere selezionati per convalidare il blocco successivo e ricevere la ricompensa. Questo problema è mitigato con l'introduzione di *Delegated Proof-of-Stake* (DPoS), che prevede un sistema di voto per la selezione dei validatori, rendendo così il processo più democratico [2].

2.1.3 Classificazione delle blockchain

Le blockchain possono essere categorizzate in vari tipi in base alla loro accessibilità e ai requisiti di utilizzo:

- Permissionless: questo tipo di blockchain permette a chiunque di accedervi senza restrizioni, senza la necessità di identificarsi. Esempi notevoli sono Bitcoin ed Ethereum. Tuttavia, questa libertà può portare a problemi di dimensionamento e sicurezza, poiché ogni nodo della rete deve elaborare e convalidare l'intera blockchain, rendendo anche più probabile l'ingresso di attori malevoli [6].

- Permissioned: al contrario, nelle blockchain ad accesso limitato, l'accesso è controllato e richiede l'autenticazione da parte di un'autorità centrale. Questo approccio riduce i problemi di dimensionamento e sicurezza riscontrati nelle blockchain aperte. Tuttavia, l'intervento di un'autorità centrale compromette parzialmente il principio di decentralizzazione [6].

Ulteriormente, le blockchain possono essere suddivise in base alle loro applicazioni e requisiti specifici:

- Public: le blockchain pubbliche incoraggiano la partecipazione di tutti gli utenti, offrendo incentivi come ricompense in criptovaluta. Sono trasparenti e flessibili. Esempi ben noti includono Bitcoin ed Ethereum [7].

- Private: le blockchain private sono utilizzate all'interno di reti aziendali o organizzative, spesso per migliorare l'efficienza dei processi interni. Sono meno trasparenti rispetto alle blockchain pubbliche e richiedono solitamente un'autorizzazione per accedervi [7].

- Consortium: queste blockchain sono progettate per la collaborazione tra più entità, come aziende o istituzioni. La gestione dell'accesso è condivisa tra i partecipanti, garantendo un certo livello di decentralizzazione senza dipendenza da un'autorità centrale [7].

2.1.4 Chiave pubblica e chiave privata

La tecnologia blockchain si basa su uno dei principi fondamentali della sicurezza informatica: la *crittografia*. La crittografia è l'insieme di tecniche utilizzate per rendere un messaggio incomprensibile a chi non è autorizzato a leggerlo, garantendo così la confidenzialità dei dati.

Nel panorama della blockchain, si fa ampio uso della crittografia asimmetrica, conosciuta anche come *crittografia a chiave pubblica*. Questo tipo di crittografia coinvolge una coppia di chiavi per ciascun partecipante alla comunicazione:

- una chiave pubblica, accessibile a chiunque debba inviare informazioni;

- una chiave privata, custodita esclusivamente dal proprietario.

In un sistema basato sulla crittografia a chiave pubblica, qualsiasi persona può cifrare un messaggio utilizzando la chiave pubblica del destinatario, ma solo il destinatario può decifrarlo con la sua chiave privata. Questo meccanismo consente a due utenti di comunicare in modo sicuro anche su canali non sicuri, poiché la sicurezza dipende esclusivamente dalla custodia segreta della chiave privata [1].

Nel contesto della blockchain, la crittografia asimmetrica viene utilizzata per consentire lo scambio di asset digitali, come le criptovalute. Ogni partecipante alla blockchain possiede una coppia di chiavi, una pubblica e una privata.

Per comprendere meglio il funzionamento, consideriamo un esempio pratico con Alice e Bob:

- Entrambi sono identificati all'interno della blockchain tramite un indirizzo pubblico (chiave pubblica), noto come *address*.

- Chiunque desideri inviare loro degli asset deve farlo tramite il loro rispettivo indirizzo pubblico.

- Ognuno di loro possiede la propria chiave privata, necessaria per autorizzare le transazioni in uscita.
- Questa chiave privata è fondamentale per garantire che solo il legittimo proprietario possa effettuare transazioni.

Quando Alice vuole inviare 1 *bitcoin* a Bob, accede ai suoi fondi utilizzando la sua chiave privata e invia la transazione verso l'indirizzo di Bob. La transazione è crittografata con la chiave pubblica di Bob, garantendo la sicurezza del trasferimento. Solo Bob, utilizzando la sua chiave privata, sarà in grado di decifrare e confermare la transazione, garantendo così la legittimità dell'operazione. È fondamentale che entrambi custodiscano le proprie chiavi private in modo sicuro per evitare accessi non autorizzati [8].

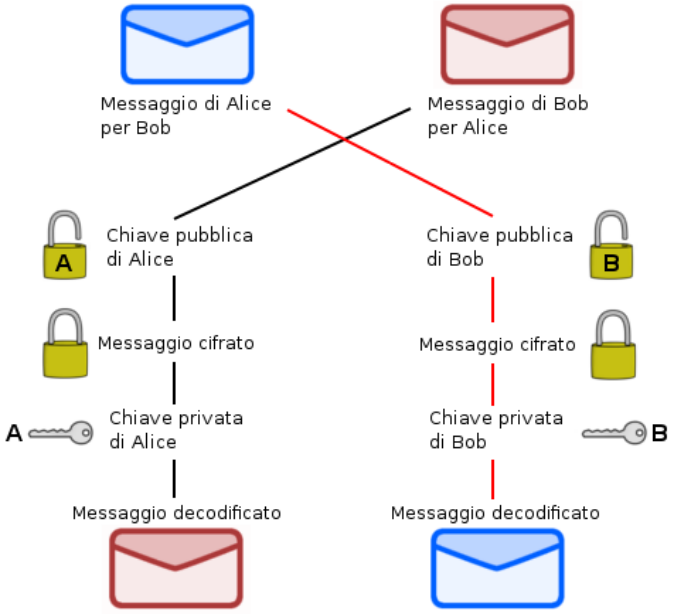


Figura 2.2: Esempio di comunicazione utilizzando un sistema di crittografia asimmetrica [1].

2.2 Ethereum

Ethereum è una blockchain aperta (*permissionless*), accessibile a chiunque (*pubblica*) e con il codice sorgente disponibile liberamente (*open-source*). È stata ideata per la prima volta da Vitalik Buterin nel 2013 [9], con l'obiettivo di creare una blockchain in grado di eseguire una vasta gamma di programmi generici.

Dal punto di vista tecnico, Ethereum può essere considerato una sorta di enorme macchina virtuale globale e *"infinita"*, che opera seguendo uno stato singolo accessibile da qualsiasi parte del mondo e una macchina virtuale che applica le modifiche a tale stato.

Tuttavia, in termini più pratici, Ethereum si presenta come un'infrastruttura informatica decentralizzata, aperta a tutti e basata su codice sorgente accessibile[2], che consente l'esecuzione di programmi denominati smart contract. Utilizza una blockchain per tenere traccia e registrare le variazioni di stato del sistema, utilizzando la criptovaluta nativa (chiamata *ether*) per misurare e regolare i costi delle risorse di elaborazione.

Attraverso la piattaforma Ethereum, gli sviluppatori hanno la possibilità di creare applicazioni decentralizzate con funzioni economiche integrate, offrendo non solo elevata disponibilità, verificabilità, trasparenza e neutralità, ma anche la riduzione o l'eliminazione della censura e alcuni rischi associati alle controparti tradizionali [10].

2.2.1 Etherem vs Bitcoin

Ethereum presenta numerose caratteristiche comuni con altre blockchain aperte: una rete peer-to-peer che collega gli utenti, un algoritmo di consenso per mantenere gli aggiornamenti di stato sincronizzati (inizialmente basato su un consenso Proof-of-Work, ma con l'avvento dell'aggiornamento *The Merge*[3], il consenso è passato a Proof-of-Stake), l'uso di tecniche crittografiche come firme digitali e hash, e una valuta digitale.

Tuttavia, sia lo scopo che la struttura di Ethereum si differenziano notevolmente da quelle delle blockchain antecedenti, come ad esempio Bitcoin.

Il principale obiettivo di Ethereum non è solo quello di fungere da sistema di pagamento digitale. Anche se l'*ether* è fondamentale per il funzionamento di Ethereum, essa viene considerata una *"valuta di utilità"* utilizzata per pagare l'utilizzo della piattaforma Ethereum come un computer globale.

A differenza di Bitcoin, che dispone di un linguaggio di scripting limitato, Ethereum è stato progettato per essere una blockchain programmabile per

[2] `https://github.com/ethereum/go-ethereum`
[3] `https://ethereum.org/it/roadmap/merge/`

scopi generici, con una macchina virtuale EVM in grado di eseguire codice di complessità arbitraria e illimitata. Mentre il linguaggio di scripting di Bitcoin si limita principalmente a valutazioni semplici (e,g,. operazioni booleane) delle condizioni di spesa di un utente, il linguaggio di Ethereum (Solidity[4]) è quasi completo in termini di capacità computazionale (Turing completeness), consentendo ad Ethereum di funzionare come un computer per scopi generali [10].

2.2.2 Funzionamento

Sulla blockchain Ethereum esistono due tipi principali di account utilizzabili per gestire le transazioni e l'interazione sulla rete: gli *Externally Owned Account* e i *Contract Account*.

Gli *Externally Owned Account* (EOA) sono controllati direttamente dagli utenti e sono associati ad una coppia di chiavi crittografiche, pubblica e privata. Questi account consentono agli individui di ricevere e inviare *ether* e di partecipare alle transazioni sulla rete. Le transazioni tra due account esterni coinvolgono esclusivamente lo scambio di *ether* e non comportano alcun costo di creazione dell'account [11].

I *Contract Account*, invece, sono associati agli smart contract. Questi account contengono il codice dello smart contract e vengono attivati quando ricevono una transazione. Le transazioni con questi account possono coinvolgere l'esecuzione di codice dello smart contract, oltre allo scambio di *ether* e comportano un costo di creazione in quanto richiedono risorse di calcolo e di archiviazione sulla rete Ethereum [11].

Affinché una transazione venga effettuata su Ethereum, il mittente deve conoscere l'indirizzo del destinatario, detto anche chiave pubblica dell'account, e firmare digitalmente la transazione con la propria chiave privata. Questo processo dimostra che il richiedente della transazione è il legittimo proprietario dell'account.

Le transazioni sono essenzialmente istruzioni crittograficamente firmate da un account che iniziano una modifica dello stato della rete Ethereum. Inoltre, per *transazione* si intende una transazione approvata e inclusa in un blocco della blockchain.

Poiché le transazioni sono atomiche, esse devono essere eseguite completamente prima di apportare cambiamenti allo stato globale della rete. Questo significa che tutte le istruzioni all'interno della transazione devono essere valide. Se una qualsiasi istruzione fallisce, gli effetti della transazione vengono annullati e lo stato viene ripristinato al momento precedente (rollback), come se la transazione non fosse mai avvenuta. Anche se una transazione fallisce,

[4] `https://docs.soliditylang.org/en/latest/`

viene comunque registrata come tentata, ma non influenza lo stato complessivo della rete [12].

Ogni transazione su Ethereum comporta un costo proporzionale alla sua complessità computazionale, misurato in *gas*. Possiamo pensare al gas come al carburante necessario per far funzionare le operazioni sulla blockchain, simile al carburante utilizzato da un'auto per percorrere una determinata distanza.

Ogni operazione sulla blockchain ha un costo specifico in gas. Ad esempio, calcolare un hash o fare una somma di due numeri richiede una differente quantità di gas (`30` e `3`, rispettivamente).

Il gas è misurato in *wei* (1 wei $= 1^{-18}$ ETH) ed è strettamente legato alle transazioni. Ogni transazione su Ethereum ha due parametri: il prezzo del gas (*gas price*) e il limite del gas (*gas limit*), che rappresentano rispettivamente il prezzo che si è disposti a pagare per unità di gas e la quantità massima di gas utilizzabile.

A differenza di Bitcoin, dove esiste un limite alla dimensione massima di un blocco, su Ethereum si fa riferimento al limite di gas, che determina la massima quantità di calcoli che la Ethereum Virtual Machine deve eseguire per blocco.

Immaginiamo di dover eseguire una transazione che richiede 10 gas e abbiamo deciso di pagare `100 wei` per ogni gas: il costo totale della transazione sarà di `1000 wei` (10×100). Se aumentiamo il prezzo del gas a `1000 wei` per gas, il costo totale della transazione sarà di `10000 wei` (10×1000). I validatori della rete Ethereum sono liberi di scegliere le transazioni da includere nel nuovo blocco, e generalmente cercano di massimizzare il profitto. Di conseguenza, le transazioni con un prezzo del gas più elevato hanno una priorità maggiore di essere aggiunte al blocco successivo.

Se una transazione richiede `15` gas ma abbiamo impostato un limite di `10` gas, l'esecuzione si interromperà quando verrà raggiunto il limite, e tutto il gas andrà perso. Se, invece, il limite di gas è superiore a quello utilizzato dalla transazione, il gas in eccesso verrà restituito al mittente.

Ricapitolando, il costo totale di una transazione può essere calcolato con la seguente formula:

$$\text{Costo transazione} = \text{Limite di gas} \times \text{Prezzo del gas}$$

Oltre a compensare i validatori, il costo delle transazioni su Ethereum serve anche a proteggere la blockchain da attacchi, come il DDoS (Distributed Denial of Service), e da errori di programmazione negli smart contract che potrebbero sovraccaricare il sistema. Ad esempio, se una computazione finisse in un ciclo infinito, la gestione del gas interromperebbe l'esecuzione una volta che il gas disponibile viene esaurito. Questa caratteristica ci consente di definire Ethereum come un linguaggio (quasi) Turing-completo, poiché siamo sicuri che ogni programma in esecuzione terminerà [5].

2.3 Smart Contract

Il concetto di smart contract è stato introdotto per la prima volta da Nick Szabo nel 1996, definendolo come *"un protocollo di transazione digitale che esegue i termini di un contratto"* [13]. Lo scopo principale di uno smart contract è quello di automatizzare l'adempimento delle condizioni contrattuali, riducendo al minimo il rischio di azioni malevole e la necessità di fidarsi degli intermediari (rischio di controparte).

Un contratto, in generale, è un accordo legalmente vincolante tra due o più parti e svolge un ruolo fondamentale nell'instaurare fiducia tra i soggetti coinvolti in una transazione[5]. Può essere tanto semplice quanto un biglietto dell'autobus o molto complesso come un contratto di lavoro.

Nel contesto delle blockchain, uno smart contract è un programma che replica tutte le caratteristiche di un contratto tradizionale, ma viene memorizzato ed eseguito all'interno di una blockchain. Si tratta di un agente autonomo che risiede su una blockchain, senza la necessità di un'entità esterna per valutare le condizioni e prendere decisioni. Questo ruolo è sostituito dal consenso della rete. Gli smart contract stabiliscono le regole e le fanno rispettare automaticamente alle parti coinvolte, senza la necessità di autorità centrali. Quando le condizioni del contratto vengono soddisfatte, lo smart contract esegue autonomamente azioni specifiche, come ad esempio il trasferimento di denaro [5].

Uno smart contract può essere paragonato a un'applicazione IFTTT (If This Then That) che risponde ad eventi specifici (Figura 2.3).

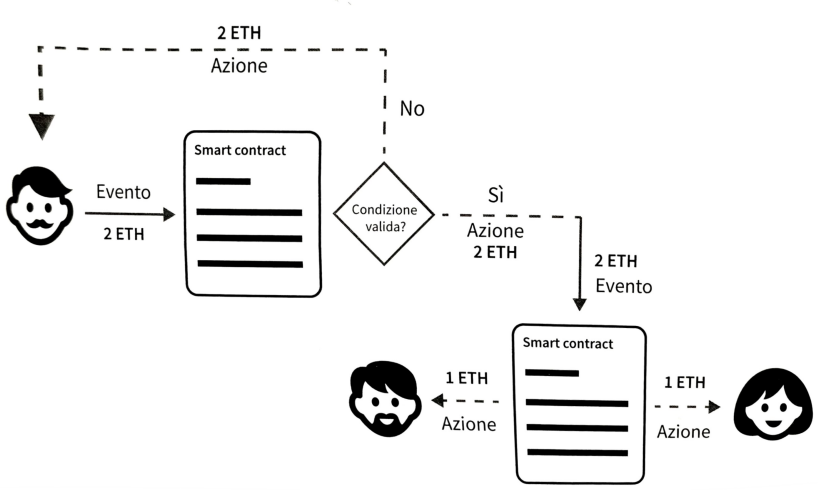


Figura 2.3: Il processo IFTTT negli smart contract.

[5] `https://www.treccani.it/enciclopedia/contratto/`

Gli smart contract, solitamente scritti in un linguaggio di programmazione di alto livello come Solidity, necessitano di essere compilati nel bytecode di basso livello eseguito nell'EVM per poter essere eseguibili. Dopo la compilazione, vengono distribuiti sulla rete Ethereum attraverso una speciale transazione di creazione del contratto, inviata all'indirizzo `0x0`, il quale è anche detto *zero address*. Ogni contratto è identificato da un indirizzo Ethereum, derivato dalla transazione di creazione del contratto in funzione dell'account di origine e del nonce. Come già menzionato nella Sezione 2.2.2, a differenza degli EOAs, gli account creati per nuovi smart contract non hanno chiavi associate.

Gli smart contract vengono eseguiti solo quando vengono richiamati da una transazione. Ogni smart contract in Ethereum viene eseguito a seguito di una transazione avviata da un EOA. Un contratto può richiamare un altro contratto, che a sua volta può richiamare un altro contratto e così via, ma il primo contratto in questa catena di esecuzione sarà sempre stato richiamato da una transazione proveniente da un EOA.

Le transazioni sono atomiche, indipendentemente dal numero di smart contract richiamati o dalle azioni che eseguono. Ogni transazione viene eseguita nella sua totalità e le eventuali modifiche allo stato globale vengono registrate solo se l'esecuzione termina con successo. Se l'esecuzione fallisce a causa di un errore, tutti gli effetti vengono annullati e la transazione viene registrata come tentata.

È importante notare che una volta distribuito, il codice di un contratto non può essere modificato. Approfondiremo ulteriormente questo aspetto in seguito.

2.3.1 Esempio di smart contract

Nel presente contesto, viene esaminato un esempio di smart contract[6], come mostrato nell'Algoritmo 1. Tale contratto si suddivide in due entità: *ReceiveEther* e *SendEther*.

Il contratto *ReceiveEther* è progettato per ricevere *ether*. Contiene due funzioni: `receive` e `fallback`. La funzione `receive` viene chiamata quando viene inviato *ether* al contratto e non ci sono dati associati alla transazione. Questo è possibile tramite l'invio di *ether* direttamente all'indirizzo del contratto senza specificare alcuna funzione. La funzione `fallback` viene invece chiamata quando viene inviato *ether* al contratto e sono presenti dati associati alla transazione (e.g., il payload dati che contiene informazioni aggiuntive inviate con la transazione). Entrambe le funzioni accettano *ether* e sono contrassegnate come `payable` per indicare che possono ricevere fondi. È importante specificare che un contratto che riceve *ether* deve avere definito almeno una di queste

[6]Tratto da: `https://solidity-by-example.org/sending-ether/`

due funzioni. La funzione `getBalance` restituisce il saldo attuale del contratto in *ether*.

Il contratto *SendEther* è progettato invece per inviare *ether*. Contiene una sola funzione `sendViaCall` che invia *ether* a un altro indirizzo Ethereum. Questa funzione prende in input l'indirizzo del destinatario e l'ammontare di *ether* da inviare. Utilizza la funzione `call` per inviare *ether* al destinatario e restituisce un booleano che indica se l'operazione è stata eseguita con successo. Infine, la funzione `require` permette di verificare che l'invio sia andato a buon fine, in caso contrario l'esecuzione viene interrotta con un messaggio di errore.

Algoritmo 1 Esempio di smart contract in Solidity.

```
pragma solidity ^0.8.24;

contract ReceiveEther {
    receive() external payable {}

    fallback() external payable {}

    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}

contract SendEther {
    function sendViaCall(address payable _to) public payable {
        (bool sent, bytes memory data) = _to.call{value: msg.value}("");
        require(sent, "Failed to send Ether");
    }
}
```

2.4 Ethereum Virtual Machine

Al cuore del protocollo e del funzionamento di Ethereum risiede l'Ethereum Virtual Machine (EVM). Questa componente software gestisce l'implementazione e l'esecuzione degli smart contract. Quasi ogni azione sulla rete Ethereum comporta un aggiornamento dello stato calcolato dall'EVM, ad eccezione delle semplici transazioni di trasferimento di valore da un EOA ad un altro. Ad un livello più alto, possiamo concepire l'EVM come un enorme computer decentralizzato, distribuito su scala globale, che contiene milioni di "programmi" eseguibili, ognuno dei quali con il proprio spazio dati permanente [10].

A livello matematico, possiamo definire l'EVM come una funzione matematica: dato un input, essa produce un output deterministico. In particolare, la funzione di transizione è definita come segue:

$$Y(S, T) = S'$$

dove S è il vecchio stato macchina valido e T è un insieme di transazioni valide che, se applicate allo stato S, producono lo stato S' [14].

Come detto in precedenza, l'EVM è una macchina quasi Turing-completa, il che significa che tutti i processi di esecuzione sono limitati a un numero finito di passaggi computazionali, determinati dalla quantità di gas disponibile per ogni esecuzione di uno smart contract. Questo limite assicura che ogni programma abbia una fine definita, evitando situazioni in cui l'esecuzione potrebbe protrarsi all'infinito, portando alla congestione dell'intera piattaforma Ethereum [10].

Dal punto di vista architetturale, l'EVM utilizza uno stack, ovvero una coda volatile di tipo LIFO (Last In First Out), per memorizzare tutti i valori in memoria, con una profondità massima di 1024 elementi. Opera su parole (*words*) di 256 bit, principalmente per agevolare le operazioni di hashing, e include diverse componenti di dati indirizzabili:

- una ROM virtuale contenente il codice del programma, immutabile una volta caricato il bytecode dello smart contract da eseguire;
- una memoria volatile (*Memory*), in cui ogni posizione è inizializzata a zero e può essere utilizzata temporaneamente durante l'esecuzione;
- uno spazio di archiviazione permanente (*Storage*), parte dello stato di Ethereum, anch'esso inizializzato a zero, che conserva informazioni a lungo termine.

Il ruolo principale dell'EVM consiste nell'aggiornare lo stato di Ethereum attraverso il calcolo di transizioni di stato valide derivanti dall'esecuzione del

codice degli smart contract. Questo concetto rende Ethereum una macchina a stati basata sulle transazioni, poiché gli attori esterni, come gli utenti della rete, iniziano le transizioni di stato creando, accettando e ordinando le transazioni.

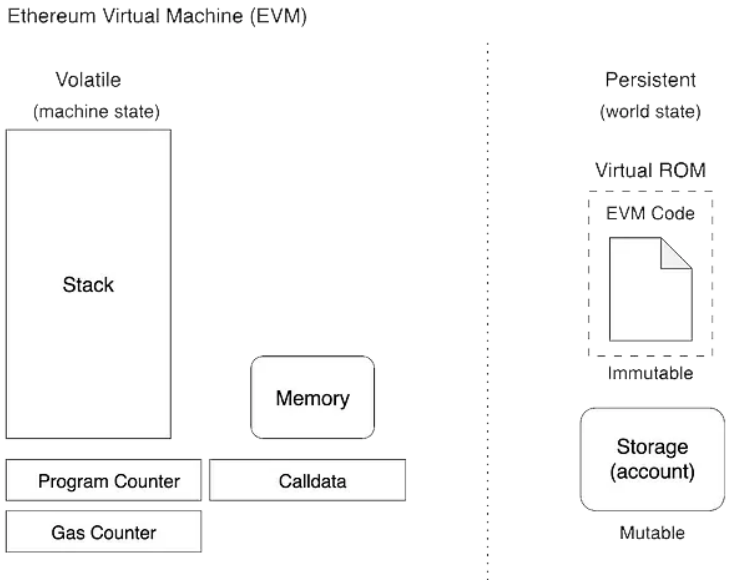


Figura 2.4: Componenti della Ethereum Virtual Machine.

Per comprendere meglio cosa costituisca lo stato di Ethereum, possiamo suddividerlo in due livelli. Al livello superiore troviamo lo stato globale di Ethereum, che è essenzialmente una mappatura degli indirizzi Ethereum (160 bit) agli account della rete [10].

Scendendo al livello inferiore, ogni indirizzo Ethereum rappresenta un account, che include diverse informazioni:

- il saldo in *ether*, espresso come il numero di wei posseduti dall'account;
- il *nonce*, che tiene traccia del numero di transazioni inviate con successo da quell'account, se si tratta di un account controllato dall'utente (EOA), o il numero di smart contract creati, se è un contract account;
- lo storage dell'account, che funge da archivio dati permanente utilizzato esclusivamente dagli smart contract;
- il codice del programma dell'account, presente solo se l'account è uno smart contract. Da notare che un account esterno controllato dall'utente avrà sempre codice nullo e uno storage vuoto.

Quando una transazione coinvolge l'esecuzione del codice di uno smart contract, viene creata un'istanza dell'EVM con tutte le informazioni necessarie

relative al blocco corrente in fase di creazione e alla transazione specifica in elaborazione. In questo processo, il codice dello smart contract viene caricato nell'EVM, il program counter viene impostato a zero, lo storage del contract account viene recuperato, la memoria viene inizializzata e le variabili di blocco e di ambiente vengono configurate.

Un aspetto fondamentale è la quantità di gas disponibile per questa esecuzione, determinata dalla quantità di gas pagata dal mittente all'inizio della transazione. Durante l'esecuzione del codice, la quantità di gas disponibile diminuisce in base al costo delle operazioni eseguite. Se il gas disponibile si esaurisce, viene lanciata un'eccezione "Out of Gas (OOG)" e l'esecuzione viene interrotta, annullando la transazione. Nessuna modifica viene apportata allo stato di Ethereum, tranne l'incremento del nonce del mittente e il pagamento delle commissioni al validatore.

Possiamo immaginare l'EVM che opera su una copia isolata dello stato effettivo di Ethereum, scartando completamente questa versione isolata se l'esecuzione non può essere completata. Tuttavia, se l'esecuzione ha successo, lo stato effettivo viene aggiornato per corrispondere alla versione isolata, inclusi eventuali cambiamenti nei dati di archiviazione del contratto chiamato, la creazione di nuovi smart contract e i trasferimenti di *ether* avviati [10].

2.5 EVM Bytecode

Come accennato in precedenza, Ethereum esegue gli smart contract all'interno dell'Ethereum Virtual Machine (EVM), un ambiente di runtime decentralizzato che si occupa dell'esecuzione del codice di tali contratti. Gli sviluppatori possono scrivere gli smart contract utilizzando diversi linguaggi di alto livello, come Solidity[7] o Vyper[8], ma affinché possano essere eseguiti dall'EVM, devono essere compilati in un linguaggio di basso livello noto come EVM bytecode. Il bytecode dell'EVM è un linguaggio a basso livello basato su stack che comprende circa 150 istruzioni chiamate opcode[9], le quali vengono interpretate dall'EVM per manipolare uno stack i cui elementi sono parole di 256 bit. Ogni istruzione è rappresentata da un numero esadecimale preceduto da `0x`.

Consideriamo questo breve frammento di bytecode dell'EVM:

Algoritmo 2 Frammento di bytecode dell'EVM.

```
60 01 60 02 01
```

[7] `https://docs.soliditylang.org/en/latest/`
[8] `https://docs.vyperlang.org/en/stable/toctree.html`
[9] Lista completa: `https://ethereum.github.io/yellowpaper/paper.pdf`

Il byte `60` corrisponde all'opcode `PUSH1`, il quale inserisce un byte nello stack. Il byte successivo è `01`, che corrisponde al valore che è stato messo in cima allo stack. Allo stesso modo, i byte `60 02` rappresentano l'istruzione `PUSH1 0x02` (lo stack dopo l'esecuzione di questi opcode è mostrato in Figura 2.5a). L'ultimo byte, ossia `01`, corrisponde all'opcode `ADD`, il quale preleva due elementi dallo stack, li somma e mette il risultato di nuovo sulla pila. Pertanto, la versione tradotta in linguaggio naturale della stringa di bytecode appena analizzata è mostrata nell'Algoritmo 3.

Algoritmo 3 Traduzione in linguaggio umano dell'Algoritmo 2.

```
PUSH1 0x01
PUSH1 0x02
ADD
```

Dopo l'esecuzione di queste istruzioni, l'elemento in cima allo stack avrà il valore `3` (lo stack dopo l'operazione `ADD` è mostrato nella Figura 2.5b).

Il set di istruzioni dell'EVM include una vasta gamma di operazioni, tra cui:

- operazioni aritmetiche e logiche bit a bit;
- richieste del contesto di esecuzione;
- accesso allo stack, alla memoria e allo storage;
- controllo del flusso di esecuzione;
- logging, calling e altre operazioni.

Oltre alle operazioni standard del bytecode, l'EVM offre anche accesso a informazioni specifiche sull'account (come indirizzo e saldo) e sul blocco (come numero di blocco e prezzo attuale del gas) [10].

Questo elaborato si concentrerà sull'analisi del controllo del flusso di esecuzione.

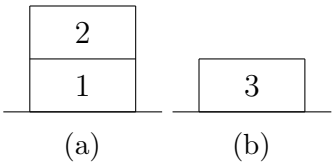


Figura 2.5: Stato dello stack (a) prima e (b) dopo l'esecuzione dell'opcode `ADD`.

2.6 Alterazione del flusso di esecuzione

Il flusso di esecuzione di uno smart contract (in EVM bytecode) inizia con il primo opcode e procede in sequenza. Gli unici opcode che possono alterare il flusso di esecuzione di uno smart contract senza interromperlo[10] sono `JUMP` e `JUMPI`.

L'istruzione `JUMP` comporta un salto incondizionato a una posizione specifica del programma, la quale è definita dall'indirizzo memorizzato nell'elemento più in alto dello stack (quindi il valore che viene estratto). Per semplificare, consideriamo il seguente frammento di codice:

Algoritmo 4 Esempio di EVM bytecode con l'opcode `JUMP`.

```
PUSH1 0x10
PUSH1 0x20
JUMP
```

Quando l'istruzione `JUMP` viene eseguita, viene prelevato l'elemento in cima allo stack (in questo caso `0x20`) e il programma viene instradato verso l'istruzione all'indirizzo `0x20`, da cui procederà l'esecuzione.

D'altra parte, l'istruzione `JUMPI` rappresenta un salto condizionato: se l'opcode `JUMPI` viene soddisfatto, l'esecuzione salta all'indirizzo memorizzato nell'elemento più in alto dello stack solo se l'elemento sottostante (cioè il secondo elemento più in alto) è diverso da zero. In caso contrario, l'esecuzione continua con l'opcode successivo. In entrambi i casi, i due elementi più in alto dello stack vengono estratti. Adesso consideriamo il frammento di codice mostrato nell'Algoritmo 5.

[10] L'esecuzione di uno smart contract può terminare in vari modi: (a) in modo implicito, quando il program counter supera l'ultimo opcode del programma; (b) in modo esplicito, durante l'elaborazione di opcode come `STOP`, `RETURN`, `REVERT`, `SELFDESTRUCT`, `INVALID`; o (c) in modo eccezionale, quando si verificano condizioni illegali, come ad esempio lo stack underflow.

Algoritmo 5 Esempio di EVM bytecode con l'opcode `JUMPI`.

```
PUSH1 0x10
PUSH1 0x20
JUMPI
```

In questo scenario, la prima istruzione `PUSH1` inserisce il valore `0x10` nello stack, mentre la seconda istruzione `PUSH1` inserisce il valore `0x20`. Successivamente, l'istruzione `JUMPI` valuta il secondo elemento più alto nello stack (che è `0x10`). Poiché questo valore non è zero, l'esecuzione salta all'indirizzo memorizzato nel primo elemento dello stack (che è `0x20`).

È importante notare che la destinazione del salto, ossia l'indirizzo corrispondente all'elemento più in alto nello stack, deve corrispondere all'opcode `JUMPDEST`, altrimenti l'esecuzione terminerà in modo eccezionale. L'istruzione `JUMPDEST` non modifica lo stack; serve semplicemente a segnalare le posizioni del programma in cui è consentito un salto (condizionato o incondizionato).

2.6.1 Esempio di alterazione

Immaginiamo di voler creare un semplice ciclo utilizzando le istruzioni `JUMP`, `JUMPI` e `JUMPDEST` (Algoritmo 6).

Algoritmo 6 Esempio di un ciclo While in EVM bytecode.

```
0     PUSH1 0x00
2     JUMPDEST
3     PUSH1 0x03
5     DUP2
6     LT
7     PUSH1 0x0D
9     JUMPI
0A    PUSH1 0x14
0C    JUMP
0D    JUMPDEST
0E    PUSH1 0x01
10    ADD
11    PUSH1 0x02
13    JUMP
14    JUMPDEST
15    STOP
```

Esaminiamo ora passo dopo passo l'esecuzione dell'Algoritmo 6:

1. `PUSH1 0x00`: Questa istruzione inserisce `0x00` nello stack.
2. `JUMPDEST`: Contrassegna una destinazione di salto.

3. `PUSH1 0x03`: Inserisce `0x03` nello stack.

4. `DUP2`: Duplica il secondo valore più alto nello stack; in questo caso, `0x00` (Figura 2.6a).

5. `LT`: Rimuove e confronta i due valori più alti nello stack. Se il valore superiore è inferiore al secondo, il risultato è `1`; altrimenti è `0` (in questo caso, è `1`).

6. `PUSH1 0x0D`: Inserisce `0x0D` nello stack (Figura 2.6b).

7. `JUMPI`: Se il secondo valore più alto nello stack è diverso da zero, passa al blocco di codice all'indirizzo in cima allo stack, altrimenti passa all'opcode successivo (in questo caso salta a `0x0D`, che corrisponde ad una destinazione di salto valida).

8. Ora il nostro Program Counter (PC) è impostato su `0x0D`. L'operazione successiva è una `PUSH`.

9. `PUSH1 0x01`: Inserisce `0x01` nello stack (Figura 2.6c).

10. `ADD`: Rimuove e somma i due valori più alti dello stack e mette il risultato in cima allo stack (in questo caso, il risultato è `1`).

11. `PUSH1 0x02`: Inserisce `0x02` nello stack (Figura 2.6d).

12. `JUMP`: Salta al blocco di codice all'indirizzo `0x02` e inizia la seconda iterazione del ciclo.

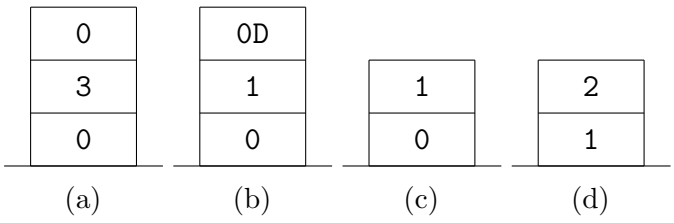


Figura 2.6: Stato dello stack dell'Algoritmo 6 (a) al punto `4`, (b) al punto `6`, (c) al punto `9` e (d) al punto `11`.

Il programma inizia un ciclo in cui ad ogni iterazione controlla una condizione `LT`. Se la condizione è vera, procede lungo un percorso specifico; altrimenti, segue un percorso alternativo. Ciò che lo rende interessante è che la condizione

cambia al termine della seconda iterazione: nelle prime due iterazioni, il percorso seguito è determinato dalla condizione vera, mentre nella terza iterazione la condizione diventa falsa, portando il programma lungo un altro percorso.

Questo comportamento può essere visualizzato con la Figura 2.7, la quale rappresenta il Control-Flow Graph (CFG) relativo al programma. Il CFG mostra tutti i possibili percorsi che il programma può intraprendere, insieme alle condizioni che determinano quale percorso seguire. In questo caso, il CFG ha un ciclo che rappresenta l'esecuzione ripetuta del programma, e due rami che rappresentano i due possibili percorsi che il programma può intraprendere a seconda della condizione `LT`.

Per semplificare, l'Algoritmo 6 può essere rappresentato in C con l'Algoritmo 7.

Algoritmo 7 Implementazione in C dell'Algoritmo 6.

```
1 int x = 0;
2 while (x < 3) {
3     x++;
4 }
5 return;
```

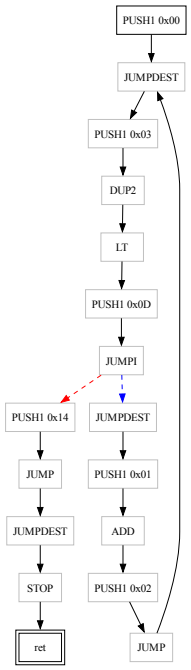


Figura 2.7: CFG dell'Algoritmo 6.

2.6.2 Pushed jumps vs Orphan jumps

Dopo aver spiegato come nell'EVM bytecode si possa influenzare il flusso di esecuzione attraverso gli opcode `JUMP` e `JUMPI`, diventa evidente che la destinazione di questi salti non è staticamente definita come i dati nelle istruzioni `PUSH`, ma bensì viene calcolata dinamicamente considerando gli elementi presenti nello stack.

Tuttavia, ci sono situazioni in cui è possibile determinare staticamente la destinazione di un salto senza dover eseguire effettivamente lo smart contract. Ad esempio, nei due frammenti di codice esaminati in precedenza (Algoritmo 4 e Algoritmo 5), le destinazioni dei salti possono essere dedotte dal codice sorgente poiché entrambi gli opcode sono preceduti da un'istruzione `PUSH`. Questi salti sono definiti *Pushed Jumps* [15]. Essi non generano complicazioni nella creazione del CFG poiché le destinazioni dei salti possono essere determinate a tempo di compilazione.

Ciononostante, una categoria di salti più complessa da gestire è rappresentata dalle *Orphan Jumps* [15]. Un esempio di jump orfana è il seguente:

In questo caso, la destinazione dell'istruzione `JUMP` non può essere immediatamente individuata tramite un'analisi puramente sintattica del codice sorgente; per gestire correttamente il salto e costruire un CFG accurato, è necessaria

Algoritmo 8 Esempio di Orphan Jump.

```
PUSH1 0x0A
PUSH1 0x0C
ADD
JUMP
```

un'analisi del programma in grado di dedurre il contenuto potenziale dello stack durante l'esecuzione. Va notato che l'analisi delle jump orfane è molto complessa e un'implementazione inaccurata dello smart contract potrebbe essere sfruttata da malintenzionati per attacchi malevoli (questo problema verrà approfondito nella sezione 2.8).

2.7 L'importanza di costruire un CFG completo

Garantire la sicurezza e l'affidabilità degli smart contract è di vitale importanza per il successo della piattaforma Ethereum. Come già detto in precedenza, una volta implementati, questi contratti non possono essere modificati, il che significa che eventuali bug o vulnerabilità presenti nel codice possono causare conseguenze irrevocabili, come la perdita di fondi o l'esecuzione di azioni indesiderate. Pertanto, è fondamentale assicurarsi che gli smart contract siano privi di problemi prima della loro implementazione.

Per raggiungere questo obiettivo, i ricercatori hanno sviluppato diverse tecniche per analizzare e verificare il codice degli smart contract. Una tecnica possibile è l'analisi statica, che esamina il codice senza effettivamente eseguirlo. L'analisi statica si basa spesso sui Control-Flow Graph (CFG) per identificare potenziali problemi di sicurezza.

Un CFG è una rappresentazione grafica di tutti i percorsi che potrebbero essere attraversati attraverso un programma durante l'esecuzione.

Tuttavia, nel contesto di Ethereum, costruire un CFG accurato e preciso per l'EVM bytecode non è banale. Questo perché alcune destinazioni dei salti sono determinate durante l'esecuzione e non durante la compilazione. Di conseguenza, non è sempre semplice costruire un CFG completo e preciso, che includa tutti i possibili percorsi di esecuzione.

Se non si riesce a costruire un CFG completo e preciso che includa tutti i possibili percorsi di esecuzione e risolva tutte le jump, si potrebbero lasciare aperture per vulnerabilità nel codice.

2.8 Vulnerabilità di rientranza

Gli smart contract, essendo programmi eseguiti sulla blockchain, possono essere soggetti a vulnerabilità, proprio come qualsiasi altro software. Tuttavia, le vulnerabilità negli smart contract sono particolarmente rischiose, poiché, come già spiegato nella Sezione 2.3, una volta che il contratto viene pubblicato sulla blockchain, non può essere più modificato. Tra le vulnerabilità più comuni individuate nella documentazione di Ethereum possiamo trovare la rientranza, l'overflow/underflow di interi e la manipolazione degli oracoli [2]. In questo elaborato, ci concentreremo sulla vulnerabilità di rientranza.

Per spiegare la rientranza, è importante notare che l'EVM non supporta la concorrenza, il che significa che due contratti coinvolti in una chiamata di messaggio non possono essere eseguiti contemporaneamente. Quando viene effettuata una chiamata esterna (`msg.call`), l'esecuzione e la memoria del

contratto chiamante vengono sospese fino al completamento della chiamata. Successivamente, l'esecuzione riprende normalmente. Questo processo può essere formalmente descritto come il trasferimento del flusso di controllo a un altro contratto.

Sebbene in gran parte innocuo, il trasferimento del flusso di controllo a contratti non fidati può causare problemi, come nel caso della rientranza. Un attacco di rientranza si verifica quando un contratto malizioso viene chiamato all'interno di un contratto vulnerabile prima che l'invocazione della funzione originale sia completata. Questo tipo di attacco può essere meglio spiegato con il seguente esempio.

Supponiamo di avere uno smart contract `Victim` che consente a chiunque di depositare e prelevare *ether* (Algoritmo 9)[11].

Il contratto in questione offre una funzione chiamata `withdraw()`, che consente agli utenti di prelevare gli ETH precedentemente depositati nel contratto. Durante un prelievo, il contratto esegue diverse operazioni:

1. verifica il saldo di `ETH` dell'utente;
2. invia i fondi all'indirizzo del chiamante;
3. azzerà il saldo dell'utente, impedendo ulteriori prelievi.

Inizialmente, verifica se le condizioni necessarie sono soddisfatte (ad esempio, se l'utente ha un saldo di `ETH` positivo). Successivamente, esegue l'interazione inviando gli `ETH` al chiamante prima di applicare gli effetti della transazione, come la riduzione del saldo dell'utente.

Quando `withdraw()` è chiamato da un utente (EOA), la funzione si comporta come previsto: `msg.sender.call.value()` trasferisce gli `ETH` al chiamante. Tuttavia, se `msg.sender` è uno smart contract e chiama `withdraw()`, l'invio dei fondi tramite `msg.sender.call.value()` attiverà anche l'esecuzione del codice archiviato a quell'indirizzo.

Supponiamo adesso di avere uno smart contract `Attacker` che vuole sfruttare la vulnerabilità di `Victim` (Algoritmo 10).

Questo contratto è progettato per eseguire tre operazioni:

1. accettare un deposito da un altro account (probabilmente l'EOA dell'utente malevolo);
2. depositare `1 ETH` nel contratto `Victim`;
3. prelevare `1 ETH` archiviato nel contratto `Victim`.

[11]Tratto da: `https://ethereum.org/it/developers/docs/smart-contracts/security/`

Algoritmo 9 Vulnerabilità di rientranza in uno smart contract vittima.

```
1  contract Victim {
2      mapping (address => uint256) public balances;
3
4      function deposit() external payable {
5          balances[msg.sender] += msg.value;
6      }
7
8      function withdraw() external {
9          uint256 amount = balances[msg.sender];
10         (bool success, ) = msg.sender.call.value(amount)("");
11         require(success);
12         balances[msg.sender] = 0;
13     }
14 }
```

In apparenza, tutto sembra regolare, tranne per un dettaglio: l'`Attacker` ha incluso una funzione che richiama nuovamente `withdraw()` in `Victim` se il gas rimanente da `msg.sender.call.value` in entrata è superiore a `40.000`. Questo consente all'`Attacker` di rientrare nel contratto `Victim` e prelevare ulteriori fondi, prima che la prima invocazione di `withdraw()` di `Victim` sia completata.

Ciò accade perché in Solidity sono presenti le *funzioni di fallback*, dette anche *funzioni di default*, le quali sono definite con la forma `function()` e vengono invocate quando un contratto riceve una transazione senza specificare alcuna funzione. Il contratto `Attacker` è progettato per accettare fondi in arrivo da un'altra fonte. Quando riceve una transazione di `ETH`, viene automaticamente invocata la funzione di fallback `function()` (riga 7).

Algoritmo 10 Smart contract che sfrutta la vulnerabilità dell'Algoritmo 9.

```
1  contract Attacker {
2      function beginAttack() external payable {
3          Victim(victim_address).deposit.value(1 ether)();
4          Victim(victim_address).withdraw();
5      }
6
7      function() external payable {
8          if (gasleft() > 40000) {
9              Victim(victim_address).withdraw();
10         }
11     }
12 }
```

In sintesi, poiché il saldo del chiamante non viene azzerato fino al termine dell'esecuzione della funzione, le successive invocazioni avranno successo, permettendo al chiamante di prelevare ripetutamente il proprio saldo [16]. Gli

attacchi di rientranza rimangono ancora oggi una grave preoccupazione per gli smart contract, come dimostrato dalle pubbliche segnalazioni di exploit di rientranza[12].

2.8.1 Attacco a The DAO (2016)

Quando si affronta il tema delle vulnerabilità di rientranza, è inevitabile menzionare l'attacco a The DAO del 2016, il quale rappresenta uno degli episodi più celebri nella storia di Ethereum.

The DAO era una startup che gestiva un fondo d'investimento in *ether* ed operava come uno smart contract su Ethereum. Il nome "The DAO" venne scelto dai fondatori come riferimento al concetto di Organizzazione Autonoma Decentralizzata (DAO). The DAO era un contratto che consentiva agli utenti di depositare *ether* e ricevere in cambio dei token, utilizzabili per votare i progetti da finanziare. La sua popolarità crebbe rapidamente, diventando il più grande crowdfunding della storia (fino ad allora), con oltre 150 milioni di dollari in *ether* raccolti da oltre 10.000 investitori [17].

Fin dall'inizio, il fondo si impegnò ad operare in conformità con i termini e le condizioni del proprio smart contract, che altro non erano che il codice di un programma situato sulla blockchain.

Tuttavia, The DAO divenne famosa a causa di una vulnerabilità all'interno del proprio programma che permise a un utente sconosciuto di prelevare un terzo dei fondi. La perdita ammontò a 3,6 milioni di *ether*, equivalenti a circa 60 milioni di dollari all'epoca (circa 12 miliardi di dollari ai prezzi attuali). Data l'ampia partecipazione e le implicazioni negative, Ethereum si trovò sotto forte pressione pubblica e i leader della rete decisero di eseguire un *hard-fork retroattivo* (rollback) della blockchain per annullare l'attacco.

Attraverso l'hard-fork, i fondi di The DAO furono trasferiti su un indirizzo di recupero, cancellando virtualmente la fuga di fondi. Successivamente, gli utenti del fondo poterono reclamare i propri investimenti. Ciononostante, l'hard-fork generò un significativo dissenso tra gli utenti della rete Ethereum, con alcuni dei quali che continuarono ad utilizzare la blockchain originale di Ethereum, ora denominata Ethereum Classic[13] (ETC). Tale rete opera ancora oggi, mantenendo la catena di blocchi originale, con l'utente sconosciuto che possiede ancora i fondi sottratti [18].

[12] `https://github.com/pcaversaccio/reentrancy-attacks`
[13] `https://ethereumclassic.org/`

Capitolo 3

Analisi Statica

Come già esposto nella Sezione 2.8, gli smart contract, essendo essenzialmente software eseguiti sulla blockchain, sono soggetti a vulnerabilità e errori di programmazione, proprio come qualsiasi altro software. Per mitigare questi rischi, possiamo ricorrere all'*analisi dinamica*, un approccio che consiste nell'osservare il comportamento di un software durante l'esecuzione, ad esempio tramite l'uso di asserzioni. Tuttavia, l'analisi dinamica presenta una limitazione fondamentale: non è in grado di determinare se un programma termina, poiché ciò richiederebbe di eseguirlo per un tempo indefinito.

Dall'altra parte, l'*analisi statica* offre un approccio più sofisticato: consente di verificare la correttezza di un programma senza eseguirlo effettivamente, permettendo di verificare a tempo di compilazione alcune proprietà che valgono a tempo di esecuzione. Tuttavia, è importante notare che l'analisi statica non risolve il problema dell'analisi dinamica, poiché è impossibile dimostrare la terminazione di un *qualsiasi* programma. Questo limite deriva dal fatto che il problema della terminazione di un programma è indecidibile, come dimostrato da Alonso Church e Alan Turing nel 1936.

3.1 Interpretazione astratta

L'*interpretazione astratta*[1] si presenta come una teoria formale per approssimare oggetti matematici, concentrandosi principalmente sulle relazioni tra di essi anziché sulla natura intrinseca degli oggetti stessi. Una volta stabilita la correlazione tra gli oggetti concreti e quelli astratti, l'obiettivo è di eseguire i calcoli in modo completo sugli oggetti astratti al fine di ottenere informazioni sui corrispondenti oggetti concreti.

L'applicazione primaria dell'interpretazione astratta è nell'ambito dell'analisi statica, dove viene impiegata per approssimare i comportamenti concreti di un sistema tramite una versione astratta. Formalmente, il comportamento effettivo di un sistema è definito come la sua *semantica concreta*, mentre l'astrazione di tale comportamento è denominata *semantica astratta*.

Ma perché ricorriamo alle astrazioni? Quando si tratta di esaminare un sistema o un programma, il suo significato, ovvero la sua semantica concreta, può essere modellato come un oggetto matematico, come ad esempio un insieme. Tuttavia, spesso questo insieme risulta infinito e, in generale, il calcolo di tutti i possibili comportamenti di un programma diventa indecidibile. Di conseguenza, a causa del *Teorema di Rice*[2], diventa difficile ragionare sulle proprietà non banali del programma basandosi esclusivamente sulla sua semantica concreta.

L'idea fondamentale dell'interpretazione astratta è quella di stabilire una connessione tra mondi concreti e astratti, e successivamente utilizzare il mondo astratto come base per dedurre informazioni sul mondo concreto. Questa introduzione di un certo livello di astrazione consente di raggiungere la decidibilità, anche se si va a perdere precisione sugli oggetti concreti osservati. Concretamente, l'interpretazione astratta si focalizza su una visione ad alto livello della semantica, interessandosi unicamente alle proprietà rilevanti e trascurando i dettagli concreti non significativi per le suddette proprietà [19].

[1] In inglese *Abstract Interpretation* (AI).

[2] Il Teorema di Rice afferma che è impossibile automatizzare la verifica della soddisfacibilità di una proprietà semantica non banale per ogni programma scritto in un linguaggio Turing-completo.

3.2 Analisi statica mediante interpretazione astratta

L'utilizzo dell'analisi statica mediante l'interpretazione astratta consente di descrivere un programma attraverso un Control-Flow Graph (CFG). Un CFG non è altro che una rappresentazione grafica di tutti i percorsi che potrebbero essere attraversati attraverso un programma durante la sua esecuzione, e in termini accademici viene indicato come $G = \langle N, E \rangle$. Questo CFG rappresenta un insieme finito di nodi $N = \{n_1, n_2, ..., n_m\}$ che corrispondono ai punti di controllo del programma, e un insieme finito di archi $E \subseteq N \times N$ che mettono in relazione i nodi.

Supponiamo di avere un dominio astratto $\mathcal{A}$ che approssima il dominio concreto $\mathcal{C}$ utilizzato per analizzare i programmi in questione. Ogni nodo $n \in N$ è associato a una *funzione di trasformazione* $f_n : \mathcal{A}^m \to \mathcal{A}$, con $m = |N|$, che cattura gli effetti del nodo n (i.e., la sua semantica astratta).

L'analisi di un CFG $C = \langle N, E \rangle$, dove $N = \{n_1, n_2, ..., n_m\}$, implica la risoluzione di un sistema di equazioni:

$$F = \{x_i = f_i(x_1, x_2, ..., x_m) \mid i = 1, 2, ..., m\}.$$

Tuttavia il *punto fisso minimo* sul dominio concreto $\mathcal{C}$ non è calcolabile in modo finito. Di conseguenza, l'approccio consiste nel calcolare un *punto fisso astratto* su un dominio astratto $\mathcal{A}$, che approssimi il *punto fisso concreto*.

Il concetto di `fixpoint` (*punto fisso*) è centrale in questo contesto. In sostanza, un fixpoint rappresenta un punto in cui il risultato di un'iterazione o di un processo di calcolo non cambia più se riapplicato. Nel contesto dell'analisi statica, il fixpoint è raggiunto quando l'approssimazione della soluzione diventa stabile e non cambia più attraverso iterazioni successive. Questo è importante perché indica che si è giunti ad una conclusione definitiva o ad una soluzione approssimativa ottimale per il problema in questione [20, 21].

Altri due concetti di rilevanza sono il *top* ($\top$) e il *bottom* ($\bot$), entrambi fondamentali per comprendere la teoria che sarà discussa nei capitoli successivi. Il *top* rappresenta un valore che esiste ma non può essere rappresentato in modo determinato, poiché la sua natura è indeterminabile con il dominio simbolico che si sta utilizzando. Al contrario, il *bottom* indica un qualcosa che non può esistere nello stato attuale, essendo irraggiungibile.

3.3 Esempio dimostrativo

Consideriamo un contesto in cui il dominio concreto $\mathcal{C}$ è rappresentato dall'insieme dei numeri reali $\mathbb{R}$. In parallelo, esiste un dominio astratto $\mathcal{A}$ che, tramite un'interpretazione astratta, mira ad emulare il comportamento di $\mathcal{C}$ utilizzando le operazioni di addizione, sottrazione e moltiplicazione, allo scopo di studiare il segno dei risultati ottenuti.

In questo esempio, ci concentriamo sull'analisi dei segni, tralasciando quindi il valore effettivo dei numeri [21]. Supponiamo di avere due numeri positivi, indicati rispettivamente con a e b.

Quando sommiamo due numeri positivi $(a + b)$, il risultato è facilmente determinabile, in quanto siamo certi che risulterà positivo.

Analogamente, nel caso della moltiplicazione di un numero positivo per un numero negativo $(a \times (-b))$, il segno del risultato è banalmente identificabile come negativo.

Tuttavia, quando sottraiamo un numero positivo da un altro numero positivo $(a - b)$, la situazione diventa più complessa e non possiamo stabilire con certezza il segno del risultato se non teniamo conto anche del valore che hanno i numeri. In questo scenario, non siamo in grado di rappresentare il risultato in modo definito e utilizziamo il simbolo *top* $(\top)$ per indicare questa indeterminatezza.

Capitolo 4

EVMLiSA

In questo capitolo verrà introdotto EVMLiSA, uno strumento progettato per l'analisi e la generazione del Control-Flow Graph (CFG) di programmi bytecode EVM. Esamineremo il processo di creazione del CFG e ci focalizzeremo sull'approccio adottato per risolvere le *jump orfane*.

4.1 LiSA

LiSA[1], noto come acronimo di **Li**brary for **S**tatic **A**nalysis, rappresenta una piattaforma open-source sviluppata nel linguaggio di programmazione Java, concepita con l'intento di semplificare il processo di progettazione e sviluppo di analizzatori statici. Il suo obiettivo primario consiste nell'elaborare il CFG di un programma dato in input. Tale CFG è un grafo diretto nel quale i nodi rappresentano le istruzioni del programma, mentre gli archi denotano il flusso di esecuzione tra tali istruzioni. Questo approccio consente di esaminare il CFG prodotto e di verificare se il programma in questione soddisfa determinate proprietà predefinite [22].

EVMLiSA[2] estende le funzionalità di LiSA per consentire l'analisi del bytecode degli smart contract sviluppati su Ethereum.

[1] `https://github.com/lisa-analyzer/lisa`
[2] `https://github.com/lisa-analyzer/evm-lisa`

4.2 Struttura dei CFG in LiSA

La struttura dei CFG in LiSA è concepita con un'ottica di massima flessibilità, adattando una struttura capace di rappresentare funzioni, metodi e procedure di diversi linguaggi di programmazione [22].

All'interno della struttura dei Control-Flow Graph, sono presenti due elementi fondamentali: i nodi e gli archi. I nodi, rappresentati dalla classe `Node`, corrispondono agli statement del programma, ovvero alle singole istruzioni che compongono il codice. Gli archi, invece, sono rappresentati dalla classe `Edge` e servono a stabilire i collegamenti tra i nodi [22]. Quest'ultimi si dividono in tre tipologie principali:

- `SequentialEdge`, il quale descrive un flusso sequenziale o incondizionato dal nodo sorgente al nodo destinazione, indicando un percorso lineare e non condizionato all'interno del grafo.
- `TrueEdge`, che viene utilizzato per modellare un flusso condizionato dal nodo sorgente al nodo destinazione. Questo tipo di arco viene attraversato solo se la condizione specificata nel nodo sorgente è valutata come vera.
- `FalseEdge`, il quale rappresenta un flusso condizionato dal nodo sorgente al nodo destinazione, ma viene percorso solo nel caso in cui la condizione specificata nel nodo sorgente non sia soddisfatta.

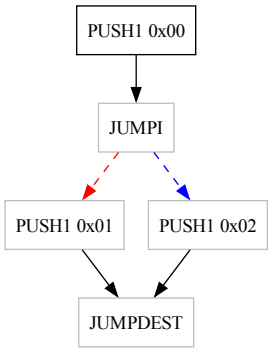


Figura 4.1: Esempio di CFG con (i) `SequentialEdge` in nero, (ii) `TrueEdge` in blu, (iii) `FalseEdge` in rosso.

4.3 Generazione dei CFG in EVMLiSA

Il processo iniziale condotto da EVMLiSA per generare il Control-Flow Graph di uno smart contract prevede la lettura del bytecode associato al contratto stesso. Una volta acquisiti tutti gli opcode pertinenti al contratto, è possibile avviare l'analisi e lo studio del flusso di esecuzione del programma.

In generale, il flusso di esecuzione è tipicamente lineare: l'ambiente di esecuzione della EVM procede nell'esecuzione delle istruzioni una dopo l'altra. Tuttavia, alcuni opcode hanno la capacità di alterare questo flusso standard.

Come precedentemente discusso nella Sezione 2.6, gli opcode principali responsabili di modificare il flusso di esecuzione sono `JUMP` e `JUMPI`:

- `JUMP`: rimuove un valore dalla cima dello stack e consente di saltare all'indirizzo specificato da tale valore. Questa operazione corrisponde ad un salto incondizionato.
- `JUMPI`: rimuove due valori dalla cima dello stack e salta all'indirizzo indicato dal primo valore estratto solo se il secondo valore estratto è diverso da zero; altrimenti prosegue il flusso sequenzialmente. Questo tipo di salto è condizionale. Al fine di modellare in modo accurato questo flusso, vengono creati sia un *true edge* che un *false edge*, riflettendo il flusso sequenziale della logica condizionale (Figura 4.1).

Un salto è considerato valido se e soltanto se raggiunge un'istruzione `JUMPDEST`, un particolare opcode utilizzato come marcatore per le destinazioni di salto. Questo marcatore serve spesso a individuare i blocchi di codice, rendendo più agevole la successiva analisi del programma [15].

Oltre agli opcode che influenzano il flusso di esecuzione, esistono anche istruzioni che determinano la terminazione dell'esecuzione del contratto. Queste istruzioni includono:

- `STOP`: interrompe l'esecuzione del contratto.
- `RETURN`: conclude l'esecuzione ritornando un valore.
- `REVERT`: interrompe l'esecuzione ripristinando eventuali cambiamenti di stato effettuati, ma restituendo i dati e il gas rimanente.
- `SELFDESTRUCT`: termina l'esecuzione del contratto e registra l'account per l'eliminazione.
- `INVALID`: segnala un'istruzione non valida e interrompe il flusso di esecuzione.

4.4 Risoluzione delle jump orfane

Per illustrare il funzionamento dell'algoritmo di risoluzione delle jump, esaminiamo l'esecuzione di un frammento di bytecode EVM, mostrato nella Figura 4.2a, che include una jump orfana.

Il processo di costruzione del Control-Flow Graph di un bytecode EVM è descritto attraverso lo pseudocodice riportato nell'Algoritmo 11, dove la procedura principale è BUILDCFG (righe 1–6).

Questa funzione accetta come input il bytecode ed inizia creando un *partial CFG*, cioè un CFG senza la risoluzione dei salti (riga 2). Supponiamo di prendere in esempio il bytecode EVM mostrato in Figura 4.2a, il *partial CFG* ottenuto è illustrato nella Figura 4.2b. È importante notare che in questa fase è stato risolto solo il ramo `false` della `JUMPI` (arco rosso), poiché rappresenta l'opcode successivo sintatticamente all'istruzione `JUMPI` nel bytecode sorgente.

Successivamente, procediamo con la funzione JUMPRESOLVER (righe 8–24), la quale esegue l'analisi basata sullo stack astratto sul CFG in ingresso (riga 10). Questa operazione calcola le invarianti di ingresso e di uscita per ciascun nodo del CFG, ovvero lo stack astratto che il nodo riceve in input e lo stack risultante dopo l'applicazione della semantica dell'opcode sullo stack astratto di input.

Le righe 13–24 sono responsabili dell'esame dei risultati dell'analisi dei nodi di salto, in particolare dei nodi `JUMP` (righe 14–18) e dei nodi `JUMPI` (righe 19–23). In entrambi i casi, la risoluzione del salto segue lo stesso principio.

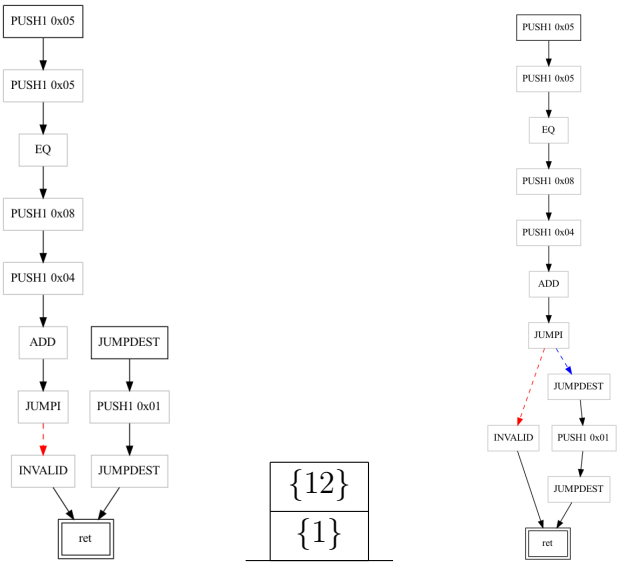
Concentrandoci sul caso della `JUMPI` dell'esempio attuale, la riga 21 verifica l'elemento in cima allo stack astratto arrivato al nodo di salto. Per ogni elemento *pc* contenuto nell'insieme intero, viene aggiunto un *true edge* (arco blu) dal nodo di salto al nodo $stmt(pc)$ (i.e., rappresenta l'istruzione corrispondente al program counter *pc*) nell'insieme degli archi.

Nel nostro esempio di esecuzione, lo stack astratto di input dell'istruzione `JUMPI` è quello illustrato in Figura 4.2c, di conseguenza l'istruzione alla riga 22 aggiunge un arco `true` dal nodo `JUMPI` al nodo con il program counter uguale a `12`, cioè il nodo `JUMPDEST`, come mostrato in Figura 4.2d.

Se almeno un arco viene aggiunto al CFG, allora la funzione JUMPSOLVER restituisce `true`. La procedura principale BUILDCFG continua a tentare di risolvere i target dei salti tramite JUMPSOLVER finché almeno un arco non viene aggiunto al CFG (righe 3–5). In caso contrario, la procedura si interrompe e restituisce il CFG (riga 6).

Infine, viene riportato il CFG finale in Figura 4.2d.

```
1  PUSH1 0x05
2  PUSH1 0x05
3  EQ
4  PUSH1 0x08
5  PUSH1 0x04
6  ADD
7  JUMPI //orphan jump
8  INVALID
9  JUMPDEST
10 PUSH1 0x01
11 JUMPDEST
```



(a) (b) (c) (d)

Figura 4.2: (a) Esempio di esecuzione con jump orfana, (b) CFG iniziale, (c) stack di input di `JUMPI`, (d) CFG finale.

Algoritmo 11 Pseudocodice dell'algoritmo per la risoluzione delle jump.

```
1:  function buildCFG(Bytecode)
2:      G ← CFG(Bytecode)
3:      do
4:          fp ← jumpSolver(G)
5:      while fp;
6:      return G;
7:
8:  function jumpSolver(G)
9:      let G = ⟨N, E⟩;          ▷ N, E: insieme dei nodi e degli archi, rispettivamente
10:     A ← runAnalysis(G);
11:     S_in, S_out ← getStacks(A);
12:     fp ← false;
13:     for all (S_in, op, S_out) ∈ A do
14:         if op is JUMP then                                   ▷ JUMP case
15:             s ← top(S_in);
16:             for all pc ∈ s do
17:                 E ← E ∪ {op → stmt(pc)};
18:             fp ← true;
19:         if op is JUMPI then                                  ▷ JUMPI case
20:             s ← top(S_in);
21:             for all pc ∈ s do
22:                 E ← E ∪ {op → stmt(pc)};
23:             fp ← true;
24:     return fp;
```

Capitolo 5

Dominio degli stack astratti

Nella Sezione 2.4, abbiamo discusso di come l'Ethereum Virtual Machine (EVM) utilizzi uno stack volatile con una dimensione massima di 1024 elementi per elaborare le istruzioni in esecuzione. Questo stack è fondamentale nell'elaborazione dell'EVM bytecode, poiché le istruzioni agiscono direttamente su di esso. Per applicare i concetti di esecuzione simbolica introdotti nella Sezione 3.2, dobbiamo creare un dominio astratto che rappresenti lo stack in modo simbolico. In questa rappresentazione, ogni elemento dello stack è rappresentato da una variabile simbolica (astratta) anziché da un valore concreto. Poiché lo stack può essere soggetto a manipolazioni durante l'esecuzione, diventa essenziale definire una rappresentazione appropriata per il dominio delle variabili simboliche.

5.1 Variabili simboliche rappresentate con Ints_k

L'idea implementata è stata quella di utilizzare un insieme di interi Ints_k di dimensione k, con $k > 0$, per la rappresentazione delle variabili simboliche. Formalmente, possiamo definirlo nel seguente modo:

$$\mathsf{Ints}_k \triangleq \langle \wp_{\leq k}(\mathbb{Z}) \cup \{\top_{\mathsf{Ints}_k}\}, \sqcup_{\mathsf{Ints}_k}, \sqcap_{\mathsf{Ints}_k}, \top_{\mathsf{Ints}_k}, \varnothing \rangle,$$

dove gli elementi in $\wp_{\leq k}(\mathbb{Z})$ sono insiemi di interi aventi cardinalità *al più* di k, parzialmente ordinati per inclusione di sottoinsiemi, così che $\varnothing$ (*bottom*) sia l'elemento inferiore, e che l'elemento speciale $\top_{\mathsf{Ints}_k}$ (*top*) denoti un valore non rappresentabile.

Dati $s_1, s_2 \in \mathsf{Ints}_k$, con $k > 0$, il *least upper bound*[1] $\sqcup_{\mathsf{Ints}_k}$ (abbreviato con *lub*) è definito come segue:

$$s_1 \sqcup_{\mathsf{Ints}_k} s_2 \triangleq \begin{cases} \top_{\mathsf{Ints}_k} & \text{se } s_1 = \top_{\mathsf{Ints}_k} \vee s_2 = \top_{\mathsf{Ints}_k}; \\ \top_{\mathsf{Ints}_k} & \text{se } |s_1 \cup s_2| > k; \\ s_1 \cup s_2 & \text{altrimenti.} \end{cases}$$

In parole più chiare, il risultato è l'intervallo più piccolo che contiene entrambi gli insiemi di partenza s_1 e s_2 (*unione*).

In modo simile, il *greatest lower bound*[2] $\sqcap_{\mathsf{Ints}_k}$ (abbreviato con *glb*) è definito come:

$$s_1 \sqcap_{\mathsf{Ints}_k} s_2 \triangleq \begin{cases} s_1 & \text{se } s_2 = \top_{\mathsf{Ints}_k}; \\ s_2 & \text{se } s_1 = \top_{\mathsf{Ints}_k}; \\ s_1 \cap s_2 & \text{altrimenti.} \end{cases}$$

Il risultato è il più grande intervallo che contiene gli elementi in comune degli insiemi di partenza s_1 e s_2 (*intersezione*).

[1] In italiano è detto *limite superiore minimo.*
[2] In italiano è detto *limite inferiore massimo.*

5.2 Stack astratto di dimensione h

Dopo aver definito il dominio delle variabili simboliche, possiamo definire il dominio degli stack astratti di dimensione h. In particolare, gli elementi astratti appartengono al seguente insieme:

$$\mathcal{S}_{\mathsf{Ints}_k,h} \triangleq \{[s_0, s_1, \ldots, s_{h-1}] \mid \forall i \in [0, h-1] \,.\, s_i \in \mathsf{Ints}_k, h, k > 0\},$$

ovvero l'insieme degli stack aventi esattamente h Ints_k elementi, dove l'elemento in cima allo stack corrisponde all'elemento più a destra (s_{h-1}).

C'è da notare che gli stack concreti aventi meno di h elementi sono modellati da stack astratti con esattamente h elementi, riempiendo gli elementi mancanti con (un prefisso composto da) $\varnothing \in \mathsf{Ints}_k$. Ad esempio, in Figura 5.1a abbiamo uno stack astratto $\mathcal{S}_{\mathsf{Ints}_2,4}$ avente dimensione 3.

Pertanto, il dominio astratto dello stack di dimensioni h è definito come segue:

$$\mathsf{St}^{\#}_{k,h} \triangleq \langle \mathcal{S}_{\mathsf{Ints}_k,h} \cup \{\bot_{\mathsf{St}^{\#}_{k,h}}\}, \sqcup_{\mathsf{St}^{\#}_{k,h}}, \sqcap_{\mathsf{St}^{\#}_{k,h}}, \top_{\mathsf{St}^{\#}_{k,h}}, \bot_{\mathsf{St}^{\#}_{k,h}} \rangle,$$

dove $\bot_{\mathsf{St}^{\#}_{k,h}}$ è l'elemento inferiore speciale (*bottom*). Anche in questo caso dobbiamo notare che questo elemento speciale *bottom* descrive l'insieme vuoto degli stack, il quale corrisponde all'insieme singleton contenente solo lo stack vuoto (e non per esempio allo stack formato da $[\varnothing, \varnothing, \ldots, \varnothing]$).

L'elemento $\top_{\mathsf{St}^{\#}_{k,h}}$ (*top*) viene rappresentato come uno stack di dimensione h avente tutti gli elementi uguali a $\top_{\mathsf{Ints}_k}$, cioè $\top_{\mathsf{St}^{\#}_{k,h}} = [\top_{\mathsf{Ints}_k}, \top_{\mathsf{Ints}_k}, \ldots, \top_{\mathsf{Ints}_k}]$.

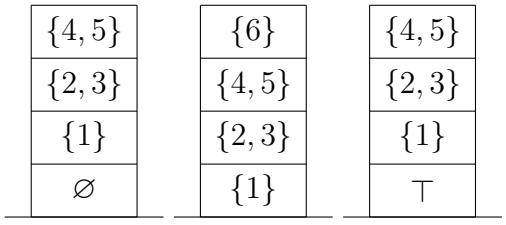


Figura 5.1: Esempi di stack astratti.

L'operatore *lub*, indicato come $\sqcup_{\mathsf{St}^{\#}_{k,h}}$, e l'operatore *glb*, indicato come $\sqcap_{\mathsf{St}^{\#}_{k,h}}$, vengono applicati agli elementi di due stack astratti di dimensione h non nulli, elemento per elemento. Questa operazione sfrutta il *lub* $\sqcup_{\mathsf{Ints}_k}$ e il *glb* $\sqcap_{\mathsf{Ints}_k}$ del dominio Ints_k delle variabili simboliche.

Dopo aver delineato gli operatori insiemistici, è necessario esaminare due operatori fondamentali che caratterizzano le strutture dati a stack: `push` e `pop`.

La funzione astratta $\mathsf{push} : \mathsf{St}^{\#}_{k,h} \times \mathsf{Ints}_k \to \mathsf{St}^{\#}_{k,h}$ inserisce un insieme di dimensione k in cima allo stack astratto, mantenendo la capacità di tracciare

solo i primi h elementi. Supponiamo di avere uno stack $\mathbb{st} = [s_0, s_1, \ldots, s_{h-1}] \in \mathsf{St}^{\#}_{k,h}$ con $s \in \mathsf{Ints}_k$. Allora la funzione push può essere definita come segue:

$$\mathsf{push}(\mathbb{st}, s) \triangleq [s_1, s_2, \ldots, s_{h-1}, s].$$

In pratica, quando inseriamo l'elemento s in $\mathbb{st}$, *spostiamo verso il basso* (ovvero a sinistra) tutti gli elementi dello stack, rimuovendo l'elemento più in basso (quello più a sinistra) s_0 e aggiungendo il nuovo elemento s nella posizione più alta (ovvero più a destra).

Ad esempio, nella Figura 5.1b è mostrato il risultato dell'esecuzione astratta dell'operazione `PUSH1 0x06` partendo dallo stack astratto della Figura 5.1a. È importante notare che tutti gli stack concreti, approssimati dallo stack astratto nella Figura 5.1a, sono noti per avere una dimensione massima di 3, poiché l'elemento di profondità 4 è $\varnothing$; al contrario, lo stack astratto nella Figura 5.1b descrive un insieme di stack concreti di dimensione arbitraria[3].

In modo simile, la funzione astratta $\mathsf{pop} : \mathsf{St}^{\#}_{k,h} \to \mathsf{St}^{\#}_{k,h}$ rimuove un elemento dalla cima di uno stack $\mathbb{st} = [s_0, s_1, \ldots, s_{h-2}, s_{h-1}] \in \mathsf{St}^{\#}_{k,h}$, e può essere definita come segue:

$$\mathsf{pop}(\mathbb{st}) \triangleq \begin{cases} \bot_{\mathsf{St}^{\#}_{k,h}} & \text{se } s_{h-1} = \varnothing; \\ [\varnothing, s_0, s_1, s_2, \ldots, s_{h-2}] & \text{se } s_0 = \varnothing; \\ [\top_{\mathsf{Ints}_k}, s_0, s_1, s_2, \ldots, s_{h-2}] & \text{altrimenti.} \end{cases}$$

In modo intuitivo, quando preleviamo l'elemento più in alto s_{h-1} dallo stack $\mathbb{st}$, ci troviamo di fronte a tre scenari:

- Se $s_{h-1} = \varnothing$, lo stack era vuoto e siamo di fronte ad un'eccezione nota come *stack underflow*, che viene rappresentata come $\bot_{\mathsf{St}^{\#}_{k,h}}$.
- Se $s_0 = \varnothing$, ossia se lo stack aveva una dimensione inferiore a h, spostiamo tutti gli altri elementi verso l'alto (cioè verso destra) e riempiamo la posizione vuota in basso (la più a sinistra) con $\varnothing$.
- Se $s_0 \neq \varnothing$, manteniamo la dimensione dello stack e inseriamo $\top_{\mathsf{Ints}_k}$ come nuovo elemento in fondo allo stack.

Per semplificare, lo stack astratto nella Figura 5.1c viene ottenuto estraendo un elemento dallo stack astratto rappresentato nella Figura 5.1b.

[3]La dimensione massima di qualsiasi stack EVM concreto è 1024 e l'esecuzione del programma si interrompe eccezionalmente se lo stack supera questo limite. Noi consideriamo ovviamente il caso con $h < 1024$.

5.3 Implementazione in EVMLiSA

Dopo aver delineato i concetti matematici e le strutture dati che saranno al centro della nostra analisi, procediamo ora ad esaminare come tradurre tali concetti in implementazioni pratiche in EVMLiSA.

5.3.1 Classe KIntegerSet

La classe `KIntegerSet` rappresenta il dominio delle variabili astratte Ints_k, come discusso nella Sezione 5.1. All'interno del codice mostrato nell'Algoritmo 12, si evidenzia che questa classe estende la classe `SetLattice`, la quale è un reticolo generico di insiemi che contiene un insieme di elementi. Di conseguenza, la classe `KIntegerSet` eredita numerosi metodi utili, inclusi il costruttore, lub e glb (righe 12–25). Successivamente, sono stati implementati i metodi necessari per definire $\top_{\mathsf{Ints}_k}$ (top) e $\varnothing$ (bottom). Infine, sono stati definiti tutti i metodi essenziali per eseguire le operazioni fondamentali, come l'addizione, la sottrazione, la moltiplicazione, e così via, come mostrato nelle righe 27–50.

5.3.2 Classe AbstractStack

La classe `AbstractStack`, invece, rappresenta il dominio degli stack astratti di dimensione h, ossia $\mathsf{St}^{\#}_{k,h}$. Come si può notare nel codice dell'Algoritmo 13, la dimensione dello stack astratto h viene definita tramite la variabile `h`, la quale di default è settata a 32 (riga 3). La struttura fisica dello stack è realizzata tramite una `LinkedList` di oggetti `KIntegerSet`. All'interno del codice (righe 4–5) sono state implementate anche le definizioni di $\top_{\mathsf{St}^{\#}_{k,h}}$ (top) e $\bot_{\mathsf{St}^{\#}_{k,h}}$ (bottom). Inoltre, sono state incluse le operazioni di push e pop (righe 9–20), come discusso nella Sezione 5.2. Infine, nelle righe 23–49, sono state implementate le operazioni $\sqcup_{\mathsf{St}^{\#}_{k,h}}$ (lub) e $\sqcap_{\mathsf{St}^{\#}_{k,h}}$ (glb), le quali effettuano operazioni elemento per elemento, proprio come descritto nel dettaglio sempre nella Sezione 5.2.

Algoritmo 12 Frammento di codice della classe `KIntegerSet`.

```
public class KIntegerSet extends SetLattice<KIntegerSet, Number> {
    public static final KIntegerSet NUMERIC_TOP
        = new KIntegerSet(Collections.emptySet(), true);
    public static final KIntegerSet NOT_JUMPDEST_TOP = new KIntegerSet(-10);
    public static final KIntegerSet BOTTOM
        = new KIntegerSet(Collections.emptySet(), false);
    // ...
    public KIntegerSet(Set<Number> elements, boolean isTop) {
        super(elements, isTop);
    }
    // ...
    @Override
    public KIntegerSet lubAux(KIntegerSet other) throws SemanticException {
        if (isTopNotJumpdest())
            return this;
        else if (other.isTopNotJumpdest())
            return other;
        KIntegerSet result = super.lubAux(other);
        return result.size() > K ? top() : result;
    }
    @Override
    public KIntegerSet glbAux(KIntegerSet other) throws SemanticException {
        KIntegerSet result = super.glbAux(other);
        return result.size() > K ? top() : result;
    }
    // ...
    public KIntegerSet sum(KIntegerSet other) {
        // Controlli vari
        Set<Number> elements = new HashSet<>(K);
        for (Number i : this.elements)
            for (Number j : other.elements) {
                Number add = i.add(j);
                if (add.compareTo(MAX) >= 0)
                    add = add.subtract(MAX);
                elements.add(add);
            }
        return new KIntegerSet(elements);
    }
    public KIntegerSet sub(KIntegerSet other) {
        // Controlli vari
        Set<Number> elements = new HashSet<>(K);
        for (Number i : this.elements)
            for (Number j : other.elements) {
                Number sub = i.subtract(j);
                if (sub.compareTo(ZERO_INT) < 0)
                    sub = sub.add(MAX);
                elements.add(sub);
            }
        return new KIntegerSet(elements);
    }
    // ...
}
```

Algoritmo 13 Frammento di codice della classe `AbstractStack`.

```
1  public class AbstractStack implements ValueDomain<AbstractStack>, BaseLattice<
       AbstractStack> {
2
3      private static int h = 32;
4      private static final AbstractStack TOP = new AbstractStack(new LinkedList<
       KIntegerSet>(Collections.nCopies(STACK_LIMIT, KIntegerSet.NUMERIC_TOP)));
5      private static final AbstractStack BOTTOM = new AbstractStack(null);
6      private final LinkedList<KIntegerSet> stack;
7      // ...
8
9      public void push(KIntegerSet target) {
10         stack.addLast(target);
11         stack.removeFirst();
12     }
13     public KIntegerSet pop() {
14         KIntegerSet result = stack.removeLast();
15         if (!stack.getFirst().isTop())
16             stack.addFirst(KIntegerSet.BOTTOM);
17         else
18             stack.addFirst(KIntegerSet.NUMERIC_TOP);
19         return result;
20     }
21     // ...
22
23     @Override
24     public AbstractStack lubAux(AbstractStack other) throws SemanticException
       {
25         LinkedList<KIntegerSet> result = new LinkedList<>();
26         Iterator<KIntegerSet> thisIterator = this.stack.iterator();
27         Iterator<KIntegerSet> otherIterator = other.stack.iterator();
28
29         while (thisIterator.hasNext() && otherIterator.hasNext()) {
30             KIntegerSet thisElement = thisIterator.next();
31             KIntegerSet otherElement = otherIterator.next();
32             result.addLast(thisElement.lub(otherElement));
33         }
34         return new AbstractStack(result);
35     }
36
37     @Override
38     public AbstractStack glbAux(AbstractStack other) throws SemanticException
       {
39         LinkedList<KIntegerSet> result = new LinkedList<>();
40         Iterator<KIntegerSet> thisIterator = this.stack.iterator();
41         Iterator<KIntegerSet> otherIterator = other.stack.iterator();
42
43         while (thisIterator.hasNext() && otherIterator.hasNext()) {
44             KIntegerSet thisElement = thisIterator.next();
45             KIntegerSet otherElement = otherIterator.next();
46             result.addLast(thisElement.glb(otherElement));
47         }
48         return new AbstractStack(result);
49     }
50     // ...
51 }
```

Capitolo 6

Dominio degli insiemi degli stack astratti

Nelle sezioni precedenti abbiamo esaminato i programmi bytecode EVM utilizzando un singolo stack astratto di dimensioni h, contenente elementi del dominio Ints_k. Tuttavia, per affrontare in modo più efficace la risoluzione dei salti condizionali, abbiamo deciso di ampliare il dominio dello stack astratto precedente ad un insieme di stack astratti, noto come *set di stack astratti*.

Quando si verificano cicli nel codice sorgente, l'analisi precedentemente presentata raccoglie gli stack astratti tramite un'operazione di lub in un unico stack astratto. Tuttavia, potremmo perdere precisione quando gli elementi *top* dello stack astratto raccolto vengono uniti tramite l'operatore lub del dominio Ints_k, superando il limite di k. In tal caso, il risultato sarebbe $\top_{\mathsf{Ints}_k}$, con la conseguente perdita di informazioni sugli interi nell'insieme e sui possibili obiettivi di un'istruzione di salto condizionale.

Introduciamo il concetto del dominio di *set di stack astratti* $\mathsf{SetSt}^{\#}_{k,h,l}$, costituito da insiemi di stack astratti con al massimo l elementi (con un elemento speciale $\top_{\mathsf{SetSt}^{\#}_{k,h,l}}$ che denota l'elemento *top*), la cui altezza è al massimo h e che contengono insiemi di interi con al massimo k elementi, e che può essere definito nel seguente modo:

$$\mathsf{SetSt}^{\#}_{k,h,l} \triangleq \langle \wp_{\leq l}(\mathcal{S}_{\mathsf{Ints}_k,h}) \cup \{\top_{\mathsf{SetSt}^{\#}_{k,h,l}}\}, \sqcup_{\mathsf{SetSt}^{\#}_{k,h,l}}, \sqcap_{\mathsf{SetSt}^{\#}_{k,h,l}}, \top_{\mathsf{SetSt}^{\#}_{k,h,l}}, \varnothing \rangle.$$

Le operazioni di *lub* e *glb* corrispondono all'unione e all'intersezione degli insiemi, rispettivamente, e quando la dimensione del risultato supera l, viene restituito $\top_{\mathsf{SetSt}^{\#}_{k,h,l}}$ (*top*).

Pertanto, possiamo adattare l'algoritmo di risoluzione delle jump (Algoritmo 11) per utilizzare il nuovo concetto di *insieme di stack astratti*. Il nuovo

pseudocodice è mostrato nell'Algoritmo 14, che funziona in modo simile all'Algoritmo 11, con la differenza che il risultato dell'analisi (riga 11) restituisce anche un insieme di stack astratti, e le righe 15–21 vengono applicate per ogni stack astratto contenuto nell'insieme.

Algoritmo 14 Pseudocodice dell'algoritmo per la risoluzione delle jump aggiornato.

```
 1: function buildCFG(𝒫)
 2:     𝒢 ← CFG(𝒫)
 3:     let h, k ∈ ℕ ∖ {0};
 4:     do
 5:         fp ← jumpSolver(𝒢, h, k)
 6:     while fp;
 7:     return 𝒢;
 8:
 9: function jumpSolver(𝒢, h, k)
10:     let 𝒢 = ⟨N, E⟩;
11:     𝒜 ← runAnalysis(𝒢, h, k);
12:     StackSet ← getStacks(𝒜)
13:     fp ← false;
14:     for all 𝒮_in, 𝒮_out ∈ StackSet do
15:         for all (𝒮_in, op, 𝒮_out) ∈ 𝒜 do
16:             if op in (JUMP, JUMPI) then
17:                 s ← top(𝒮_in);
18:                 if s ≠ ⊤_Ints_k then
19:                     for all pc ∈ s do
20:                         E ← E ∪ {op → stmt(pc)};
21:                     fp ← true;
22:     return fp;
```

6.1 AbstractStackSet in EVMLiSA

L'implementazione del *set di stack astratti* $\mathsf{SetSt}^{\#}_{k,h,l}$ in EVMLiSA è piuttosto semplice. La classe `AbstractStackSet` può essere vista essenzialmente come un insieme di oggetti `AbstractStack`. Un frammento di questa implementazione è illustrato nell'Algoritmo 15. Come già visto nelle classi precedenti, troviamo anche qui le costanti `TOP` e `BOTTOM` (righe 3–7), oltre alla costante `SIZE` che indica la dimensione del set di `AbstractStack`.

I costruttori della classe `AbstractStackSet` sfruttano i costruttori della classe estesa `SetLattice`, proprio come nella classe `KIntegerSet`. Infine, il metodo `add` (righe 18–21) viene utilizzato per aggiungere un oggetto `AbstractStack` all'insieme, a condizione che non sia *bottom*.

Algoritmo 15 Frammento di codice della classe `AbstractStackSet`.

```
1  public class AbstractStackSet extends SetLattice<AbstractStackSet,
       AbstractStack> {
2
3      private static int SIZE = 32;
4      private static final AbstractStackSet BOTTOM
5          = new AbstractStackSet(null, false);
6      private static final AbstractStackSet TOP
7          = new AbstractStackSet(Collections.emptySet(), true);
8
9      public AbstractStackSet() {
10         super(new HashSet<AbstractStack>(), false);
11         this.elements.add(new AbstractStack());
12     }
13
14     public AbstractStackSet(Set<AbstractStack> elements, boolean isTop) {
15         super(elements, isTop);
16     }
17
18     public void add(AbstractStack other) {
19         if (!other.isBottom())
20         this.elements.add(other);
21     }
22
23     // ...
24
25     @Override
26     public AbstractStackSet lubAux(AbstractStackSet other) throws
       SemanticException {
27         AbstractStackSet lubAux = super.lubAux(other);
28         if (lubAux.size() > SIZE)
29             return TOP;
30         return lubAux;
31     }
32
33     // ...
34 }
```

Capitolo 7

EVMAbstractState

Nel contesto del codice Java di EVMLiSA, è stata implementata la classe `EVMAbstractState`, che implementa l'interfaccia `ValueDomain` di LiSA. Secondo la documentazione di LiSA[1], redatta da Luca Negrini, questa classe rappresenta un dominio semantico per valutare la semantica degli statement che agiscono sui valori anziché sulle posizioni di memoria.

La classe `EVMAbstractState` presenta diversi attributi. Oltre alle costanti `TOP` e `BOTTOM`, come nelle altre classi, sono presenti uno stack astratto (già discusso nel capitolo precedente), e come menzionato nella Sezione 2.4 dedicata all'EVM, una memoria volatile (`Memory`) e uno spazio di archiviazione permanente (`Storage`). Quest'ultimi verranno approfonditi nella Sezione 7.1.

Il frammento di codice nell'Algoritmo 16 offre una rappresentazione di queste componenti, mostrando come sono implementate all'interno della classe `EVMAbstractState`.

[1] `https://github.com/lisa-analyzer/lisa`

Algoritmo 16 Frammento di codice della classe `EVMAbstractState`.

```
1  public class EVMAbstractState implements ValueDomain<EVMAbstractState>,
       BaseLattice<EVMAbstractState> {
2
3      private static final EVMAbstractState TOP = new EVMAbstractState(true, "")
       ;
4      private static final EVMAbstractState BOTTOM = new EVMAbstractState(new
       AbstractStackSet().bottom(), new Memory().bottom(), new Memory().bottom(),
        KIntegerSet.BOTTOM);
5
6      private final boolean isTop;
7      private static String CONTRACT_ADDRESS;
8      private final AbstractStackSet stacks;
9      private final Memory memory;
10     private final KIntegerSet mu_i;
11     private final Memory storage;
12
13     // Costruttori
14     private EVMAbstractState(boolean isTop, String contractAddress) {
15         this.isTop = isTop;
16         this.stacks = new AbstractStackSet();
17         this.memory = new Memory();
18         this.storage = new Memory();
19         this.mu_i = KIntegerSet.ZERO;
20         CONTRACT_ADDRESS = contractAddress;
21     }
22
23     public EVMAbstractState(AbstractStackSet stacks, Memory memory, Memory
       storage, KIntegerSet mu_i) {
24         this.isTop = false;
25         this.stacks = stacks;
26         this.memory = memory;
27         this.storage = storage;
28         this.mu_i = mu_i;
29     }
30     // ...
31 }
```

7.1 Memory e Storage

In EVMLiSA, sia la memoria che lo storage sono gestiti attraverso la classe `Memory`. Quest'ultima è essenzialmente una mappa chiave-valore in cui entrambi i parametri sono oggetti di tipo `KIntegerSet`. Approfondendo il contesto di Ethereum, la memoria rappresenta la memoria volatile nell'architettura, i cui dati non sono persistenti attraverso la blockchain. È una struttura dati ad accesso casuale utilizzata per conservare dati temporanei durante l'esecuzione di uno smart contract.

D'altra parte, lo storage è uno spazio non volatile che serve per memorizzare dati tra le chiamate di funzione. È destinato a conservare informazioni che devono persistere oltre la singola esecuzione dello smart contract, come variabili e strutture dati. Analogamente alla memoria, lo storage è implemen-

tato come una struttura chiave-valore, comunemente rappresentata come una mappa.

7.2 Small-step semantics

Dopo aver delineato le strutture dati astratte necessarie per simulare l'esecuzione di uno smart contract su Ethereum, è essenziale definire la semantica astratta delle diverse operazioni bytecode. Ciò comporta stabilire le conseguenze che avranno gli opcode di interesse sulle strutture dati astratte.

Per affrontare questo compito, facciamo affidamento sul metodo denominato `smallStepSemantics`. Secondo quanto riportato nella documentazione di LiSA, questo metodo restituisce una copia del dominio utilizzato, opportunamente modificato in base alla semantica dell'espressione data. Prende come argomenti una `ValueExpression`, che rappresenta l'espressione da valutare, e un `ProgramPoint`, che indica il punto del programma in cui si trova l'istruzione da valutare.

Nel contesto di EVMLiSA, sono stati gestiti un numero significativo di opcode, oltre 140. Il metodo `smallStepSemantics`, dopo aver effettuato le necessarie verifiche, è strutturato in uno switch-case in cui viene valutata la `ValueExpression`. Una parte dell'implementazione di `smallStepSemantics` è mostrata nell'Algoritmo 17. È da notare come l'algoritmo gestisce la possibile propagazione del valore `BOTTOM` nel caso di un percorso di esecuzione irraggiungibile. Inoltre, il metodo controlla se l'espressione è una costante (`Constant`) e, in tal caso, restituisce `this`, poiché le costanti non modificano lo stack. Infine, se non è stato possibile valutare l'istruzione, viene restituito `TOP`.

Successivamente, esamineremo alcuni esempi relativi al modo in cui sono state gestite le operazioni degli operatori.

Algoritmo 17 Frammento di codice della `smallStepSemantics`.

```
@Override
public EVMAbstractState smallStepSemantics(ValueExpression expression,
    ProgramPoint pp) throws SemanticException {
    // bottom state is propagated
    if (this.isBottom())
        return this;
    if (expression instanceof Constant) {
        return this;
    } else if (expression instanceof UnaryExpression) {
        UnaryExpression un = (UnaryExpression) expression;
        UnaryOperator op = un.getOperator();
        if (op != null) {
            switch (op.getClass().getSimpleName()) {
                case "PushOperator": { // PUSH
                    // ...
                }
                case "AddOperator": { // ADD
                    // ...
                }
                case "SubOperator": { // SUB
                    // ...
                }
                // ...
            }
        }
    }
    return top();
}
```

7.2.1 Esempio di operazioni astratte

Ora esamineremo due esempi di implementazione (Algoritmo 18): gli operatori `PushOperator` e `AddOperator`.

L'implementazione astratta dell'opcode `PushOperator`, che corrisponde all'istruzione `PUSH`, gestisce l'inserimento di un valore nello stack. Innanzitutto, viene creato un nuovo `AbstractStackSet`, che rappresenta l'insieme degli stack astratti, inizializzato con una dimensione uguale al numero di stack già presenti. Successivamente, viene creato un oggetto `KIntegerSet`, che contiene il valore da inserire nello stack e viene ottenuto dall'espressione numerica specificata nell'istruzione `PUSH`.

In seguito, per ogni stack astratto presente, viene creato un clone dello stack originale. Il valore appena ottenuto viene quindi inserito in cima a ciascuno di questi stack clonati utilizzando il metodo `push()`. Ogni stack risultante, contenente il valore inserito, viene aggiunto all'oggetto `AbstractStackSet` creato in precedenza.

Infine, viene restituito un nuovo stato astratto `EVMAbstractState`, il cui stack è stato aggiornato con il valore inserito, mentre la memoria e lo storage

rimangono invariati.

L'astrazione dell'opcode `AddOperator` segue un approccio analogo. Anch'esso coinvolge la creazione di un nuovo oggetto `AbstractStackSet`, seguito dalla scansione di tutti gli stack contenuti nell'insieme astratto degli stack. Per ciascuno di essi, viene creata una copia. Subito dopo, estraiamo i due elementi in cima allo stack clonato per eseguire l'operazione di somma, inserendo il risultato ottenuto in cima allo stack. Successivamente, i nuovi stack vengono aggiunti al nuovo oggetto `AbstractStackSet` creato in precedenza, che viene infine restituito.

Algoritmo 18 Implementazione della semantica di `PUSH` e `ADD`.

```
case "PushOperator": { // PUSH
    AbstractStackSet result = new AbstractStackSet(new HashSet<>(stacks.size()
    ), false);
    KIntegerSet toPush = new KIntegerSet(toBigInteger(un.getExpression()));
    for (AbstractStack stack : stacks) {
        AbstractStack resultStack = stack.clone();
        resultStack.push(toPush);
        result.add(resultStack);
    }
    return new EVMAbstractState(result, memory, storage, mu_i);
}
case "AddOperator": { // ADD
    AbstractStackSet result = new AbstractStackSet(new HashSet<>(stacks.size()
    ), false);
    for (AbstractStack stack : stacks) {
        AbstractStack resultStack = stack.clone();
        KIntegerSet opnd1 = resultStack.pop();
        KIntegerSet opnd2 = resultStack.pop();

        resultStack.push(opnd1.sum(opnd2));
        result.add(resultStack);
    }
    return new EVMAbstractState(result, memory, storage, mu_i);
}
```

7.2.2 Top numerico e top non jumpdest

Va notato che in questo contesto non è presente un'unica rappresentazione di *top*, come invece riscontrato nelle altre classi astratte. In questa situazione, la decisione è stata quella di suddividere la semantica di *top* in due categorie: *top numerici* e *top not jumpdest*.

Entrambi sono utilizzati per gestire le situazioni in cui non è possibile valutare l'istruzione, ma differiscono nel fatto che nei *top numerici*, indicati dalla costante `NUMERIC_TOP`, un'istruzione potrebbe restituire una destinazione valida di salto, mentre i *top not jumpdest*, rappresentati con la costante

`NOT_JUMPDEST_TOP`, restituiscono valori che non possono rappresentare una destinazione valida di salto. Un esempio chiarificatore è dato dall'opcode `TIMESTAMP`, il quale restituisce il timestamp attuale e inserisce questo valore in cima allo stack. È evidente che il valore appena inserito non può essere una destinazione di salto valida.

Questa distinzione è essenziale per comprendere quali *top* sono rilevanti per la risoluzione delle jump. L'implementazione dei due *top* si può osservare alle righe 3–4 dell'Algoritmo 12.

7.3 Least upper bound

Il *least upper bound* (*lub*), noto anche come *limite superiore minimo*, è un operatore commutativo nella semantica astratta. Come già spiegato nella sezione 5.1, esso consente di determinare il minimo limite superiore tra due elementi di un dominio astratto. Nel contesto dell'`EVMAbstractState`, il metodo *lub* sfrutta le proprietà degli oggetti astratti al suo interno. In pratica, il calcolo del *lub* avviene elemento per elemento, applicando il *lub* su ciascun elemento individuale e restituendo i risultati attraverso un nuovo stato astratto dell'EVM, rappresentato dall'oggetto `EVMAbstractState`. Questo nuovo stato conterrà i risultati ottenuti dal calcolo del *lub* su ciascun elemento del dominio astratto.

Algoritmo 19 Implementazione del *lub* nella classe `EVMAbstractState`.

```
1 @Override
2 public EVMAbstractState lubAux(EVMAbstractState other) throws
      SemanticException {
3     return new EVMAbstractState(
4         stacks.lubAux(other.stacks),
5         memory.lub(other.getMemory()),
6         storage.lub(other.storage),
7         mu_i.lub(other.getMu_i())
8     );
9 }
```

7.4 Greatest lower bound

Il *greatest lower bound* (*glb*), noto anche come *limite inferiore massimo*, è un altro operatore commutativo fondamentale nella semantica astratta. Questo operatore restituisce l'intervallo più grande che contiene tutti gli elementi comuni di due insiemi appartenenti a un dominio astratto. Analogamente all'implementazione del *lub*, anche per il *glb* sfruttiamo le proprietà degli oggetti astratti all'interno dell'`EVMAbstractState`. Il calcolo del *glb* avviene elemento per elemento, applicando il *glb* su ciascun elemento individuale e restituendo i risultati attraverso un nuovo stato astratto dell'EVM.

Algoritmo 20 Implementazione del *glb* nella classe `EVMAbstractState`.

```
1 @Override
2 public EVMAbstractState glbAux(EVMAbstractState other) throws
      SemanticException {
3     return new EVMAbstractState(
4         stacks.glbAux(other.stacks),
5         memory.glb(other.getMemory()),
6         storage.glb(other.storage),
7         mu_i.glb(other.getMu_i())
8     );
9 }
```

Capitolo 8

Checker Semantico

Come discusso nella Sezione 7.2 dedicata alla semantica astratta degli operatori, il metodo `smallStepSemantics` è essenziale per stabilire lo stato astratto dell'EVM in ogni nodo del CFG, fornendo così una visione dettagliata dello stato del programma ad ogni punto di esecuzione. Tuttavia, una singola iterazione di questo metodo non è sufficiente per completare e correggere il CFG per l'intero programma. Questo perché le istruzioni di salto condizionato non vengono risolte in questa fase preliminare.

Pertanto, è necessario un approccio iterativo per risolvere tali istruzioni di salto. Come evidenziato nello Pseudocodice 14, il processo di risoluzione dei salti è iterativo e si conclude solo quando non è più possibile risolvere alcuna istruzione di salto.

Da ciò emerge chiaramente la necessità di un metodo aggiuntivo, oltre a `smallStepSemantics`, che analizzi semanticamente il CFG generato, tenendo conto degli stati vari dell'EVM in ogni punto del programma e sia in grado di risolvere le istruzioni di salto per garantire un'analisi accurata e completa del comportamento del programma.

8.1 La classe `JumpSolver`

La classe `JumpSolver` svolge un ruolo fondamentale nell'analisi dello stato dell'EVM in ogni punto del programma e nella risoluzione delle istruzioni di salto. Utilizzando il metodo `visit()`, essa esamina il grafo del programma nodo per nodo, permettendo di analizzare gli stati associati a ciascun punto del programma. Questo metodo, dato che sovrascrive il metodo `GraphVisitor` di LiSA, viene invocato durante la visita del CFG e restituisce `true` se la visita può continuare, `false` altrimenti.

Al metodo vengono passati come argomenti un *tool* (lo strumento di controllo semantico che esegue questa analisi), un *grafo* (di tipo `EVMCFG`) e un

nodo (di tipo `Statement`). Durante l'esecuzione, il metodo visita tutti i nodi del grafo, ma concentra la sua attenzione sulle istruzioni di salto (`JUMP` e `JUMPI`).

All'interno del metodo `visit()`, si trova un *for-each* principale che itera sui risultati dell'analisi semantica del tool relativi al CFG. Questi risultati vengono esaminati poiché potrebbero esserci più analisi diverse eseguite sullo stesso CFG, ma nostro caso ce ne sarà solo una. L'obiettivo è ottenere lo stato astratto dell'EVM corrispondente al nodo attuale.

Se lo stato astratto è `BOTTOM`, ciò indica che stiamo seguendo un flusso di esecuzione irraggiungibile, e di conseguenza la jump corrente è irraggiungibile. Se lo stato astratto è `TOP`, significa che stiamo seguendo un flusso di esecuzione percorribile, ma lo stato astratto è indeterminabile, e quindi potremmo non risolvere la jump.

Nell'Algoritmo 21 è possibile osservare l'implementazione in EVMLiSA di ciò che è stato descritto finora.

Algoritmo 21 Frammento del metodo `visit()`: ciclo principale.

```
// ...
this.cfgToAnalyze = (EVMCFG) graph;

if (this.jumpDestinations == null)
    this.jumpDestinations = this.cfgToAnalyze.getAllJumpdest();

if (!(node instanceof Jump) && !(node instanceof Jumpi))
    return true;
else if (cfgToAnalyze.getAllPushedJumps().contains(node))
    return true;

for (AnalyzedCFG< ... > result : tool.getResultOf(this.cfgToAnalyze)) {
    AnalysisState< ... > analysisResult = null;

    try {
        analysisResult = result.getAnalysisStateBefore(node);
    } catch (SemanticException e1) {
        e1.printStackTrace();
    }

    EVMAbstractState valueState = analysisResult.getState().getValueState();

    if (valueState.isBottom()) {
        System.err.println("Unreachable jump");
        continue;
    } else if (valueState.isTop()) {
        System.err.println("Maybe unsolved jump");
        continue;
    }

    // ...
}
```

Se invece lo stato non è né `TOP` né `BOTTOM`, tentiamo di risolvere la jump. In tal caso, eseguiamo un altro *for-each* che scorre tutti gli insiemi di interi `KIntegerSet` presenti in cima agli `AbstractStack` dell'`AbstractStackSet`. Prima di tutto, applichiamo un filtro per selezionare le `JUMPDEST` potenzialmente raggiungibili dal nodo corrente. Se nessuna destinazione viene selezionata, significa che non è possibile risolvere la jump e passiamo alla prossima iterazione del ciclo.

Altrimenti, per ciascuna destinazione ottenuta, controlliamo se esiste già un arco che collega il nodo attuale a quella destinazione. Se non esiste, creiamo un nuovo arco (`SequentialEdge` se si tratta di una `JUMP` e `TrueEdge` se si tratta di una `JUMPI`) per collegare i due nodi.

Se viene aggiunto un nuovo arco, è essenziale impostare il flag `fixpoint` su `false`. Questo perché l'aggiunta di un arco comporta una modifica del CFG, rendendo necessario un nuovo ciclo di analisi completa del grafo.

Nell'Algoritmo 22 è mostrato il ciclo interno con il filtro sulle `JUMPDEST`.

Algoritmo 22 Frammento del metodo `visit()`: ciclo interno con filtro.

```
// ...

for (KIntegerSet topStack : valueState.getTop()) {
    Set<Statement> filteredDests = this.jumpDestinations
    .stream()
    .filter(pc -> topStack.contains(new Number(((ProgramCounterLocation) pc.
    getLocation()).getPc())))
    .collect(Collectors.toSet());

    if (node instanceof Jump) { // JUMP
        for (Statement jmp : filteredDests) {
            SequentialEdge edge = new SequentialEdge(node, jmp);
            if (!this.cfgToAnalyze.containsEdge(edge)) {
                this.cfgToAnalyze.addEdge(edge);
                fixpoint = false;
            }
        }
    } else { // JUMPI
        for (Statement jmp : filteredDests) {
            TrueEdge edge = new TrueEdge(node, jmp);
            if (!this.cfgToAnalyze.containsEdge(edge)) {
                this.cfgToAnalyze.addEdge(edge);
                fixpoint = false;
            }
        }
    }
}

// ...
```

8.1.1 Il metodo `afterExecution()`

Il metodo `afterExecution()` all'interno di `JumpSolver` viene invocato da LiSA una volta completata l'analisi dell'intero CFG tramite la procedura `visit()`.

Come si può vedere dal frammento di codice mostrato nell'Algoritmo 23, inizialmente questo metodo controlla lo stato del flag `fixpoint`. Se il flag è impostato su `false`, viene impostato in `true` (e sarà nuovamente impostato su `false` se `visit()` dovesse risolvere un'altra jump). Successivamente, si crea una nuova istanza di LiSA con la stessa configurazione e la si esegue sull'oggetto `cfgToAnalyze`, il quale rappresenta il CFG aggiornato da `JumpSolver` dopo l'analisi delle jump tramite `visit()`. Questo passaggio permette di eseguire un'ulteriore analisi semantica completa del CFG aggiornato, la quale potrebbe risolvere altre jump [2].

Tuttavia, se il flag `fixpoint` è già impostato su `true`, significa che la visita del CFG ha raggiunto un punto fisso e non sono necessarie ulteriori analisi. In questo caso, il metodo procede direttamente al calcolo delle statistiche. Utilizzando specifiche strutture dati, `JumpSolver` classifica le jump in base a determinati criteri. Ad esempio, una jump viene contrassegnata come *unsound* se gli elementi in cima a tutti gli stack astratti in quel punto sono `TOP`, mentre se sono tutti `BOTTOM`, la jump viene etichettata come *unreachable* (irraggiungibile). Inoltre, vengono registrati tutti gli elementi in cima agli stack per ogni jump, i quali saranno successivamente utilizzati per il salvataggio delle statistiche alla fine dell'analisi.

Algoritmo 23 Frammento del metodo `afterExecution()` di `JumpSolver`.

```
if (fixpoint) {
    // ...
    if (valueState.isBottom()) {
        this.unreachableJumps.add(node);
    } else if (valueState.isTop()) {
        this.maybeUnsoundJumps.add(node);
    } else {
        // ...
        AbstractStackSet stacks = valueState.getStacks();
        for (AbstractStack stack : stacks) {
            stacksSize.add(Integer.valueOf(stack.size()));
            KIntegerSet topStack = stack.getTop();
            if (allNumericTop && !topStack.isTopNumeric())
                allNumericTop = false;
            if (allBottom && !topStack.isBottom())
                allBottom = false;
            stacksTop.add(topStack);
        }
        stacksSizePerJump.put(node, stacksSize);
        topStackValuesPerJump.put(node, stacksTop);

        if (allNumericTop)
            this.unsoundJumps.add(node);
        if (allBottom)
            this.unreachableJumps.add(node);
    }
} else {
    this.fixpoint = true;
    LiSAConfiguration conf = tool.getConfiguration();
    LiSA lisa = new LiSA(conf);
    // ...
    try {
        lisa.run(program);
    } catch (AnalysisException e) {
        e.printStackTrace();
    }
}
```

8.1.2 Il metodo `dumpStatistics()` della classe `EVMLiSA`

Ultimo ma non meno importante, il metodo `dumpStatistics()` presente nella classe `EVMLiSA` riveste un ruolo fondamentale nel processo di analisi dell'EVM bytecode e nella valutazione della correttezza del programma. Sebbene non sia direttamente contenuto nella classe `JumpSolver`, è strettamente collegato e dipende dai risultati ottenuti da essa. Questo metodo viene invocato alla fine dell'esecuzione del programma per stampare le statistiche finali.

Come mostrato nell'Algoritmo 24, il primo passo consiste nell'ottenere il CFG da analizzare utilizzando il metodo `getComputedCFG()` di `JumpSolver`. Successivamente, vengono recuperati gli insiemi di nodi di jump irraggiungibili e unsound tramite i metodi `getUnreachableJumps()` e `getUnsoundJumps()`

rispettivamente, sempre di `JumpSolver`.

Una volta fatto ciò, si procede iterando su tutti i nodi di jump presenti nel CFG attraverso un ciclo. Per ciascun nodo di jump, vengono eseguite le operazioni mostrate nelle righe 7–41:

1. Si verifica se il nodo di jump è raggiungibile dal punto di ingresso tramite il metodo `reachableFrom()` del CFG. Se il nodo di jump è stato risolto, ossia se ha almeno uno o due archi di uscita (a seconda se è una `JUMP` o `JUMPI`), viene incrementato il contatore delle jump risolte e si passa al nodo successivo.

2. Se la jump non è stata risolta, vengono eseguite ulteriori verifiche per classificarla in base alle sue caratteristiche:

   (a) Se la jump è raggiungibile ma è anche contenuta nell'insieme dei nodi di jump irraggiungibili (`unreachableJumpNodes`), viene incrementato il contatore delle jump sicuramente irraggiungibili (`definitelyUnreachable`).

   (b) Se la jump non è raggiungibile, viene incrementato il contatore delle jump potenzialmente irraggiungibili (`maybeUnreachable`).

   (c) Se gli elementi in cima agli stack sono tutti diversi da `NUMERIC_TOP`, la jump è classificata come `definitelyFakeMissedJump`.

   (d) Se la jump ha solo un singolo stack e il valore in cima è `NUMERIC_TOP`, la jump è considerato `unsound` e non risolta.

3. In tutti gli altri casi, la jump è classificata come `maybeFakeMissedJump`.

Algoritmo 24 Calcolo delle statistiche in `dumpStatistics()` di `EVMLiSA`.

```
EVMCFG cfg = checker.getComputedCFG();
Set<Statement> unreachableJumpNodes = checker.getUnreachableJumps();
Set<Statement> unsoundJumpNodes = checker.getUnsoundJumps();

Statement entryPoint = cfg.getEntrypoints().stream().findAny().get();
for (Statement jumpNode : cfg.getAllJumps()) {
    if ((jumpNode instanceof Jump) || (jumpNode instanceof Jumpi)) {

        boolean reachableFrom = cfg.reachableFrom(entryPoint, jumpNode);
        boolean skip = false;

        if (jumpNode instanceof Jump) {
            if (cfg.getOutgoingEdges(jumpNode).size() >= 1) {
                resolvedJumps++;
                skip = true;
            }
        } else if (jumpNode instanceof Jumpi) {
            if (cfg.getOutgoingEdges(jumpNode).size() >= 2) {
                resolvedJumps++;
                skip = true;
            }
        }
        if (!skip) {
            Set<KIntegerSet> topStackValuesPerJump = checker.
    getTopStackValuesPerJump(jumpNode);
            Set<Integer> stacksSizePerJump = checker.getStacksSizePerJump(
    jumpNode);

            if (reachableFrom && unreachableJumpNodes.contains(jumpNode))
                definitelyUnreachable++;
            else if (!reachableFrom)
                maybeUnreachable++;
            else if (topStackValuesPerJump == null) {
                maybeFakeMissedJumps++;
            } else if (!topStackValuesPerJump.contains(KIntegerSet.NUMERIC_TOP
    )) {
                definitelyFakeMissedJumps++;
            } else if (topStackValuesPerJump.contains(KIntegerSet.NUMERIC_TOP)
     && stacksSizePerJump.size() == 1) {
                notSolvedJumps++;
                unsoundJumps++;
            } else {
                maybeFakeMissedJumps++;
            }
        }
    }
    // ...
}
```

Capitolo 9

Discussione e valutazione sperimentale

In quest'ultimo capitolo verrà esaminata l'efficacia e le prestazioni di EVM-LiSA attraverso una serie di benchmark e analisi approfondite. Inoltre verrà condotto un confronto con altre soluzioni software simili presenti online.

9.1 Cosa è stato realizzato

Per riassumere le azioni svolte per la realizzazione di questo progetto, possiamo evidenziare le seguenti fasi salienti:

- È stato introdotto il dominio astratto denominato `AbstractStack`, costituito da un insieme finito di elementi `KIntegerSet` (Capitolo 5).
- Si è esteso il suddetto dominio `AbstractStack` ad un insieme più ampio `AbstractStackSet`, allo scopo di garantire una maggiore precisione nell'esecuzione dell'analisi (Capitolo 6).
- Sono stati modellati gli opcode pertinenti alla gestione della memoria e dello storage all'interno della Ethereum Virtual Machine (Capitolo 7).
- È stata introdotta una distinzione tra due versioni dell'operatore *top*: una denominata *top not jumpdest*, che identifica i valori non idonei a rappresentare una destinazione valida di salto, e una versione *numerica*, che invece può rappresentare una destinazione di salto valida (Capitolo 7).
- Si è proceduto all'aggiornamento della classe `JumpSolver`, integrando tutte le modifiche apportate durante il processo di sviluppo (Capitolo 8).

9.2 Benchmark

La nuova versione di EVMLiSA è stata sottoposta a un rigoroso processo di test, inizialmente confrontandola con il benchmark utilizzato nella tesi di Davide Tarpini [2] (Tabella 9.2), e successivamente su un insieme di 5000 smart contract raccolti casualmente da EtherScan[1] (Tabella 9.1). I risultati ottenuti da entrambi i benchmark dimostrano che EVMLiSA risolve il 100% delle jump in tutti i casi, confermando la sua capacità di costruire un CFG completo e affidabile.

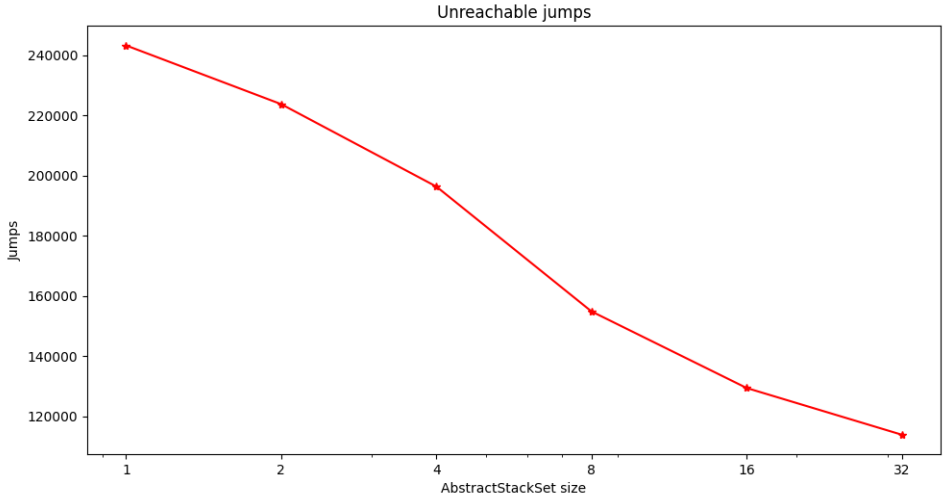
È importante sottolineare il ruolo cruciale svolto dall'implementazione di `AbstractStackSet`. Come evidenziato nella Tabella 9.1, senza l'implementazione di `AbstractStackSet` (simulabile con i dati della prima riga), la precisione dell'analisi sarebbe stata compromessa, e di conseguenza non sarebbe stato possibile risolvere tutte le jump. L'impatto della dimensione di `AbstractStackSet` è evidente anche nell'aumento della complessità del flusso di esecuzione, con la possibile creazione di percorsi inesistenti, come indicato dalle colonne *Definitely* e *Maybe Fake (Path)*.

Infine, l'analisi dei dati della Tabella 9.1 rivela una relazione diretta tra l'aumento della precisione di EVMLiSA, determinata dalla dimensione dall'insieme degli stack astratti `AbstractStackSet`, e la diminuzione delle jump irraggiungibili (Figura 9.1a). Tuttavia, va notato che un aumento della precisione comporta un notevole incremento dei tempi di esecuzione dell'analisi, come si può vedere graficamente nella Figura 9.1b.

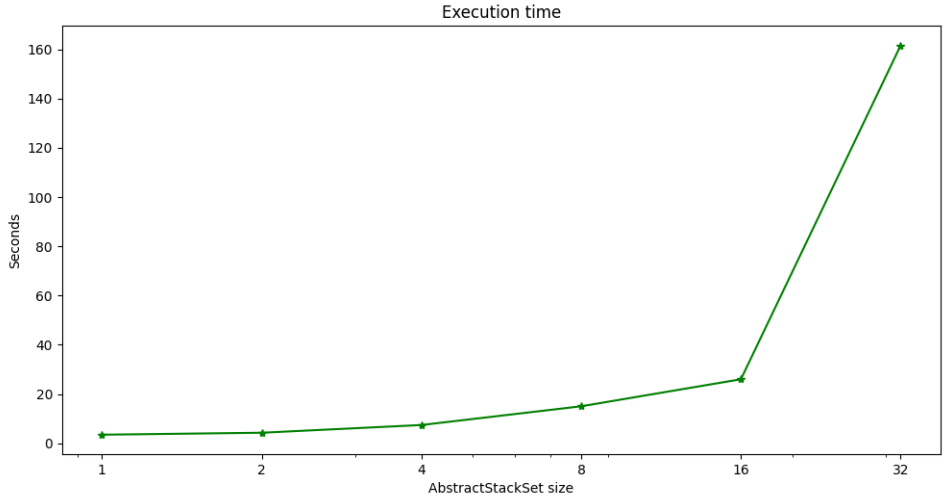
Dimensione StackSet	Jump Risolte	Jump Unsound	Jump Irragg.	Maybe Unsound	Definitely Fake	Maybe Fake	% Jump Solved	Time (sec)
1	1728979	1	243285	315	851	48	99.9999%	3.44
2	1728688	6	223758	333	1125	60	99.9997%	4.24
4	1727825	20	196366	421	1950	84	99.9988%	7.38
8	1726589	15	154845	482	3089	186	99.9991%	14.99
16	1727152	18	129387	520	2321	367	99.9990%	25.97
32	1728251	0	113854	479	1491	137	100%	161.65

Tabella 9.1: Benchmark sui 5000 smart contract al variare della dimensione di `AbstractStackSet`.

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



(a) Jump irraggiungibili al variare della dimensione di `AbstractStackSet`.



(b) Tempo di esecuzione di un'analisi al variare della dimensione di `AbstractStackSet`.

Figura 9.1: Rappresentazione grafica dei dati della Tabella 9.1.

	Indirizzo del contratto	Opcode totali	Jump totali	% Jump Risolte (prima)	% Jump Risolte (adesso)
1	0xfDb878237E95032CfFC084ebd720c63C3a32C662	9352	774	75.84%	100%
2	0xC4a21c88C3fA5654F51a2975494b752557DDaC2c	7504	587	87.73%	100%
3	0xdabb0c3f9a190b6fe4df6cb412ba66c3dd3e2ad1	8772	539	57.14%	100%
4	0x576501abd98ce5472b03b7ab4f5980941db7ef37	6801	442	91.40%	100%
5	0x251f752b85a9f7e1b3c42d802715b5d7a8da3165	4644	369	75.07%	100%
6	0xcc2ba2eac448d60e0f943ebe378f409cb7d1b58a	7363	336	89.58%	100%
7	0x0f4f45f2edba03d4590bd27cf4fd62e91a2a2d6a	5364	322	91.30%	100%
8	0x61CEAc48136d6782DBD83c09f51E23514D12470a	7016	298	77.52%	100%
9	0xeC31e65e1cBdc22B1dfBC3F04a53db5505F6A9C0	3890	290	85.17%	100%
10	0x0bb4f50d431942986941a68721fe8b8f0b9fe051	6208	271	88.56%	100%
11	0x49d743dc7f0aafda02a736ecdcce174e9cae6261	5188	241	77.59%	100%
12	0x50668356cadfb5cae006c06ed16bcffc11524218	4478	225	79.11%	100%
13	0x46d4FDfeAA126356D7196d418178D925835Ce3C0	6467	522	73.18%	100%
14	0x5facec6e543e1a4acf46f0dbe25d85e6cc48eaec	5207	198	76.26%	100%
15	0x1d9bb9efc11ce34cb16751ab35ba5a7a0a5e1a4a	4825	196	76.53%	100%
16	0x4c8563be01a48f4e8520fba68f826010abacb6c6	3852	187	82.35%	100%
17	0x5dcc6f5dc918fc399db0cb6cf3d160ae3bfb0c2f	3887	176	78.41%	100%
18	0x1d5ad987b743eb624662fe5c62b8f6015554203a	1988	173	78.03%	100%
19	0x6D1f367390F1c9E2Da6d5d7ec12D66e2eD09420d	1899	156	87.82%	100%
20	0x72a40175f5f24cf3933c37c19ffe82dd1417e33c	3428	150	86.67%	100%
21	0x732eBfefFDF57513f167b2d3D384E13246f60034	1780	142	88.03%	100%
22	0x083D41d6DD21EE938f0c055CA4fb12268DF0EfaC	2209	134	87.31%	100%
23	0x1964f2f3ce45ac518b18ef4aa4265f8aadcef4ae	2961	131	73.28%	100%
24	0x541d676940Fd68f2755c1585ec06a6f4857cf193	2862	118	83.90%	100%
25	0x24d6193066d21d36938dc336322c64034af202d9	2683	117	83.76%	100%
26	0x3668e5b02640dba32edb7fd6360f80f2b287640e	2833	115	77.39%	100%
27	0x8313675d1405f3f7aee3da9d63e0bf5c30c75832	2518	115	92.17%	100%
28	0x1dd80016e3d4ae146ee2ebb484e8edd92dacc4ce	2632	112	76.79%	100%
29	0x6d1d1da8145c26f37e032179e3434e293c3aea51	1685	101	85.15%	100%
30	0x6736077ae034e16f7fafd2d5ed1358370fda1f88	2746	93	84.95%	100%
31	0x6e08427f6f472342d0ce286c875956be232d6af4	2128	93	89.25%	100%
32	0x31179edfd7ef37b16c7643407321466cd0b33b53	1413	86	75.58%	100%
33	0x5cef8c37ab75fc56cd50e17692f39ac7bc189075	2067	80	87.50%	100%
34	0x679131f591b4f369acb8cd8c51e68596806c3916	1450	77	77.92%	100%
35	0x000000000d38df53b45c5733c7b34000de0bdf52	1704	76	85.53%	100%
36	0x0122db5fba163b123ebc047d735437c6a6677e6f	1243	76	76.32%	100%
37	0x66ca1f903a43942992c41d610e8a33a951914d33	1327	74	77.03%	100%
38	0x22895ba3ee81ab5f12753bd13b52858f8857d518	1490	68	82.35%	100%
39	0x442e735978155ed54ab19201ec834bb519f60168	1270	65	81.54%	100%
40	0x3af2aE62F0D3353C9F15B7fe678ccDAF2b2157C9	1278	61	88.52%	100%
41	0x289a7d1d22abf90595b4f8d109e7de71f03d42d4	1208	60	85.00%	100%

Tabella 9.2: Confronto con il benchmark effettuato da Davide Tarpini [2].

9.3 Confronto con EtherSolve

Nell'ambito delle analisi degli smart contract online, si riscontrano diverse alternative, tra cui EtherSolve[2] [23], il quale, analogamente a EVMLiSA, si focalizza sull'analisi del bytecode degli smart contract con la creazione di un CFG completo. Tuttavia, è importante notare che EtherSolve non riesce ad ottenere un CFG completo in tutti i casi [15]. Ad esempio, su un benchmark costituito da 1000 smart contract scaricati casualmente da EtherScan, EtherSolve raggiunge un tasso di successo (i.e., la percentuale di risoluzione delle jump) del 94,61%.

Effettuando lo stesso benchmark su EVMLiSA, e fissando le dimensioni massime sia di `AbstractStack` che di `AbstractStackSet` a 32, otteniamo una percentuale di successo del 100%, mantenendo così la coerenza con i risultati dei due benchmark menzionati in precedenza.

[2] `https://github.com/SeUniVr/EtherSolve`

Capitolo 10

Conclusione

Questo studio si è focalizzato sulla creazione di un Control-Flow Graph accurato e completo del bytecode dell'Ethereum Virtual Machine (EVM). Durante l'analisi, è stato esaminato attentamente il flusso di controllo del bytecode EVM e l'importanza di risolvere le jump orfane per prevenire vulnerabilità nel codice. Inoltre, sono stati esplorati gli errori comuni negli smart contract, concentrandosi particolarmente sulle problematiche legate alla rientranza e alle possibili conseguenze di errori di programmazione.

Per raggiungere l'obiettivo di sviluppare un CFG accurato e completo, è stata adottata l'interpretazione astratta come framework teorico di riferimento. In questo contesto, sono stati implementati diversi domini astratti, tra cui il dominio delle variabili astratte, degli stack astratti e degli insiemi degli stack astratti. Questi domini astratti hanno consentito un'analisi statica precisa degli smart contract, garantendo una rappresentazione accurata del comportamento del programma e la creazione di un CFG completo.

Attraverso il mio contributo allo sviluppo di EVMLiSA, abbiamo potuto esplorare il ruolo fondamentale della generazione del CFG nell'analisi e nella verifica della sicurezza degli smart contract. Grazie alla versione attuale di EVMLiSA, siamo in grado di costruire CFG completi, consentendo un'analisi più accurata e dettagliata del flusso di esecuzione dei programmi bytecode EVM.

I risultati ottenuti durante questa ricerca hanno dimostrato l'efficacia dell'approccio proposto nella creazione di un CFG mediante interpretazione astratta. Utilizzando diversi benchmark, è stato possibile confermare che il metodo basato sull'analisi semantica e sulla risoluzione delle jump orfane è in grado di generare un CFG accurato e completo, fondamentale per identificare e prevenire potenziali vulnerabilità nel codice.

Per quanto riguarda i lavori futuri, si prevede lo sviluppo di nuove analisi avanzate per migliorare ulteriormente la sicurezza degli smart contract. Tra le prospettive future vi sono l'implementazione di un *reentrancy checker*, un

gas estimator e un *buffer overflow checker*, che permetteranno di identificare e prevenire in modo più efficace possibili vulnerabilità nel codice. Queste nuove funzionalità potrebbero rappresentare un passo avanti significativo nell'ambito dell'analisi statica degli smart contract e potrebbero contribuire a rafforzare la sicurezza e l'affidabilità delle applicazioni basate su blockchain.

Bibliografia

[1] Wikipedia, "Crittografia asimmetrica," 2024. [Online]. Available: https://it.wikipedia.org/wiki/Crittografia_asimmetrica

[2] D. Tarpini, "Risoluzione di jump orfane in bytecode evm tramite stack simbolici e interpretazione astratta," 2023.

[3] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[5] G. Chiap, "Blockchain: tecnologia e applicazioni per il business," 2019.

[6] Permisison.io, "Permissioned vs. permissionless blockchains explained," 2021. [Online]. Available: https://www.permission.io/blog/permissioned-vs-permissionless-blockchains-explained

[7] T. K. Sharma, "Types of blockchains explained - public vs. private vs. consortium," 2023. [Online]. Available: https://www.blockchain-council.org/blockchain/types-of-blockchains-explained-public-vs-private-vs-consortium/

[8] criptoinvestire.com, "Crittografia: la sicurezza delle blockchain," 2018. [Online]. Available: https://www.criptoinvestire.com/come-funziona-la-crittografia-nelle-blockchain.html#:~:text=La%20blockchain%20utilizza%20la%20crittografia,e%20di%20una%20chiave%20privata

[9] V. Buterin, "Ethereum white paper," *GitHub repository*, vol. 1, pp. 22–23, 2013.

[10] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[11] Nico *et al.*, "Ethereum accounts," 2023. [Online]. Available: https://ethereum.org/en/developers/docs/accounts/

[12] ——, "Ethereum smart contracts," 2023. [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/

[13] N. Szabo, "Smart contracts: building blocks for digital markets," *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, no. 2, p. 28, 1996.

[14] Nico *et al.*, "Ethereum virtual machine," 2023. [Online]. Available: https://ethereum.org/en/developers/docs/evm/

[15] M. Pasqua, A. Benini, F. Contro, M. Crosara, M. Dalla Preda, and M. Ceccato, "Enhancing ethereum smart-contracts static analysis by computing a precise control-flow graph of ethereum bytecode," *Journal of Systems and Software*, vol. 200, p. 111653, 2023.

[16] Herbie *et al.*, "Smart contract security," 2023. [Online]. Available: https://ethereum.org/it/developers/docs/smart-contracts/security/

[17] D. Siegel, "Understanding the dao attack," 2023. [Online]. Available: https://www.coindesk.com/learn/understanding-the-dao-attack/

[18] K. Olwksii, "Takeaways: 5 years after the dao crisis and ethereum hard fork," 2021. [Online]. Available: https://it.cointelegraph.com/news/takeaways-5-years-after-the-dao-crisis-and-ethereum-hard-fork

[19] V. Arceri, "Doctoral thesis: Taming strings in dynamic languages an abstract interpretation-based static analysis approach," 2020.

[20] V. Arceri, I. Mastroeni, and E. Zaffanella, "Decoupling the ascending and descending phases in abstract interpretation," in *Asian Symposium on Programming Languages and Systems*. Springer, 2022, pp. 25–44.

[21] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.

[22] P. Ferrara, L. Negrini, V. Arceri, and A. Cortesi, "Static analysis for dummies: experiencing lisa," in *SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021*, L. N. Q. Do and C. Urban, Eds. ACM, 2021, pp. 1–6. [Online]. Available: https://doi.org/10.1145/3460946.3464316

[23] F. Contro, M. Crosara, M. Ceccato, and M. Dalla Preda, "Ethersolve: Computing an accurate control-flow graph from ethereum bytecode," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 127–137.

Ringraziamenti

Vorrei esprimere la mia più profonda gratitudine a tutte le persone che mi sono state vicine durante questo viaggio.

Un grazie speciale va alla mia famiglia, per il loro sostegno costante e gli incoraggiamenti che mi hanno dato la forza di superare ogni sfida.

Un ringraziamento di cuore ai miei colleghi, che con la loro collaborazione e il loro supporto hanno reso questo percorso più stimolante e arricchente. La loro presenza è stata fondamentale per la mia crescita professionale e personale.

Infine, un pensiero speciale va alla mia ragazza, la mia fonte d'ispirazione. La sua motivazione costante e il suo amore mi hanno accompagnato in ogni momento, dandomi la spinta per andare avanti e credere in me stesso.

A tutti voi, va la mia eterna riconoscenza. Grazie di cuore.