UNIVERSITÀ
DI PARMA

Dipartimento di Scienze Matematiche, Fisiche e Informatiche

Corso di Laurea Magistrale in Scienze Informatiche

Rilevamento Statico di Access Control Incompleteness in Smart Contract EVM: Analisi Taint Relazionale per Bridge Cross-Chain

Candidato:
Dott. Merenda Saverio Mattia

Relatore:
Prof. Arceri Vincenzo

Matricola:
376544

Anno Accademico 2025–2026

Dedicata a mamma e papà.

Indice

Elenco delle figure

Elenco degli algoritmi

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Contesto e motivazioni

Gli *smart contract* sono programmi distribuiti su una blockchain in modo immutabile: ogni bug presente nel codice resta sfruttabile per l'intera vita del contratto e può portare alla sottrazione dei fondi che esso gestisce [4]. Individuare le vulnerabilità prima del *deployment* è quindi essenziale; l'analisi statica è lo strumento adatto a questo scopo, poiché ragiona sulle proprietà del codice senza eseguirlo e considera simultaneamente tutte le esecuzioni possibili.

Tra le classi di vulnerabilità più sfruttate negli ultimi anni si colloca l'*Access Control Incompleteness*, condizione in cui una funzione che modifica lo stato critico di un contratto non è protetta da un controllo di autorizzazione adeguato. Le compromissioni dei *bridge cross-chain* Ronin e Nomad, con perdite per 625 e 190 milioni di dollari nel solo 2022 [6, 16], derivano da incompletezze di questo tipo, e un'analisi trasversale degli incidenti recenti [15] conferma che si tratta della classe di vulnerabilità più ricorrente in questo contesto.

La presente tesi propone un *checker* statico per il rilevamento di Access Control Incompleteness nei *bridge cross-chain* a partire dal solo *bytecode* EVM. Il *checker* è costruito su EVMLiSA,[1] un analizzatore statico basato sul *framework* LiSA [25, 23] e sull'Interpretazione Astratta. La tesi formalizza il nucleo di EVMLiSA e dimostra *soundness* e terminazione della costruzione del *Control-Flow Graph* [3]; su tale grafo opera un'analisi *taint* relazionale che traccia la provenienza dei valori non fidati a livello di *program point* e verifica che ogni cammino da una *source* a un *sink* attraversi un *sanitizer* di autorizzazione. La valutazione coinvolge 16 *bridge cross-chain* reali, 1.003 contratti in totale, con *recall* del 100% e *precision* fino al 94,36%.

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

1.2 Contributi della tesi

1. **Formalizzazione della costruzione *sound* dei *Control-Flow Graph*.** Si formalizza la semantica concreta dell'EVM, si definisce il dominio astratto Σ_l^h come reticolo di insiemi di *stack* astratti e si dimostra che la semantica astratta sovra-approssima quella concreta. Si stabiliscono inoltre le condizioni di terminazione dell'iterazione verso il punto fisso. Il contributo è presentato nel Capitolo 4.

2. **Analisi *taint* relazionale con *provenance*.** Il dominio *taint* classico viene esteso con un componente che associa a ogni dato *tainted* il *program point* di origine, così da distinguere flussi diversi che confluiscono nello stesso *sink*. Il dominio è presentato nel Capitolo 5.

3. ***Checker* a due fasi per *Access Control Incompleteness*.** Le istruzioni `JUMPI` con condizione *tainted* sono identificate come *sanitizer* di autorizzazione; ogni cammino fra una *source* e un *sink* privo di tale copertura viene segnalato come potenziale Access Control Incompleteness. Il *checker* è descritto nel Capitolo 5.

4. **Interfaccia di annotazione per i *modifier* Solidity.** Un meccanismo opzionale di annotazione recupera la semantica dei *modifier* dispersa nel *bytecode* dall'inlining del compilatore, e riduce i falsi positivi quando l'ABI è disponibile. L'interfaccia è descritta nel Capitolo 5.

5. **Valutazione sperimentale su *bridge* reali.** Il *checker* è valutato su 16 applicazioni *cross-chain* per un totale di 1.003 contratti e 109.952 righe di *bytecode*, con una *ground truth* di 301 vulnerabilità validate manualmente. Il *recall* è del 100% in entrambe le configurazioni, mentre la *precision* passa dal 73,24% con la sola ABI al 94,36% nella configurazione *modifier-aware*. I dettagli sono riportati nel Capitolo 6.

1.3 Struttura della tesi

Il Capitolo 2 copre i fondamenti della blockchain, il modello di esecuzione di Ethereum, l'architettura dell'EVM e le principali classi di vulnerabilità degli *smart contract*. Il Capitolo 3 tratta l'Interpretazione Astratta: dominio astratto, connessione di Galois, semantica astratta e calcolo del punto fisso.

Il Capitolo 4 presenta EVMLiSA e la costruzione *sound* del *Control-Flow Graph* tramite il dominio Σ_l^h: semantica concreta, gerarchia dei domini astratti, dimostrazioni di correttezza e terminazione, algoritmo iterativo di risoluzione dei salti. Il Capitolo 5 descrive il *checker* per il rilevamento di Access

Control Incompleteness nei *bridge cross-chain*: dominio *taint* relazionale, identificazione di *source*, *sink* e *sanitizer*, interfaccia di annotazione per i *modifier* Solidity.

Il Capitolo 6 riporta la valutazione sperimentale: costruzione della *ground truth*, configurazioni di analisi e risultati di *recall* e *precision* sui 16 *bridge*. Il Capitolo 7 discute limitazioni e possibili estensioni. Il Capitolo 8 riassume i contributi e indica le direzioni di lavoro futuro. L'Appendice A raccoglie le dimostrazioni complete dei teoremi del Capitolo 4 e del Capitolo 5.

Capitolo 2

Background

Questo capitolo introduce i fondamenti tecnologici e concettuali necessari alla comprensione del lavoro presentato nei capitoli successivi. Si trattano la tecnologia *blockchain* e i meccanismi di consenso distribuito, la piattaforma Ethereum e il concetto di *smart contract*, l'architettura dell'Ethereum Virtual Machine (EVM) con il problema dei salti dinamici al centro, le principali vulnerabilità degli *smart contract* e, infine, i *cross-chain bridge* e la vulnerabilità di *Access Control Incompleteness* oggetto del contributo originale di questa tesi.

2.1 Blockchain e consenso distribuito

La *blockchain* è stata introdotta nel 2008 da Satoshi Nakamoto nel *white-paper* che definisce il protocollo Bitcoin [22], con la prima implementazione operativa nel 2009. L'idea centrale consiste nel realizzare un registro digitale decentralizzato e distribuito, strutturato come una catena di blocchi collegati crittograficamente, in modo che la storia delle transazioni sia verificabile da chiunque ma non alterabile da nessuno. La *blockchain* è un caso particolare di *Distributed Ledger Technology* (DLT) [7]: un sistema in cui più nodi mantengono copie identiche e sincronizzate di un registro condiviso, senza alcuna autorità centrale di controllo. Bitcoin è soltanto una delle possibili applicazioni di questa tecnologia; la *blockchain* è un paradigma più generale.

Ogni blocco della catena contiene un insieme di transazioni e il riferimento crittografico al blocco precedente, tipicamente calcolato tramite funzioni di hash come SHA-256. Questa struttura garantisce l'*immutabilità* della catena: alterare anche un solo blocco invalida il suo hash e, a cascata, quello di tutti i blocchi successivi, rendendo la manomissione rilevabile immediatamente da qualunque nodo della rete.

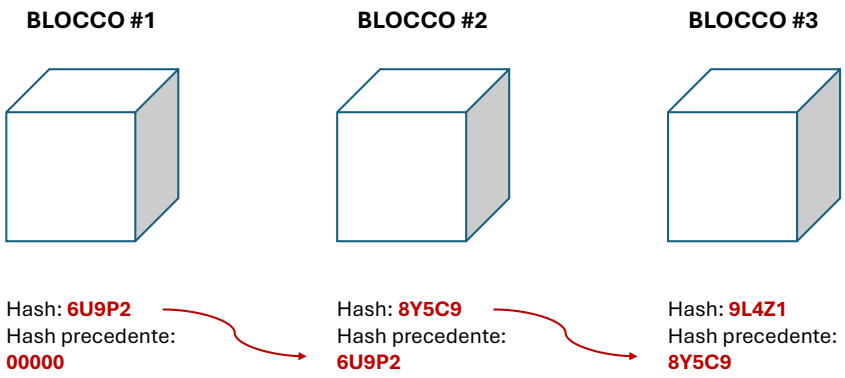


Figura 2.1: Struttura di una blockchain: ogni blocco contiene l'hash del blocco precedente, garantendo l'immutabilità della catena.

Per aggiungere un nuovo blocco alla catena è necessario raggiungere un accordo collettivo tra i nodi della rete: questo processo è noto come *consenso distribuito*. I due meccanismi più diffusi sono il *Proof of Work* (PoW) e il *Proof of Stake* (PoS). Nel PoW, i nodi, detti *miner*, competono per risolvere un problema crittografico computazionalmente costoso; il primo a trovare la soluzione propone il blocco successivo e riceve una ricompensa in criptovaluta. Il meccanismo è robusto ma comporta un elevato consumo energetico. Nel PoS, i *validator* mettono in *stake* una quota della propria criptovaluta: la probabilità di essere selezionati per proporre il blocco successivo è proporzionale alla quota depositata. Il PoS riduce il consumo energetico rispetto al PoW con garanzie di sicurezza comparabili.

Dal punto di vista delle politiche di accesso, le reti *blockchain* si distinguono in *permissionless*, aperte alla partecipazione di chiunque senza necessità di autorizzazione [36], e *permissioned*, accessibili solo a partecipanti noti e autorizzati. In base alla visibilità dei dati si distinguono inoltre reti pubbliche, private e *consortium* [38]: le reti pubbliche sono accessibili a qualsiasi partecipante in lettura e scrittura; le reti private sono controllate da un singolo operatore che ne determina l'accesso; le reti *consortium* sono gestite collettivamente da un gruppo ristretto di organizzazioni che si accordano sui criteri di partecipazione. Ethereum, piattaforma centrale in questa tesi, è una rete *permissionless* e pubblica.

La partecipazione alla rete richiede un meccanismo di autenticazione basato su crittografia asimmetrica [42, 11]. Ogni utente dispone di una coppia di chiavi: una chiave privata, nota solo al proprietario, e una chiave pubblica derivata da essa, visibile a tutti. Le transazioni vengono firmate con la chiave privata del mittente e verificate da qualunque nodo attraverso la corrispondente chiave pubblica: l'autenticità e il non-ripudio sono così assicurati senza alcuna autorità centrale di certificazione.

2.2 Ethereum e smart contract

Ethereum, ideata da Vitalik Buterin nel 2013 [5], è una piattaforma *blockchain* general-purpose che estende le capacità di Bitcoin introducendo un ambiente di esecuzione programmabile nel quale è possibile definire ed eseguire logica arbitraria direttamente sulla rete distribuita. Mentre Bitcoin offre un linguaggio di *scripting* volutamente limitato, privo di capacità di calcolo generale, Ethereum mette a disposizione un ambiente di esecuzione quasi Turing-completo che consente di definire logica arbitraria da eseguire sulla *blockchain*; la limitazione pratica di questo modello è garantita dal meccanismo del *gas*, descritto in seguito. Ethereum può essere concepita come una "macchina virtuale globale": un calcolatore distribuito il cui stato è mantenuto in modo coerente da tutti i nodi della rete, con elevata disponibilità, trasparenza e neutralità [1].

La piattaforma distingue due tipologie di account [26]. Gli *Externally Owned Account* (EOA) sono controllati direttamente da utenti tramite una coppia di chiavi crittografiche e possono originare transazioni. I *Contract Account* non hanno una chiave privata associata: il loro comportamento è interamente determinato dal codice dello *smart contract*, un programma autonomo compilato in *bytecode* EVM e distribuito sulla *blockchain* tramite una transazione speciale [41], che li governa; essi vengono attivati solo in risposta a messaggi ricevuti.

Le transazioni su Ethereum sono firmate crittograficamente e sono atomiche: in caso di errore durante l'esecuzione, tutte le modifiche allo stato vengono annullate tramite *rollback* e la transazione risulta fallita, senza effetti parziali [27].

Un elemento caratterizzante di Ethereum è il meccanismo del *gas*: un'unità di misura del costo computazionale associato all'esecuzione di ogni istruzione. Operazioni diverse hanno costi diversi, ad esempio, un'operazione di somma (`ADD`) costa 3 gas, mentre il calcolo di un hash (`KECCAK256`) ne costa 30 [7]. Il costo totale di una transazione è determinato dalla formula:

$$\text{Costo} = \text{Gas Limit} \times \text{Gas Price} \tag{2.1}$$

dove il *Gas Limit* è il numero massimo di unità di gas che il mittente è disposto a consumare e il *Gas Price* è il prezzo per unità, espresso in *wei*, con 1 wei = 10^{-18} ETH. Questo meccanismo assolve una duplice funzione: protegge la rete da attacchi di tipo *Denial of Service* (DoS) basati su cicli infiniti, e rende il costo dell'esecuzione proporzionale alle risorse consumate. Quest'ultimo aspetto giustifica la definizione di Ethereum come sistema "quasi" Turing-completo.

Il concetto di *smart contract* fu teorizzato da Nick Szabo nel 1996 [41] come un programma in grado di replicare automaticamente le clausole di un contratto legale senza la necessità di un intermediario; su Ethereum questa visione

diventa concreta grazie alla possibilità di scrivere tali programmi tipicamente in Solidity e di distribuirli sulla *blockchain* indirizzando una transazione speciale all'account zero (`0x0`). Una volta distribuito, il contratto riceve un indirizzo Ethereum permanente e può essere richiamato da qualsiasi EOA o da altri contratti.

Una caratteristica degli *smart contract* che ne determina sia l'utilità sia i rischi è l'**immutabilità**: il codice distribuito sulla *blockchain* non può essere modificato in alcun modo [27]. Le regole definite dal contratto vengono rispettate automaticamente e non possono essere alterate unilateralmente da nessuna delle parti, con un grado di affidabilità non raggiungibile con sistemi centralizzati. L'immutabilità ha però un costo: eventuali difetti nel codice non possono essere corretti dopo la distribuzione, e la verifica preventiva della correttezza diventa necessaria. Questa criticità motiva lo sviluppo di strumenti di analisi statica come EVMLiSA, oggetto dei capitoli successivi.

Il *bytecode* generato dal compilatore Solidity viene eseguito da una macchina virtuale dedicata, l'Ethereum Virtual Machine, la cui architettura e il cui set di istruzioni sono descritti nella sezione seguente.

2.3 Ethereum Virtual Machine

L'Ethereum Virtual Machine (EVM) è il componente software al cuore della piattaforma Ethereum: è responsabile dell'esecuzione degli *smart contract* e dell'aggiornamento dello stato globale della rete a ogni transazione. Nelle sottosezioni che seguono si descrivono l'architettura e il modello di esecuzione dell'EVM (Sezione 2.3.1), il formato del *bytecode* e il set di istruzioni che la macchina virtuale interpreta (Sezione 2.3.2), e il problema dei salti dinamici e degli *orphan jump* (Sezione 2.3.3). Quest'ultimo è la sfida centrale che EVMLiSA affronta mediante analisi statica *sound*,[1] come approfondito nei capitoli successivi.

2.3.1 Architettura e modello di esecuzione

L'EVM è il componente che esegue gli *smart contract* su Ethereum: ogni azione sulla rete produce un aggiornamento dello stato globale calcolato dall'EVM. A un livello d'astrazione elevato, l'EVM è un calcolatore decentralizzato e globale [1].

[1]Il termine *sound*, nella teoria dell'interpretazione astratta, indica che l'analisi non omette mai comportamenti reali: ogni esecuzione concreta è inclusa nell'approssimazione calcolata. In italiano si traduce con *completo*, nel senso di privo di falsi negativi.

Il comportamento dell'EVM è catturato formalmente dalla *funzione di transizione di stato*:

$$Y(S, T) = S' \tag{2.2}$$

dove S rappresenta lo stato corrente della *blockchain*, T un insieme di transazioni da applicare e S' il nuovo stato risultante [28]. L'EVM è definita come macchina *quasi* Turing-completa: teoricamente è in grado di eseguire qualsiasi computazione, ma l'esecuzione è limitata dalla quantità di *gas* disponibile, che impedisce cicli infiniti e rende il sistema *de facto* non Turing-completo, come già osservato a proposito del meccanismo del *gas* nella sezione precedente.

L'EVM è una macchina a *stack* di tipo LIFO (*Last In, First Out*), con una profondità massima di 1024 elementi, ciascuno dei quali è una parola di 256 bit [43]. Questa scelta di un tipo di dato ampio facilita le operazioni crittografiche tipiche degli *smart contract*. Oltre allo *stack*, lo stato di esecuzione dell'EVM è descritto da un insieme di componenti che la Figura 2.2 raggruppa in base alla loro persistenza.

Le componenti *volatili* esistono soltanto per la durata di una singola transazione e vengono reinizializzate a ogni esecuzione. Lo *stack*, già introdotto, conserva i valori intermedi della computazione. La *memory* è un'area di memoria lineare e indirizzabile a byte, inizializzata a zero, impiegata per i dati temporanei che eccedono la capacità dello *stack*. Il *Program Counter* (PC) indica l'offset dell'opcode in corso di esecuzione. La *calldata* contiene i dati di input immutabili forniti dalla transazione che ha invocato il contratto. Il *gas counter* tiene traccia del *gas* residuo, che decrementa a ogni istruzione eseguita.

Le componenti *persistenti* sopravvivono invece alla singola transazione. Il codice del contratto, ossia il *bytecode* EVM, risiede in un'area in sola lettura e immutabile. Lo *storage* è una memoria associata a ciascun *Contract Account* e parte integrante dello stato di Ethereum, le cui modifiche permangono oltre la transazione che le ha prodotte.

Lo stato globale di Ethereum è organizzato su due livelli [1]. Il livello *globale* è un *mapping* che associa indirizzi a 160 bit agli account. Il livello *account* descrive ciascun account tramite quattro campi: il saldo espresso in *wei*, il *nonce* che conteggia le transazioni originate, lo *storage* e il codice, quest'ultimo vuoto per gli EOA e presente per i *Contract Account*.

Il ciclo di vita di una transazione che invoca uno *smart contract* segue una sequenza precisa. Viene creata un'istanza EVM dedicata, il *Program Counter* viene inizializzato a zero, lo *storage* del contratto viene recuperato dallo stato globale e la *memory* viene inizializzata a zero. L'EVM procede poi all'esecuzione sequenziale degli opcode, detraendo il *gas* corrispondente a ogni istruzione. Qualora il *gas* si esaurisca prima del completamento dell'esecuzione, viene sollevata un'eccezione di tipo *Out of Gas* (OOG), che causa il ripristino dello

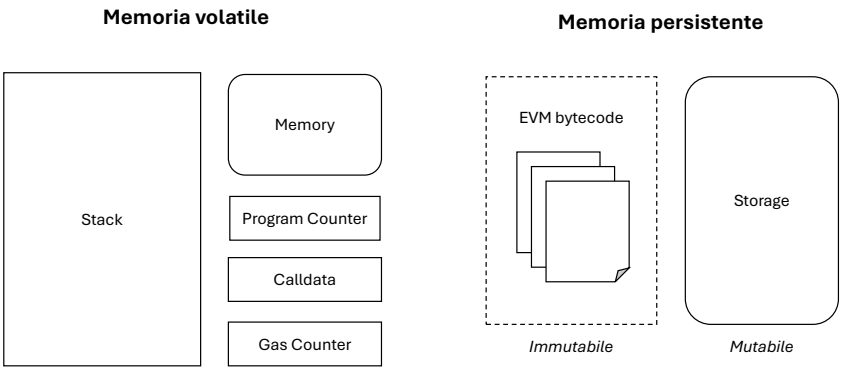


Figura 2.2: Componenti dell'Ethereum Virtual Machine.

stato precedente, ossia un *rollback*, senza effetti parziali sullo stato. Se invece l'esecuzione termina con successo, le modifiche alla copia isolata dello stato vengono propagate allo stato globale; in caso di fallimento dovuto ad altre cause, come l'esecuzione di un'istruzione `REVERT`, la copia viene scartata. Questo modello garantisce l'atomicità di ogni transazione [1].

2.3.2 Bytecode e set di istruzioni

Il *bytecode* EVM è un linguaggio di basso livello, quasi Turing-completo e basato su *stack*, composto da circa 150 istruzioni denominate *opcode*. Ciascun opcode è codificato come numero esadecimale a partire da `0x00` [43]. Gli opcode vengono interpretati dall'EVM per manipolare uno *stack* di profondità massima 1024, i cui elementi sono parole a 256 bit [1].

Come semplice esempio di esecuzione, si consideri la sequenza di *byte* `60 01 60 02 01`: il prefisso `60` corrisponde all'opcode `PUSH1`, che carica nello *stack* il byte immediatamente successivo. La sequenza viene pertanto decodificata come `PUSH1 0x01`, `PUSH1 0x02`, `ADD`, che porta in cima allo *stack* il valore $1 + 2 = 3$. La Figura 2.3 illustra questo frammento di esecuzione in dettaglio.

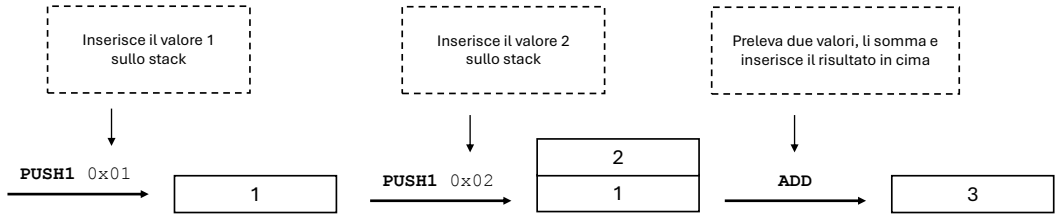


Figura 2.3: Esecuzione di un frammento di bytecode EVM che calcola la somma di due interi.

Le istruzioni del set EVM possono essere raggruppate nelle seguenti categorie [43]:

- **Operazioni aritmetiche e logiche**: addizione, sottrazione, moltiplicazione, divisione, confronto, operazioni bit a bit e crittografiche, quali `ADD`, `MUL`, `LT`, `KECCAK256`.
- **Accesso a stack, memory e storage**: lettura e scrittura degli elementi dello *stack* e dei due spazi di memoria, tramite istruzioni come `PUSH`, `POP`, `DUP`, `SWAP`, `MLOAD`, `MSTORE`, `SLOAD`, `SSTORE`.
- **Controllo del flusso di esecuzione**: istruzioni che alterano il valore del *Program Counter*, ovvero `JUMP`, `JUMPI` e `JUMPDEST`.
- **Logging**: emissione di eventi registrati nella ricevuta della transazione, mediante le istruzioni `LOG0`–`LOG4`.
- **Chiamate a contratti e creazione**: interazione con altri contratti e distribuzione di nuovi contratti, attraverso istruzioni come `CALL`, `DELEGATECALL` e `CREATE`.

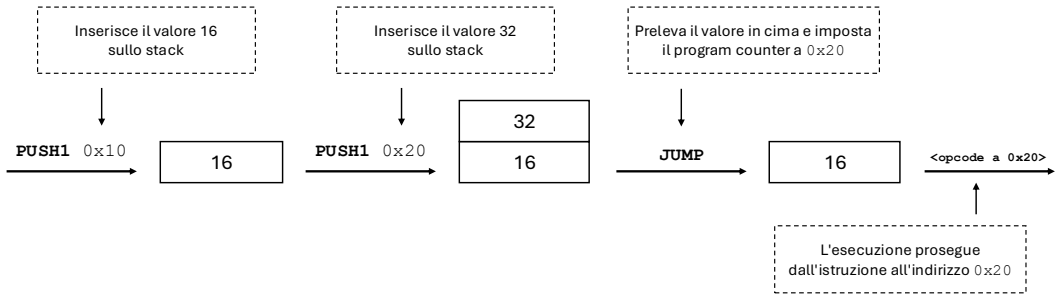
Il flusso di esecuzione inizia con il primo opcode e prosegue in sequenza, incrementando il PC a ogni istruzione. Le uniche istruzioni che alterano il flusso senza interrompere l'esecuzione sono `JUMP` e `JUMPI`:

- `JUMP`: salto incondizionato. L'indirizzo di destinazione è prelevato dalla cima dello *stack*; l'esecuzione riprende dall'opcode presente a quell'indirizzo.
- `JUMPI`: salto condizionato. I due elementi in cima allo *stack* sono, nell'ordine, l'indirizzo di destinazione e la condizione: il salto viene effettuato se e solo se la condizione è diversa da zero.
- `JUMPDEST`: marcatore che identifica le posizioni del *bytecode* valide come destinazione di un salto; ogni salto verso una posizione non contrassegnata da `JUMPDEST` causa un'eccezione.

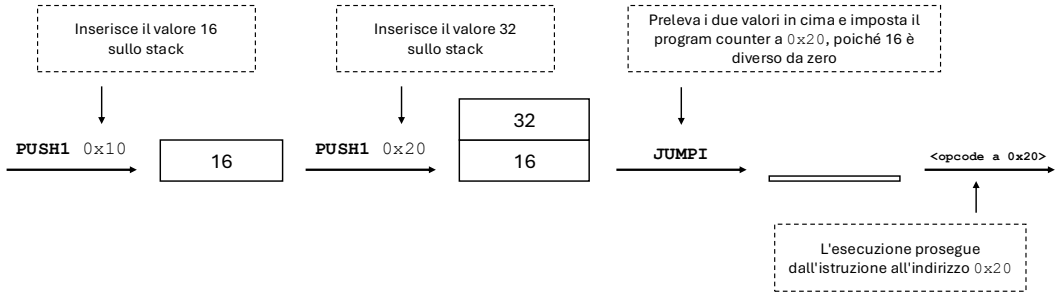
La Figura 2.4 illustra l'esecuzione di frammenti di *bytecode* che coinvolgono entrambe le istruzioni di salto.

A titolo esemplificativo, si consideri il *bytecode* generato dalla compilazione di un ciclo `while(i < 3) { i++; }`. Lo Snippet di Codice 2.1 riporta la sequenza di opcode corrispondente, affiancata dal *Control-Flow Graph* generato nella Figura 2.5.

Il *bytecode* può essere analizzato istruzione per istruzione per ricavare la struttura di controllo del programma. All'indirizzo `0x00`, l'istruzione `PUSH1`



(a) Istruzione `JUMP`.



(b) Istruzione `JUMPI`.

Figura 2.4: Esecuzione di frammenti di bytecode EVM con istruzioni di salto.

`0x00` carica il valore 0 sullo *stack*, inizializzando la variabile contatore `i`. L'indirizzo `0x02` è marcato da `JUMPDEST` e costituisce l'intestazione del ciclo, ovvero il punto al quale l'esecuzione ritorna a ogni iterazione. Le istruzioni `PUSH1 0x03` e `DUP2` agli indirizzi `0x03` e `0x05` caricano la costante 3 e duplicano in cima allo *stack* il valore corrente di `i`; la successiva `LT` a `0x06` consuma entrambi i valori e deposita 1 se `i < 3`, 0 altrimenti.

A questo punto, `PUSH1 0x0D` a `0x07` carica la destinazione del salto condizionato, e `JUMPI` a `0x09` preleva l'indirizzo di destinazione e la condizione dallo *stack*: se `i < 3`, il controllo salta al corpo del ciclo all'indirizzo `0x0D`; in caso contrario, l'esecuzione prosegue in sequenza. Nel ramo del corpo del ciclo, le istruzioni agli indirizzi `0x0E`–`0x10` incrementano `i` di 1 aggiungendo la costante 1 al valore in cima allo *stack*; `PUSH1 0x02` a `0x11` carica l'indirizzo dell'intestazione del ciclo, e `JUMP` a `0x13` trasferisce incondizionatamente il controllo a `0x02`, chiudendo il ciclo.

Nel ramo della condizione falsa, `PUSH1 0x14` a `0x0A` carica l'indirizzo della destinazione di uscita e `JUMP` a `0x0C` salta a `0x14`, dove `JUMPDEST` marca l'uscita e `STOP` a `0x15` termina l'esecuzione. Figura 2.5, è un *Control-Flow Graph* con tre nodi di base: l'intestazione del ciclo, il corpo e il blocco di uscita, collegati da due archi di salto, uno condizionato e uno incondizionato, che codificano rispettivamente la continuazione e la terminazione del ciclo. Questo

```
0x00: PUSH1 0x00
0x02: JUMPDEST
0x03: PUSH1 0x03
0x05: DUP2
0x06: LT
0x07: PUSH1 0x0D
0x09: JUMPI  # salto condizionato: (i < 3)
0x0A: PUSH1 0x14
0x0C: JUMP
0x0D: JUMPDEST
0x0E: PUSH1 0x01
0x10: ADD
0x11: PUSH1 0x02
0x13: JUMP
0x14: JUMPDEST
0x15: STOP
```

Codice 2.1: Bytecode EVM di un ciclo `while`.

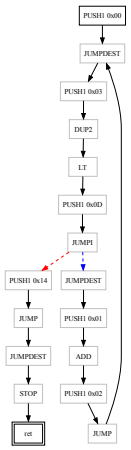


Figura 2.5: Control-Flow Graph del ciclo `while` in bytecode EVM.

esempio mostra anche come le destinazioni dei salti siano valori concreti spinti esplicitamente sullo *stack* dall'istruzione `PUSH` immediatamente precedente: si tratta del caso più semplice, denominato *pushed jump*. Quando invece la destinazione è il risultato di un'espressione aritmetica calcolata sullo *stack*, il suo valore non è determinabile da un'ispezione sintattica del *bytecode*, dando origine al problema degli *orphan jump* trattato nella sezione seguente.

2.3.3 Salti dinamici e orphan jump

Un *Control-Flow Graph* è una rappresentazione del programma come grafo orientato in cui i nodi corrispondono a blocchi di istruzioni eseguiti in sequenza senza salti, e gli archi codificano i trasferimenti di controllo tra blocchi [35]. La costruzione di un *Control-Flow Graph* preciso per il *bytecode* EVM è resa complessa da una proprietà del set di istruzioni: le destinazioni dei salti `JUMP` e `JUMPI` non sono codificate staticamente come operandi dell'istruzione stessa, bensì calcolate dinamicamente a partire dai valori presenti in cima allo *stack* al momento dell'esecuzione. Per risolvere le destinazioni dei salti è pertanto necessario approssimare il contenuto dello *stack* durante l'esecuzione [43].

In base alla possibilità di determinare staticamente l'indirizzo di destinazione, i salti si classificano in due categorie [35]:

Definizione 2.3.1 (Pushed jump)**.** Un salto `JUMP` o `JUMPI` si dice *pushed jump* se la sua destinazione è determinabile staticamente, poiché l'istruzione di salto

è sintatticamente preceduta da un'istruzione `PUSH` che ne specifica direttamente l'indirizzo di destinazione [35].

Definizione 2.3.2 (Orphan jump)**.** Un salto `JUMP` o `JUMPI` si dice *orphan jump* se la sua destinazione non è determinabile da un'ispezione puramente sintattica del *bytecode*, ma richiede un'analisi del programma in grado di approssimare il contenuto dello *stack* durante l'esecuzione [35].

I *pushed jump* costituiscono il caso più comune e semplice da trattare: nella maggioranza dei contratti reali, la destinazione di un salto è il valore immediatamente preceduto da un'istruzione `PUSH`. Tuttavia, un compilatore può generare sequenze in cui l'indirizzo di destinazione è il risultato di un'espressione aritmetica calcolata sullo *stack*, dando origine a un *orphan jump*. Si consideri ad esempio la sequenza:

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

La destinazione del salto è il valore `0x0A` + `0x0C` = `0x16`, che non è determinabile da una semplice lettura sintattica del *bytecode*: è necessario propagare i valori dello *stack* attraverso le istruzioni precedenti. La Figura 2.6 illustra questo caso.

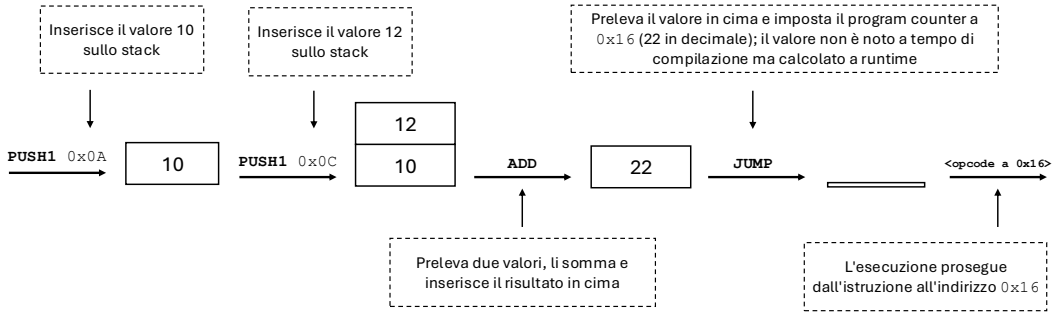


Figura 2.6: Esecuzione di un frammento di bytecode EVM con un orphan jump.

La risoluzione degli *orphan jump* è dunque un problema di analisi del programma: occorre uno strumento in grado di dedurre i possibili contenuti dello *stack* a ogni punto del programma, tenendo conto della semantica di ciascuna istruzione. Questa problematica motiva direttamente l'adozione di tecniche di analisi statica fondate sull'interpretazione astratta, introdotte nel Capitolo 3 e applicate alla costruzione di *Control-Flow Graph* completi per il *bytecode* EVM nel Capitolo 4.

2.4 Vulnerabilità degli smart contract

Gli *smart contract*, come qualsiasi artefatto software, possono contenere difetti logici o implementativi che li rendono vulnerabili ad attacchi. Ciò che distingue questa classe di vulnerabilità da quelle presenti nel software tradizionale è la natura immutabile del codice distribuito sulla *blockchain*: una volta pubblicato il contratto, non è possibile applicare alcuna patch correttiva, il che rende ogni difetto potenzialmente permanente e sfruttabile a tempo indeterminato [27]. Il valore economico custodito dai contratti, sotto forma di criptovaluta o diritti digitali, rende le conseguenze di un attacco gravi. L'OWASP[2] *Smart Contract Top 10* [33] cataloga le classi di vulnerabilità più frequenti e pericolose; tra queste, la *reentrancy*, oggetto di questa sezione, ha prodotto gli incidenti di sicurezza di maggiore impatto nella storia di Ethereum. Le sottosezioni che seguono ne illustrano prima il caso storico più noto, l'attacco al contratto The DAO, e successivamente il meccanismo tecnico generale che lo rese possibile.

2.4.1 The DAO Attack (2016)

The DAO, acronimo di *Decentralized Autonomous Organization*, era un fondo di investimento decentralizzato costruito su Ethereum e lanciato nella primavera del 2016. Il contratto consentiva ai propri sottoscrittori di votare collettivamente l'allocazione del capitale raccolto e arrivò a custodire oltre 150 milioni di dollari in ether, la criptovaluta nativa di Ethereum, e divenne la più grande applicazione finanziaria mai distribuita sulla piattaforma [39].

Nel giugno 2016 un attaccante sfruttò una debolezza nella funzione di prelievo del contratto, la quale trasferiva ether al chiamante prima di aggiornare i saldi interni. Attraverso un ciclo di rientri ricorsivi, nel quale lo stesso prelievo veniva invocato più volte prima dell'azzeramento del saldo, l'avversario riuscì a sottrarre circa 3,6 milioni di ether, equivalenti a circa 60 milioni di dollari al cambio dell'epoca [32].

L'evento aprì un dibattito nella comunità Ethereum che portò a una scelta controversa: per recuperare i fondi sottratti fu eseguito un *hard fork* retroattivo della *blockchain*, un'operazione che modifica la storia delle transazioni e contrasta il principio di immutabilità sul quale la piattaforma è fondata. Una parte della comunità si oppose a tale intervento per ragioni di principio, dando origine a una scissione permanente: la catena originale, non modificata, sopravvive ancora oggi sotto il nome Ethereum Classic (ETC), mentre la catena modificata ha proseguito come Ethereum [39]. L'episodio è ancora oggi citato nel dibattito sulla sicurezza degli *smart contract*: dimostra che un singolo difetto in un contratto ad alto valore può avere ripercussioni irreversibili sul

[2]Open Worldwide Application Security Project.

patrimonio degli utenti coinvolti e sull'intera struttura di una rete *blockchain*. Il meccanismo tecnico che rese possibile l'attacco, noto come *reentrancy*, viene illustrato nella sottosezione successiva.

2.4.2 Reentrancy

La *reentrancy*, la stessa classe di vulnerabilità sfruttata nell'attacco al DAO, deriva da una proprietà strutturale dell'EVM. Poiché l'EVM non supporta la concorrenza, l'esecuzione di una transazione è sempre sequenziale e atomica. Quando però uno *smart contract* effettua una chiamata esterna verso un altro account, mediante l'opcode `CALL` o istruzioni equivalenti, l'esecuzione del chiamante viene temporaneamente sospesa, cedendo il controllo al destinatario della chiamata. Se il destinatario è a sua volta un contratto malevolo, il suo codice può richiamare nuovamente una funzione del contratto chiamante prima che quest'ultimo abbia aggiornato il proprio stato interno. In questa finestra di incoerenza temporanea, l'avversario può compiere operazioni non autorizzate [18].

Il meccanismo può essere illustrato attraverso un esempio concreto. Lo Snippet di Codice 1 mostra un contratto Victim che gestisce saldi in ether tramite funzioni di deposito e prelievo.

Algoritmo 1 Contratto vulnerabile a reentrancy.

```
contract Victim {
    mapping (address => uint256) public balances;

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

    function withdraw() external {
        uint256 amount = balances[msg.sender];
        (bool success, ) = msg.sender.call.value(amount)("");
        require(success);
        balances[msg.sender] = 0;
    }
}
```

La funzione `withdraw()` esegue tre operazioni nell'ordine seguente: legge il saldo del chiamante, trasferisce la corrispondente quantità di ether via `msg.sender.call.value()`, e infine azzera il saldo. Il difetto consiste nel fatto che l'aggiornamento dello stato (`balances[msg.sender] = 0`) avviene *dopo* il trasferimento, lasciando il saldo invariato durante l'intera durata della chiamata esterna.

Lo Snippet di Codice 2 mostra il contratto Attacker che sfrutta questa vulnerabilità.

Algoritmo 2 Contratto attaccante che sfrutta la reentrancy.

```
contract Attacker {
    function beginAttack() external payable {
        Victim(victim_address).deposit.value(1 ether)();
        Victim(victim_address).withdraw();
    }

    function() external payable {
        if (gasleft() > 40000) {
            Victim(victim_address).withdraw();
        }
    }
}
```

L'attaccante avvia l'attacco tramite `beginAttack()`: deposita una piccola quantità di ether per ottenere un saldo legittimo, quindi invoca `withdraw()` su Victim. Non appena Victim trasferisce gli ether, l'EVM esegue automaticamente la *fallback function* del contratto Attacker, ovvero una funzione speciale priva di nome che viene invocata ogni volta che il contratto riceve ether senza che la transazione specifichi un selettore di funzione. Tale funzione richiama a sua volta `withdraw()` su Victim. Poiché il saldo nel contratto Victim non è ancora stato azzerato, la verifica del saldo ha esito positivo e il trasferimento viene ripetuto. Il ciclo si ripete per ogni iterazione in cui il *gas* residuo è sufficiente, consentendo prelievi multipli a fronte di un deposito iniziale unitario [18].

La mitigazione più efficace è il pattern *checks-effects-interactions*: tutte le verifiche sullo stato, dette *checks*, devono essere eseguite per prime, seguite dall'aggiornamento dello stato, detto *effects*, e solo al termine si effettuano le interazioni con contratti esterni, dette *interactions*. Applicando questo pattern a `withdraw()`, il saldo verrebbe azzerato prima dell'invio degli ether, rendendo nullo il vantaggio di qualsiasi rientro nel contratto.

2.5 Cross-chain bridge

I *cross-chain bridge* sono protocolli che consentono il trasferimento di asset e messaggi tra reti eterogenee che, per progetto, non possono interoperare nativamente. La concentrazione di valore e la complessità dei percorsi di esecuzione distribuiti su più *blockchain* li espongono ad attacchi in cui le lacune di controllo degli accessi producono perdite ingenti. Le sottosezioni seguenti ne descrivono l'architettura, i principali incidenti documentati, e la vulnerabilità di *Access Control Incompleteness* come difetto strutturale proprio di questa categoria di sistemi.

2.5.1 Architettura dei bridge

La moltiplicazione di piattaforme *blockchain* eterogenee ha prodotto una situazione in cui asset e liquidità restano confinati all'interno di ciascuna rete, senza meccanismi nativi di comunicazione con le reti adiacenti. I *bridge* sono protocolli che coordinano lo stato tra due o più *blockchain* distinte, gestendo il trasferimento di valore e messaggi attraverso contratti distribuiti su entrambe le *blockchain* coinvolte.

I modelli operativi adottati dai bridge si differenziano in base alla semantica scelta per la rappresentazione dell'asset durante il transito [30]:

- **Lock and unlock**: gli asset vengono immobilizzati in un contratto deposito sulla *blockchain* sorgente e resi disponibili da un contratto speculare sulla *blockchain* di destinazione, con la garanzia che la somma degli asset circolanti nei due sistemi rimanga costante.

- **Burn and mint**: l'asset viene distrutto sulla *blockchain* di origine e ricreato su quella di destinazione; questo approccio elimina la necessità di mantenere riserve bloccate, ma richiede un coordinamento preciso tra le operazioni di distruzione e creazione per evitare emissioni non autorizzate.

- **Lock and mint**: l'asset originale viene bloccato nella *blockchain* sorgente, mentre sulla *blockchain* di destinazione viene coniata una rappresentazione sintetica, riscattabile in qualsiasi momento contro l'asset sottostante.

Dal punto di vista architetturale, un bridge tipico si articola in tre livelli, illustrati nella Figura 2.7. Il *livello contrattuale* comprende gli *smart contract* distribuiti su ciascuna *blockchain*, responsabili del blocco, dello sblocco e della verifica dei messaggi in ingresso. Il *livello di relay* è formato da nodi esterni alla *blockchain*, detti *relayer* o *validator*, che osservano gli eventi emessi dai contratti su una catena e li trasmettono all'altra. Il *meccanismo di consenso* garantisce l'autenticità dei messaggi in arrivo tramite schemi multi-firma, in cui un numero sufficiente di validatori deve attestare la validità di un messaggio prima che il contratto di destinazione lo elabori.

Le piattaforme EVM-compatibili, principalmente Ethereum e i suoi derivati, detengono il maggiore *Total Value Locked* tra tutti gli ecosistemi *blockchain*, con un valore bloccato superiore a 50 miliardi di dollari [12]. La natura asincrona e distribuita dei percorsi di esecuzione cross-chain introduce superfici di attacco che non hanno equivalenti nei contratti monolitici.

Un contratto che opera in un contesto cross-chain deve gestire messaggi provenienti da una *blockchain* esterna attraverso il *relay*, verificarne l'autenticità, e agire di conseguenza senza poter interrogare direttamente lo stato

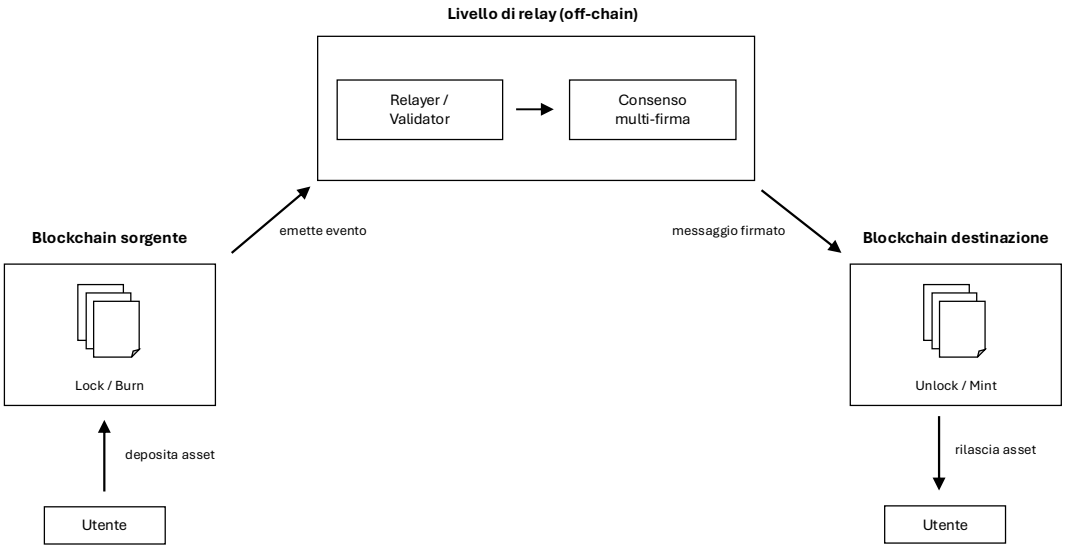


Figura 2.7: Architettura ad alto livello di un bridge cross-chain.

della *blockchain* sorgente al momento dell'esecuzione. Questa separazione tra chi origina l'azione e chi la verifica spiega la maggior parte delle vulnerabilità descritte nella sezione seguente.

2.5.2 Attacchi noti

I *bridge* cross-chain hanno subito attacchi di notevole entità economica. I casi ricorrenti hanno in comune routine di inizializzazione non protette, validazione insufficiente dell'identità del mittente e dipendenza eccessiva da insiemi ristretti di validatori. Si descrivono di seguito tre episodi significativi; l'analisi tecnica dei pattern sottostanti è presentata nel Capitolo 5.

Parity Multisig (2017). Sebbene il protocollo Parity non sia un bridge cross-chain, il suo incidente è considerato un riferimento canonico per le vulnerabilità di controllo degli accessi negli *smart contract*. La falla era situata in una libreria condivisa tra più portafogli multi-firma: la routine di inizializzazione della libreria risultava priva di qualunque protezione, e permetteva a qualsiasi chiamante esterno di assumerne la proprietà e successivamente di distruggerla. La conseguenza fu la perdita definitiva di fondi per un controvalore nell'ordine delle centinaia di milioni di dollari [34]. Il caso Parity dimostra che un singolo punto di accesso non protetto è sufficiente a compromettere l'intera catena di responsabilità di un protocollo, indipendentemente dalla correttezza del resto del codice.

Nomad Bridge (2022). Il protocollo Nomad, un bridge cross-chain per messaggi e asset, subì nel 2022 un attacco che causò perdite superiori a 190 milioni di dollari [16, 17]. La causa tecnica risiedeva in un difetto nella logica di verifica dei messaggi cross-chain: a seguito di un'inizializzazione errata, il contratto accettava come validi messaggi arbitrari in specifiche condizioni di stato, senza verificare che il mittente fosse un *updater* autorizzato. Il meccanismo di accettazione dei messaggi non era vincolato a un'identità fidata, cosicché attori non autorizzati poterono portare il sistema in uno stato permissivo e far passare transazioni di prelievo fraudolente. L'exploit fu replicato da centinaia di indirizzi diversi nell'arco di poche ore, una dinamica resa possibile proprio dall'assenza di qualsiasi controllo sull'identità del chiamante.

Ronin Bridge (2022). Il bridge Ronin, infrastruttura alla base del gioco Axie Infinity, fu compromesso per un importo di circa 625 milioni di dollari [40, 6]. A differenza degli episodi precedenti, la vulnerabilità non era un difetto logico nel codice dei contratti, bensì la compromissione delle chiavi private di un numero sufficiente di validatori. Il bridge era governato da un insieme di nove validatori, di cui cinque necessari per autorizzare i prelievi: una volta che l'attaccante ottenne il controllo di cinque chiavi, l'accesso al capitale custodito fu autorizzato in piena conformità con le regole codificate nel contratto. Il caso Ronin mostra come un modello di controllo degli accessi concentrato in un insieme ristretto di firmatari diventi fragile qualora la gestione delle chiavi risulti insufficiente.

Questi tre incidenti, pur differenti nelle cause immediate, hanno in comune la mancanza o la violazione di invarianti di autorizzazione su percorsi di esecuzione critici. L'analisi sistematica di questi pattern e la metodologia sviluppata per rilevarli staticamente sono oggetto del Capitolo 5.

2.5.3 Access Control Incompleteness

La classe di vulnerabilità che accomuna gli attacchi descritti nella sezione precedente prende il nome di *Access Control Incompleteness* [33]. Nel contesto dei bridge, le conseguenze di questa vulnerabilità si propagano su più catene simultaneamente, con un raggio d'azione che supera quello dei contratti monolitici. La definizione seguente ne formalizza la proprietà centrale.

Definizione 2.5.1 (Access Control Incompleteness)**.** Uno *smart contract* presenta *access control incompleteness* se esiste almeno un punto di ingresso raggiungibile dall'esterno in cui uno o più input controllabili da un attaccante influenzano un'azione privilegiata senza che l'autorità del chiamante venga preventivamente verificata.

La definizione esprime il requisito in modo diretto: non è sufficiente che l'autorizzazione sia verificata su alcuni percorsi di esecuzione; occorre che lo sia su *tutti* i percorsi attraverso i quali un attaccante può raggiungere un'azione privilegiata. Un singolo percorso non coperto è una violazione della proprietà.

In un ambiente *permissionless* come Ethereum, l'*Access Control Incompleteness* non può essere corretta retroattivamente: una volta distribuito il *bytecode*, la logica di controllo degli accessi è immutabile e nessuna governance esterna alla *blockchain* può intervenire [33]. A differenza dei sistemi software tradizionali, dove una patch può essere distribuita in poche ore, uno *smart contract* vulnerabile rimane sfruttabile finché l'ultima unità di valore non è stata sottratta o il contratto non viene disabilitato tramite meccanismi di emergenza, qualora previsti.

I *bridge* cross-chain espongono una superficie di attacco più ampia dei contratti monolitici. Ciascun bridge gestisce almeno due contratti distribuiti su *blockchain* distinte, collegati da percorsi di esecuzione che attraversano *relay* asincroni e insiemi di validatori: ogni segmento introduce un potenziale punto di fallimento dell'autorizzazione. Un messaggio accettato senza verificarne l'origine o un asset coniato senza confermare il blocco sulla *blockchain* sorgente sono casi di *Access Control Incompleteness* le cui conseguenze si propagano su più *blockchain* simultaneamente [30].

Gli strumenti esistenti [19] affrontano le vulnerabilità cross-chain in modo generico e ottengono risultati modesti sulla categoria specifica del controllo degli accessi. EVMLiSA [3] fornisce gli strumenti per costruire *Control-Flow Graph* completi direttamente dal *bytecode* EVM, senza richiedere il codice sorgente; l'approccio per il rilevamento automatico dell'*Access Control Incompleteness* nei *bridge* cross-chain, sviluppato a partire da questa base analitica, è presentato nel Capitolo 5.

Capitolo 3

Analisi Statica e Interpretazione Astratta

Questo capitolo presenta i fondamenti teorici su cui poggiano le analisi di *bytecode* EVM sviluppate nei capitoli successivi. La prima sezione chiarisce le differenze tra analisi statica e analisi dinamica e motiva la scelta dell'approccio statico per la verifica degli *smart contract*. La seconda riepiloga il framework dell'interpretazione astratta [9, 8, 21], con ordini parziali, reticoli, connessioni di Galois e calcolo di punto fisso nella forma usata nel seguito. La terza introduce la nozione di grafo di flusso di controllo e il concetto di CFG *sound* formalizzato in [3]. La quarta, infine, tratta l'analisi *taint* come istanza di analisi di flusso informativo [13, 37] e ne estende il dominio alla variante relazionale con tracciamento della provenienza, alla base del *checker* per l'*Access Control Incompleteness* presentato nel Capitolo 5.

3.1 Analisi statica e analisi dinamica

Le tecniche di verifica del software si dividono in due categorie principali. L'analisi dinamica ragiona su un numero finito di esecuzioni concrete, osservando il comportamento su input specifici tramite *testing*, *fuzzing* o *symbolic execution* parziale. L'analisi statica, invece, ragiona sul codice senza eseguirlo, derivando proprietà valide per ogni possibile esecuzione a partire dalla sola rappresentazione del programma.

Nel contesto degli *smart contract* distribuiti su Ethereum, la verifica statica è preferibile. Il bytecode di un contratto è immutabile dopo la pubblicazione sulla blockchain, non è correggibile a posteriori e, nel caso dei contratti ad alto valore, un singolo percorso di esecuzione non coperto dai test può costare la perdita dell'intero capitale custodito [27]. Un'analisi dinamica fornisce garanzie soltanto sugli input effettivamente eseguiti e non esclude l'esistenza

di percorsi di esecuzione vulnerabili non esplorati. Un'analisi statica correttamente costruita può invece ragionare in modo esaustivo su tutti i possibili stati raggiungibili, con garanzie di copertura che il *testing* non può assicurare.

L'approccio statico paga tale esaustività con un'inevitabile perdita di precisione: per essere computabile, l'analisi deve lavorare su una rappresentazione astratta e approssimata delle esecuzioni concrete. Il framework dell'interpretazione astratta, di cui la Sezione 3.2 richiama i tratti essenziali, offre gli strumenti matematici per costruire analisi statiche *sound* rispetto a una semantica di riferimento, con approssimazioni esplicite e verificabili.

3.2 Interpretazione astratta

L'interpretazione astratta [9, 8] è un framework per il ragionamento *sound* sulle proprietà semantiche di un programma. Stabilisce una corrispondenza tra la semantica concreta, che descrive il comportamento esatto del programma, e una semantica astratta, la cui approssimazione è computabile in modo finito. Questa sezione richiama i concetti di teoria degli ordini e di punto fisso necessari nel seguito; per un'introduzione sistematica si rinvia a [21, Sezione 2].

3.2.1 Ordini parziali, reticoli e catene

Definizione 3.2.1 (Insieme parzialmente ordinato)**.** Un insieme parzialmente ordinato, o *poset*, è una coppia $\langle X, \sqsubseteq \rangle$ costituita da un insieme X dotato di una relazione d'ordine parziale $\sqsubseteq \subseteq X \times X$, vale a dire riflessiva, antisimmetrica e transitiva.

Definizione 3.2.2 (Reticolo)**.** Un reticolo limitato è una struttura algebrica $\langle X, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$, dove $\langle X, \sqsubseteq \rangle$ è un poset con elemento minimo $\bot \in X$ ed elemento massimo $\top \in X$, chiuso rispetto agli operatori binari di estremo superiore, indicato con $\sqcup$ e detto anche *least upper bound*, ed estremo inferiore, indicato con $\sqcap$ e detto anche *greatest lower bound*: per ogni $x_0, x_1 \in X$ si ha $x_0 \sqcup x_1 \in X$ e $x_0 \sqcap x_1 \in X$. Il reticolo si dice *completo* quando la chiusura si estende a sottoinsiemi arbitrari, ossia $\bigsqcup Y \in X$ e $\bigsqcap Y \in X$ per ogni $Y \subseteq X$.

Un esempio canonico di reticolo completo è il *powerset lattice* $\langle \wp(S), \subseteq, \cup, \cap, \emptyset, S \rangle$, indotto da un insieme S mediante l'inclusione insiemistica. Nel seguito della tesi il *powerset lattice* interviene come strumento per descrivere insiemi di configurazioni concrete di un programma.

Definizione 3.2.3 (Catena e condizione di catena ascendente)**.** Dato un poset $\langle X, \sqsubseteq \rangle$, una catena $C \subseteq X$ è un sottoinsieme totalmente ordinato di X; il poset si dice *completo* (*cpo*) se ogni catena ammette estremo superiore $\bigsqcup C \in X$.

Una catena ascendente è una successione $x_0, \ldots, x_i, x_{i+1}, \ldots$ tale che $i \leq j$ implica $x_i \sqsubseteq x_j$. Il poset soddisfa la condizione di catena ascendente, abbreviata ACC, se per ogni catena ascendente esiste $k \in \mathbb{N}$ tale che $x_j = x_k$ per ogni $j \geq k$. Ogni poset ACC è in particolare un *cpo*.

La proprietà ACC ha rilevanza pratica diretta: quando il dominio astratto di un'analisi la soddisfa, il calcolo di punto fisso termina in un numero finito di iterazioni senza ricorrere a operatori di *widening*, come precisato nella Sezione 3.2.4.

Definizione 3.2.4 (Monotonia e continuità)**.** Una funzione $f\colon \langle X, \sqsubseteq\rangle \to \langle X, \sqsubseteq\rangle$ si dice *monotona* se $\forall x, y \in X \mid x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$. Se $\langle X, \sqsubseteq\rangle$ è un *cpo*, la funzione è *continua* quando preserva l'estremo superiore delle catene, ossia $f(\bigsqcup C) = \bigsqcup_{x_i \in C} f(x_i)$ per ogni catena C.

Definizione 3.2.5 (Iterata e punto fisso)**.** La n-esima iterata f^n di $f\colon \langle X, \sqsubseteq \rangle \to \langle X, \sqsubseteq\rangle$ a partire da un punto $x_0 \in X$ è definita da $f^0 = x_0$ e $f^{n+1} = f(f^n)$. Le iterate di una funzione monotona a partire da $\bot$ formano una catena ascendente. Un elemento $x \in X$ tale che $x = f(x)$ è detto punto fisso di f, ed è il minimo punto fisso, indicato con $\operatorname{lfp} f$, se $x \sqsubseteq y$ per ogni $y \in X$ con $y = f(y)$.

3.2.2 Semantica concreta e semantica astratta

La semantica concreta di un programma viene classicamente definita come il minimo punto fisso $\operatorname{lfp} f$ di una funzione continua $f\colon C \to C$ su un reticolo completo $\langle C, \sqsubseteq, \sqcup, \sqcap, \bot, \top\rangle$, detto dominio di calcolo concreto. Per il teorema di Kleene, tale punto fisso si calcola come estremo superiore della catena delle sue iterate a partire da $\bot$:

$$\operatorname{lfp} f = \bigsqcup_{n \in \mathbb{N}} f^n. \tag{3.1}$$

In generale il limite in Eq. (3.1) non è calcolabile in un numero finito di passi, poiché la catena può non stabilizzarsi. Per ovviare a questa difficoltà si introduce una funzione semantica astratta $g\colon A \to A$, monotona e definita su un dominio astratto $\langle A, \sqsubseteq^\sharp\rangle$ con elemento minimo $\bot^\sharp \in A$, che sovra-approssima correttamente f. Detta $\gamma\colon A \to C$ la funzione di concretizzazione che associa a ciascun elemento astratto l'insieme degli stati concreti che esso descrive, g si dice *sound* rispetto a f se soddisfa la condizione

$$\forall a \in A \mid f(\gamma(a)) \sqsubseteq \gamma(g(a)). \tag{3.2}$$

Le iterate di g a partire da $\bot^\sharp$ formano una catena ascendente i cui elementi approssimano correttamente le corrispondenti iterate concrete, ossia $f^n \sqsubseteq \gamma(g^n)$

per ogni $n \in \mathbb{N}$. Il dominio astratto non è obbligato a essere un reticolo completo: se $\langle A, \sqsubseteq^\sharp \rangle$ soddisfa la condizione ACC, la catena delle iterate converge in un numero finito $k \in \mathbb{N}$ di passi a un elemento $g^k \in A$ che sovra-approssima correttamente il minimo punto fisso concreto [3].

3.2.3 Connessioni di Galois

La formulazione originale del framework [9, 10] si appoggia alla nozione di *connessione di Galois*, che esprime in modo esatto la corrispondenza fra dominio concreto e dominio astratto.

Definizione 3.2.6 (Connessione di Galois)**.** Siano $\langle C, \leq \rangle$ e $\langle A, \sqsubseteq \rangle$ due poset. Una coppia di funzioni monotone $\alpha \colon \langle C, \leq \rangle \to \langle A, \sqsubseteq \rangle$ e $\gamma \colon \langle A, \sqsubseteq \rangle \to \langle C, \leq \rangle$ forma una connessione di Galois se

$$\forall x \in C, \forall y \in A \mid \alpha(x) \sqsubseteq y \iff x \leq \gamma(y).$$

Una connessione di Galois richiede l'esistenza di un'astrazione ottimale, costituita da $\alpha(x)$ per ogni elemento concreto x, e questa condizione è spesso troppo restrittiva nella pratica. L'interpretazione astratta è stata pertanto riformulata anche sotto ipotesi più deboli: la sola monotonia della funzione di concretizzazione $\gamma \colon \langle A, \sqsubseteq \rangle \to \langle C, \leq \rangle$ è sufficiente a stabilire una relazione di correttezza, ossia di *soundness*, senza richiedere l'esistenza dell'aggiunta α. Nella presente tesi si adotta la formulazione basata sulla sola γ monotona poiché le analisi presentate nei capitoli successivi non richiedono un'astrazione ottimale ma soltanto la correttezza delle approssimazioni prodotte [3].

3.2.4 Calcolo di punto fisso

Il calcolo effettivo della semantica astratta di un programma si riduce, nella pratica, alla determinazione del minimo punto fisso della funzione di trasferimento astratta g. Per il teorema di Kleene applicato al dominio astratto, tale punto fisso si ottiene come estremo superiore della catena ascendente delle iterate

$$\operatorname{lfp} g = \bigsqcup_{n \in \mathbb{N}} g^n \quad \text{con} \quad g^0 = \bot^\sharp,\ g^{n+1} = g(g^n). \tag{3.3}$$

Se il dominio astratto $\langle A, \sqsubseteq^\sharp \rangle$ soddisfa la condizione ACC, la catena si stabilizza dopo un numero finito $k \in \mathbb{N}$ di iterazioni e il valore g^k rappresenta esattamente il minimo punto fisso astratto.

Quando il dominio non soddisfa la condizione ACC, le iterate possono crescere indefinitamente senza convergere in tempo finito. In tali casi l'analisi ricorre a operatori di *widening*, che sostituiscono l'estremo superiore in alcuni

punti della catena con approssimazioni più grossolane ma capaci di forzare la convergenza. Nelle analisi presentate nei capitoli successivi i domini astratti utilizzati per il bytecode EVM soddisfano tutti la condizione ACC, e il calcolo di punto fisso converge in un numero finito di passi senza necessità di *widening* [3].

3.3 Grafi di flusso di controllo

La struttura di controllo di un programma è descritta da un grafo di flusso di controllo, o *control-flow graph*, di seguito indicato con l'acronimo CFG.

Definizione 3.3.1 (Grafo di flusso di controllo)**.** Dato un programma $\mathtt{P}$, un grafo di flusso di controllo di $\mathtt{P}$ è un grafo diretto $\mathtt{G_P} = (N, E)$ in cui i nodi N corrispondono agli indirizzi delle istruzioni di $\mathtt{P}$ e gli archi $E \subseteq N \times N$ descrivono le possibili transizioni di controllo fra le istruzioni. Tutti i costrutti sintattici che formano cicli, diramazioni e salti arbitrari sono codificati direttamente nella struttura del grafo [3].

Per impieghi di sicurezza è essenziale che il CFG contenga almeno tutti gli archi effettivamente percorribili a tempo di esecuzione, in modo che ogni analisi che ragiona sulla struttura del grafo non perda percorsi di esecuzione critici. Tale requisito si formalizza nella nozione di CFG *sound*. Si denotano con $\mathbb{S}$ l'insieme degli operand stack di un programma EVM, con $[\![\cdot]\!]$ la semantica concreta di un'istruzione, e con $\Xi^{\ell}(\mathtt{P}, \langle s, \ell_0 \rangle)$ l'esecuzione parziale del programma $\mathtt{P}$ a partire dallo stato $\langle s, \ell_0 \rangle$ fino al raggiungimento dell'istruzione in posizione ℓ.

Definizione 3.3.2 (CFG sound)**.** Dato un programma EVM $\mathtt{P}$ la cui prima etichetta è ℓ_0, un grafo $\mathtt{G_P} = (N, E)$ è un CFG *sound* di $\mathtt{P}$ se, per ogni opcode $\mathtt{op}$ in $\mathtt{P}$, si ha $\ell \in N$ e

$$\left\{ \ell \to \ell' \;\middle|\; \exists s, s' \in \mathbb{S} \mid \Xi^{\ell}(\mathtt{P}, \langle [], \ell_0 \rangle) = \langle s, \ell \rangle \wedge [\![\mathtt{op}]\!]\langle s, \ell \rangle = \langle s', \ell' \rangle \right\} \subseteq E. \quad (3.4)$$

In altri termini, ogni opcode è connesso a ciascuno dei possibili successori che può avere durante l'esecuzione.

La condizione in Eq. (3.4) richiede di conoscere, per ciascun opcode, l'insieme delle possibili configurazioni di stack che lo raggiungono, informazione che non è direttamente disponibile nel bytecode e deve essere ricostruita tramite analisi statica. Il Capitolo 4 descrive l'algoritmo iterativo con cui EVMLiSA [3] costruisce un CFG *sound* per un arbitrario programma EVM a partire da un'approssimazione astratta degli stack che raggiungono ciascun opcode.

3.4 Analisi taint

L'analisi *taint* è una tecnica di analisi statica derivata dall'analisi di flusso dei dati, detta anche *data-flow analysis* [13, 37]. L'obiettivo è determinare se informazioni sensibili originate da un insieme di istruzioni, dette *source*, possano raggiungere punti critici del programma, detti *sink*, percorrendo catene di propagazione lungo le operazioni del programma. Le variabili si ripartiscono in due insiemi disgiunti: quelle *tainted*, che contengono informazioni provenienti dalle *source*, e quelle *clean*, che ne sono prive. Un percorso di propagazione che conduce un valore *tainted* a un *sink* senza essere intercettato da un opportuno controllo, detto *sanitizer*, segnala un potenziale flusso informativo indesiderato.

L'analisi *taint* è consolidata per linguaggi di alto livello. Il *bytecode* EVM richiede però un trattamento specifico: l'architettura basata su *stack*, priva di variabili nominate, e i flussi di controllo dinamici non consentono di applicare direttamente le tecniche pensate per linguaggi ad alto livello. Il framework sviluppato nel presente lavoro si costruisce sull'analisi *taint* di EVMLiSA [3] e la specializza, tramite un'estensione relazionale, al rilevamento dell'*Access Control Incompleteness* nei *cross-chain bridge*.

3.4.1 Source, propagazione e sink

L'analisi *taint* si fonda sull'interpretazione astratta [9, 10]: ogni valore manipolato dal programma è approssimato da un elemento del dominio astratto *taint*, che registra se il valore sia di provenienza sensibile o meno, indipendentemente dal valore concreto. Il dominio è definito come segue.

Definizione 3.4.1 (Dominio taint)**.** Il dominio astratto *taint* è la struttura

$$\mathcal{T} \triangleq \langle \{\bot_{\mathcal{T}}, \mathsf{T}, \mathsf{C}, \top_{\mathcal{T}}\}, \leq_{\mathcal{T}}, \sqcap_{\mathcal{T}}, \sqcup_{\mathcal{T}}, \top_{\mathcal{T}}, \bot_{\mathcal{T}} \rangle, \tag{3.5}$$

in cui T e C indicano rispettivamente che un valore è *tainted* o *clean*, mentre $\top_{\mathcal{T}}$ e $\bot_{\mathcal{T}}$ sono gli elementi di massimo e di minimo, con il primo che rappresenta un valore sconosciuto e il secondo uno stato di errore. La relazione d'ordine parziale $\leq_{\mathcal{T}}$ è definita da

$$\forall t \in \mathcal{T} \mid \bot_{\mathcal{T}} \leq_{\mathcal{T}} t,\ t \leq_{\mathcal{T}} \top_{\mathcal{T}}, \quad \mathsf{T} \not\leq_{\mathcal{T}} \mathsf{C},\ \mathsf{C} \not\leq_{\mathcal{T}} \mathsf{T}. \tag{3.6}$$

La struttura reticolare del dominio è illustrata in Figura 3.1. Gli elementi T e C sono incomparabili fra loro e si collocano a metà del diagramma di Hasse, con $\top_{\mathcal{T}}$ sopra di essi e $\bot_{\mathcal{T}}$ sotto.

I concetti di *source*, *sink* e *sanitizer* non sono codificati nel dominio, ma costituiscono parametri forniti dall'analisi specifica che istanzia il framework.

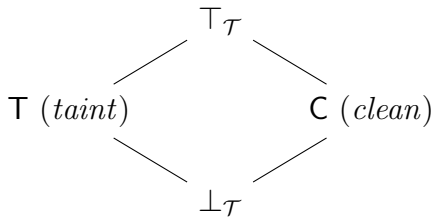


Figura 3.1: Struttura reticolare del dominio astratto *taint*.

Una *source* è un punto del programma la cui esecuzione introduce un nuovo valore *tainted*; un *sink* è un punto in cui la presenza di un valore *tainted* denota un potenziale flusso informativo indesiderato; un *sanitizer* è un percorso di controllo la cui esecuzione garantisce l'assenza di quel flusso. Le funzioni di trasferimento astratte propagano lo stato *taint* attraverso le istruzioni del programma: se un operando è *tainted*, anche il risultato dell'operazione lo è, secondo le regole di propagazione discusse nella Sezione 3.4.2.

La specializzazione di $\mathcal{T}$ a un'architettura basata su stack, con funzioni di trasferimento per ciascuna istruzione della macchina virtuale, e la successiva istanziazione dell'analisi *taint* relazionale ai *cross-chain bridge* con *source*, *sink* e *sanitizer* specifici, sono presentate nel Capitolo 5 come applicazione del framework sul CFG *sound* costruito nel Capitolo 4.

3.4.2 Analisi taint relazionale

L'analisi *taint* di base è sufficiente a rilevare la presenza di un flusso informativo fra *source* e *sink*, ma perde ogni distinzione fra le diverse *source* che possono aver introdotto un valore *tainted*. Questo limite è rilevante quando il *checker* deve stabilire se un determinato argomento di un *sink* sia riconducibile a una specifica *source* e se sia stato sottoposto a un controllo appropriato: qualsiasi *sanitizer* su un percorso condiviso maschererebbe il flusso anche quando non interviene su quella particolare provenienza.

Per preservare l'informazione di origine si estende il dominio *taint* a una variante relazionale, che associa a ogni valore *tainted* l'insieme dei *program point* che lo hanno introdotto nel corso dell'analisi; questa corrispondenza è preservata anche dopo la propagazione attraverso operazioni aritmetiche o logiche. La definizione formale del dominio è la seguente.

Definizione 3.4.2 (Dominio taint relazionale)**.** Il dominio astratto *taint* relazionale è la struttura

$$\mathcal{T}^{\mathsf{pp}} \triangleq \langle \{\bot_{\mathcal{T}}, \top_{\mathcal{T}}\} \cup (\{\mathsf{T}, \mathsf{C}\} \times \wp(\mathbb{N})), \leq_{\mathcal{T}}^{\mathsf{pp}}, \sqcap_{\mathcal{T}}^{\mathsf{pp}}, \sqcup_{\mathcal{T}}^{\mathsf{pp}}, \top_{\mathcal{T}}, \bot_{\mathcal{T}} \rangle, \tag{3.7}$$

in cui un elemento astratto è $\bot_{\mathcal{T}}$, $\top_{\mathcal{T}}$, oppure una coppia (τ, P) con $\tau \in \{\mathsf{T}, \mathsf{C}\}$ e $P \subseteq \mathbb{N}$ insieme dei *program point* di origine. L'ordine parziale è definito da

$$\bot_{\mathcal{T}} \leq^{\mathsf{pp}}_{\mathcal{T}} t \leq^{\mathsf{pp}}_{\mathcal{T}} \top_{\mathcal{T}} \text{ per ogni } t, \quad (\tau, P) \leq^{\mathsf{pp}}_{\mathcal{T}} (\tau', P') \iff \tau = \tau' \wedge P \subseteq P'. \tag{3.8}$$

Gli operatori di estremo superiore e inferiore agiscono componente per componente: quando la componente τ coincide, P è rispettivamente unione e intersezione dei corrispondenti insiemi di *program point*.

I *program point* sono registrati soltanto per i valori *tainted*: ogni introduzione di un valore *tainted* lo etichetta con il proprio *program point*, mentre i valori *clean* seguono la semantica del dominio base e non riportano informazioni di provenienza. Quando due valori *tainted* partecipano a un'operazione, il risultato resta *tainted* e riporta l'unione dei rispettivi insiemi di *program point*. Per ciascun percorso di esecuzione è dunque possibile risalire al valore che raggiunge un determinato punto e all'insieme delle *source* che hanno contribuito al suo stato *tainted*.

Formalmente, per un nuovo valore *tainted* introdotto al *program point* pp si pone

$$\mathsf{new\_taint}(pp) \triangleq (\mathsf{T}, \{pp\}), \tag{3.9}$$

mentre per un'operazione binaria $\odot$ con operandi $t_1, t_2 \in \mathcal{T}^{\mathsf{pp}}$ la propagazione del *taint* è definita come

$$\mathsf{taint}(t_1 \odot t_2) \triangleq \begin{cases} \top_{\mathcal{T}} & \text{se } t_1 = \top_{\mathcal{T}} \text{ oppure } t_2 = \top_{\mathcal{T}}; \\ \bot_{\mathcal{T}} & \text{se } t_1 = \bot_{\mathcal{T}} \text{ oppure } t_2 = \bot_{\mathcal{T}}; \\ (\mathsf{T}, \pi_2(t_1) \cup \pi_2(t_2)) & \text{se } \pi_1(t_1) = \mathsf{T} \text{ oppure } \pi_1(t_2) = \mathsf{T}; \\ \mathsf{C} & \text{altrimenti,} \end{cases} \tag{3.10}$$

dove π_1 e π_2 denotano rispettivamente la proiezione sulla prima e sulla seconda componente di una coppia $(\tau, P) \in \{\mathsf{T}, \mathsf{C}\} \times \wp(\mathbb{N})$. Il risultato di un'operazione è *tainted* ogni volta che almeno uno degli operandi lo è, e in tal caso eredita l'unione dei loro *program point* di origine. La definizione si estende alle operazioni che consumano più di due operandi: per un'operazione n-aria il risultato è *tainted* se almeno un operando lo è, e il suo insieme di *program point* è l'unione degli insiemi dei *program point* degli operandi *tainted*.

Semantica astratta. La semantica astratta di un'operazione nel dominio $\mathcal{T}^{\mathsf{pp}}$ è definita dalle equazioni Eq. (3.9) e Eq. (3.10). Ogni punto del programma che introduce un valore controllato da una *source* esterna produce un elemento $\mathsf{new\_taint}(pp)$, etichettato con il proprio *program point*. Le operazioni che non modificano il contenuto informativo di un valore propagano l'elemento

astratto invariato, mentre le operazioni che combinano più valori calcolano il risultato secondo la funzione taint di Eq. (3.10). I valori introdotti da punti non influenzabili dall'esterno sono modellati da C. I ruoli di *source*, *sink* e *sanitizer* restano quelli introdotti nella Sezione 3.4.1 e sono parametri forniti dall'istanza di analisi che impiega $\mathcal{T}^{\mathsf{pp}}$; il dominio è quindi applicabile a classi di vulnerabilità differenti senza modificarne la struttura.

Il Capitolo 5 presenta l'istanziazione di questo dominio relazionale ai *cross-chain bridge*, con la definizione esplicita di *source*, *sink* e *sanitizer* per il rilevamento dell'*Access Control Incompleteness*.

Capitolo 4

Costruzione di CFG Sound per Bytecode EVM

La Definizione 3.3.2 e la condizione (3.4) del Capitolo 3 formalizzano la nozione di CFG *sound* rispetto alla semantica concreta: un grafo è *sound* se e solo se ogni percorso eseguito dal contratto in esecuzione reale corrisponde a un percorso nel grafo. Soddisfare tale condizione richiede di conoscere, per ciascun *opcode*, l'insieme degli *stack* che lo raggiungono durante tutte le possibili esecuzioni, informazione che non è direttamente ricavabile dal bytecode poiché le destinazioni dei salti sono calcolate a *runtime* a partire dai valori presenti sullo *stack*.

Il contributo principale di questo capitolo è un dominio astratto parametrico organizzato come gerarchia a tre livelli. Il primo livello approssima i singoli valori di *stack* tramite k-interi. Il secondo livello aggrega tali valori in *stack* astratti di altezza massima h, che descrivono una singola esecuzione del contratto. Il terzo livello raccoglie fino a l *stack* astratti in un insieme, raccogliendo l'unione degli stati prodotti dai diversi percorsi di esecuzione. Da questa gerarchia si ottiene una sovra-approssimazione computabile degli *stack* raggiunti da ciascun *opcode*, sulla quale si fonda la risoluzione iterativa dei salti orfani, ossia di quei salti la cui destinazione dipende da espressioni aritmetiche calcolate sullo *stack*, secondo il modello introdotto nella Sezione 2.3.3.

EVMLiSA[1] è implementato come *front-end* di LiSA,[2] una libreria di analisi statica basata sull'interpretazione astratta che fornisce il motore di calcolo, ossia l'*abstract interpreter* che propaga le proprietà astratte lungo il CFG fino al raggiungimento di un punto fisso, e le infrastrutture su cui si innestano gli algoritmi di costruzione del CFG [3].

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

La Sezione 4.1 formalizza la semantica concreta dell'EVM, definendo programmi, stati e funzione di transizione. La Sezione 4.2 introduce il dominio astratto gerarchico, dalle operazioni sui k-interi fino alla struttura reticolare degli insiemi di *stack* astratti. La Sezione 4.3 presenta gli algoritmi iterativi BUILDCFG e JUMPSOLVER, responsabili della costruzione del grafo e della risoluzione dei salti. La Sezione 4.4 raccoglie le proprietà formali, dalla chiusura *additive closure condition* alla *soundness* del CFG prodotto. Il CFG così ottenuto, annotato con gli *stack* astratti calcolati durante l'analisi, è il punto di partenza dell'analisi *taint* relazionale per il rilevamento dell'*Access Control Incompleteness* presentata nel Capitolo 5.

4.1 Semantica concreta dell'EVM

La costruzione di un CFG *sound* per bytecode EVM richiede una base formale precisa che definisca che cosa sia uno stato di programma valido e come le istruzioni lo trasformino. Si definiscono il concetto di programma, di indirizzo e di stato di programma, la semantica delle istruzioni non di salto e quella delle istruzioni condizionali e incondizionali, e la funzione di esecuzione complessiva. Tale semantica costituisce il riferimento formale rispetto al quale il dominio astratto della Sezione 4.2 è dimostrato *sound*. Per la descrizione in prosa degli *opcode* e dell'architettura della macchina virtuale si rimanda alle Sezioni 2.3.1 e 2.3.2 del Capitolo 2.

4.1.1 Programmi, indirizzi e stati

Un programma EVM è una sequenza di *opcode* $\mathsf{op} \in \mathbb{O}$, dove $\mathbb{O}$ denota l'insieme di tutti gli *opcode* ammessi dalla macchina virtuale. Una volta distribuito sulla blockchain, ciascun *opcode* viene memorizzato a un indirizzo ℓ univoco, che lo identifica e consente ai salti di raggiungerlo, e ha quindi la funzione di *program counter*. Due *opcode* distinti non possono condividere lo stesso indirizzo.

Definizione 4.1.1 (Indirizzi e funzioni di navigazione)**.** L'insieme di tutti gli indirizzi possibili è

$$\mathbb{L} \subseteq \mathbb{N} \cup \{\epsilon\},$$

dove ϵ è l'etichetta vuota, ossia un'etichetta che non punta ad alcun *opcode* e viene impiegata per segnalare la terminazione dell'esecuzione. Dato un indirizzo $\ell \in \mathbb{N}$, si denota con $\Pi(\ell)$ l'*opcode* corrispondente a quell'indirizzo, e con $\mathsf{next}(\ell)$ l'indirizzo dell'*opcode* che segue $\Pi(\ell)$ nel testo del programma. Quando ℓ è l'ultimo *opcode* del programma, si pone $\mathsf{next}(\ell) = \epsilon$.

La notazione $\mathsf{op}^{\rho}_{\delta}$ denota un *opcode* op all'indirizzo ℓ, con $\delta \in \mathbb{N}$ valori prelevati dallo *stack* e $\rho \in \mathbb{N}$ valori inseriti, includendo nell'*opcode* stesso gli

eventuali parametri immediati, come avviene ad esempio per `PUSH1`, che porta con sé il byte da impilare.

Definizione 4.1.2 (Stack operandi e stati di programma)**.** Uno *stack* operandi è una lista LIFO $s = [z_0, \ldots, z_{n-1}]$ di lunghezza $|s| = n \leq 1024$, con elementi $z_i \in \mathbb{Z}$. Poiché gli elementi dello *stack* sono parole a 256 bit, possono essere trattati come interi senza perdita di generalità. L'insieme $\mathbb{S}$ di tutti gli *stack* operandi include anche lo *stack* non valido $\bot_{\mathbb{S}}$, che rappresenta il risultato di computazioni erronee. Uno stato di programma è una coppia $\langle s, \ell \rangle \in \mathbb{M}$, dove $s \in \mathbb{S}$ è lo *stack* corrente e $\ell \in \mathbb{L}$ è l'indirizzo dell'*opcode* da eseguire. Lo stato speciale

$$\bot \triangleq \langle \bot_{\mathbb{S}}, \epsilon \rangle \in \mathbb{M}$$

denota un errore di esecuzione, corrispondente alla situazione in cui la macchina virtuale interrompe l'esecuzione sollevando un'eccezione.

4.1.2 Semantica delle istruzioni non di salto

La semantica di un *opcode* è definita tramite la funzione $[\![\mathtt{op}]\!] \colon \mathbb{M} \to \mathbb{M}$, che a uno stato di ingresso in $\mathbb{M}$ associa lo stato risultante dopo l'esecuzione di `op`. Per qualsiasi *opcode* vale $[\![\mathtt{op}]\!]\bot = \bot$: uno stato di errore non può progredire. Analogamente, $[\![\mathtt{op}]\!]\langle s, \ell' \rangle = \bot$ se $\ell \neq \ell'$, ossia la semantica è definita solo per gli *opcode* effettivamente collocati all'indirizzo considerato.

Tutte le istruzioni che non modificano il flusso di controllo, ad eccezione degli *opcode* di arresto, seguono lo schema uniforme descritto di seguito. Sia $s = [z_0, \ldots, z_{n-1}]$ e sia $s' = [z_0, \ldots, z_{n-1-\delta}, \dot{z}_0, \ldots, \dot{z}_{\rho-1}]$ lo *stack* risultante dopo il prelievo dei δ valori in cima e l'inserimento dei ρ valori calcolati. La semantica generica è:

$$[\![\mathtt{op}^{\rho}_{\delta}]\!]\langle s, \ell \rangle = \begin{cases} \bot & \text{se } |s| < \delta \text{ oppure } |s| - \delta + \rho > 1024; \\ \langle s',\, \mathsf{next}(\ell) \rangle & \text{altrimenti.} \end{cases} \tag{4.1}$$

La funzione ritorna $\bot$ in due situazioni eccezionali: lo *stack underflow*, che si verifica quando il numero di elementi disponibili è inferiore a δ, e lo *stack overflow*, che si verifica quando la dimensione dello *stack* dopo l'esecuzione supererebbe il limite di 1024 elementi. Negli altri casi, i δ valori in cima allo *stack* vengono prelevati e impiegati per calcolare ρ nuovi valori $\dot{z}_0, \ldots, \dot{z}_{\rho-1}$, specifici dell'*opcode*; questi vengono quindi inseriti in cima allo *stack* e il *program counter* avanza a $\mathsf{next}(\ell)$. Il calcolo dei valori risultanti $\dot{z}_i$ è dipendente dall'*opcode* e non influisce sul flusso di controllo, pertanto non viene formalizzato ulteriormente.

Esempio 4.1.1 (Semantica di `ADD`)**.** L'*opcode* `ADD` preleva due valori dallo *stack*, ne calcola la somma e la inserisce in cima: $\delta = 2$, $\rho = 1$. Sia $s = [z_0, z_1, z_2]$ e sia ℓ l'indirizzo dell'istruzione. Poiché $|s| = 3 \geq 2$ e $3-2+1 = 2 \leq 1024$, l'esecuzione restituisce lo stato $\langle [z_0, z_1 + z_2], \mathsf{next}(\ell)\rangle$. Se invece $s = [z_0]$, si ha $|s| = 1 < 2$, quindi la semantica restituisce $\bot$ per *stack underflow*.

4.1.3 Semantica di JUMP e JUMPI

Le istruzioni di salto sono il principale ostacolo alla costruzione statica del CFG: come discusso nella Sezione 2.3.3, la destinazione di un salto viene calcolata a tempo di esecuzione a partire da valori presenti sullo *stack*, e non è in generale determinabile senza un'analisi del flusso di dati. La semantica concreta di `JUMP` e `JUMPI` è dunque necessaria per chiarire quali stati siano raggiungibili e quali debbano invece produrre un errore.

Salto incondizionale. La semantica dell'istruzione di salto incondizionale è definita come segue, con $s = [z_0, \ldots, z_{n-1}]$:

$$[\![\texttt{JUMP}^0_1]\!]\langle s, \ell\rangle = \begin{cases} \bot & \text{se } |s| < 1 \text{ oppure} \\ & \quad \Pi(z_{n-1}) \neq \texttt{JUMPDEST}; \\ \langle [z_0, \ldots, z_{n-2}],\ z_{n-1}\rangle & \text{altrimenti.} \end{cases} \tag{4.2}$$

L'istruzione preleva il valore in cima allo *stack*, z_{n-1}, e lo interpreta come indirizzo della prossima istruzione da eseguire. Se tale indirizzo non ospita un *opcode* `JUMPDEST`, la macchina virtuale solleva un'eccezione e l'esecuzione si arresta con $\bot$; analogamente se lo *stack* è vuoto. Negli altri casi, z_{n-1} viene rimosso dallo *stack* e il contatore di programma viene impostato a quel valore.

Salto condizionale. La semantica dell'istruzione di salto condizionale `JUMPI` è definita come segue, con $s = [z_0, \ldots, z_{n-1}]$ e $\bar{s} = [z_0, \ldots, z_{n-3}]$:

$$[\![\texttt{JUMPI}^0_2]\!]\langle s, \ell\rangle = \begin{cases} \bot & \text{se } |s| < 2 \text{ oppure} \\ & \quad (z_{n-2} \neq 0 \wedge \Pi(z_{n-1}) \neq \texttt{JUMPDEST}); \\ \langle \bar{s},\ z_{n-1}\rangle & \text{se } z_{n-2} \neq 0; \\ \langle \bar{s},\ \mathsf{next}(\ell)\rangle & \text{se } z_{n-2} = 0. \end{cases} \tag{4.3}$$

L'istruzione preleva due valori dallo *stack*: z_{n-2} è la condizione di ramificazione e z_{n-1} è la destinazione del salto, mentre $\bar{s}$ denota lo *stack* residuo dopo la rimozione di entrambi. Se la condizione è diversa da zero, l'esecuzione prosegue dall'indirizzo z_{n-1}, a patto che vi si trovi un `JUMPDEST`; in caso contrario si ritorna $\bot$. Se invece la condizione è zero, il salto non viene effettuato: z_{n-1}

viene comunque rimosso dallo *stack* senza che il suo valore venga verificato, e il *program counter* avanza al successore naturale $\mathsf{next}(\ell)$. Il controllo su `JUMPDEST` viene quindi eseguito solo quando il salto è effettivamente preso. Nell'analisi statica ciò significa che un valore potenzialmente non valido come destinazione di salto non costituisce necessariamente un errore, purché il ramo corrispondente non sia mai percorso durante l'esecuzione concreta.

4.1.4 Esecuzione del programma

Le semantiche delle singole istruzioni si compongono nella funzione di esecuzione del programma, che descrive la computazione a partire da uno stato iniziale.

Definizione 4.1.3 (Funzione di esecuzione Ξ)**.** Dato un programma `P` composto da $k+1$ istruzioni, la funzione di esecuzione $\Xi\colon \mathbb{O}^\star \times \mathbb{M} \to \mathbb{M}$ è definita ricorsivamente come:

$$\Xi(\mathtt{P}, \langle s, \ell \rangle) = \begin{cases} \langle [], \epsilon \rangle & \text{se } \ell = \epsilon \text{ oppure } \Pi(\ell) \in \{\mathtt{STOP}, \mathtt{SELFDESTRUCT}\}; \\ \langle s, \epsilon \rangle & \text{se } \Pi(\ell) \in \{\mathtt{RETURN}, \mathtt{REVERT}\}; \\ \Xi(\mathtt{P}, \langle s', \ell' \rangle) & \text{se } [\![\Pi(\ell)]\!]\langle s, \ell \rangle = \langle s', \ell' \rangle \neq \bot; \\ \bot & \text{altrimenti.} \end{cases} \tag{4.4}$$

I primi due casi modellano la terminazione del programma: il primo copre la terminazione implicita, che avviene quando il contatore di programma raggiunge ϵ, e la terminazione esplicita tramite `STOP` o `SELFDESTRUCT`, che non restituiscono dati; il secondo copre `RETURN` e `REVERT`, che terminano l'esecuzione restituendo al chiamante i dati ancora presenti sullo *stack*. Il terzo caso descrive un singolo passo di esecuzione non eccezionale: se la semantica dell'istruzione corrente produce uno stato valido, l'esecuzione riprende da quello stato. Il quarto caso cattura ogni altra situazione, incluse quelle in cui la semantica di un'istruzione restituisce $\bot$, arrestando l'esecuzione con un errore.

Per gli scopi dell'analisi statica è utile disporre anche di una variante parziale di Ξ, che esegue il programma soltanto fino al raggiungimento di un indirizzo prestabilito.

Definizione 4.1.4 (Funzione di esecuzione parziale $\Xi^{\overline{\ell}}$)**.** Dato un indirizzo $\overline{\ell} \neq \epsilon$, la funzione di esecuzione parziale $\Xi^{\overline{\ell}}\colon \mathbb{O}^\star \times \mathbb{M} \to \mathbb{M}$ è definita ricorsivamente

come:

$$\Xi^{\overline{\ell}}(\mathtt{P}, \langle s, \ell\rangle) = \begin{cases} \langle s, \ell\rangle & \text{se } \ell = \overline{\ell}; \\ \bot & \text{se } \ell = \epsilon \text{ oppure } \Pi(\ell) \in \{\mathtt{STOP}, \mathtt{SELFDESTRUCT}\}; \\ \bot & \text{se } \Pi(\ell) \in \{\mathtt{RETURN}, \mathtt{REVERT}\}; \\ \Xi^{\overline{\ell}}(\mathtt{P}, \langle s', \ell'\rangle) & \text{se } [\![\Pi(\ell)]\!]\langle s, \ell\rangle = \langle s', \ell'\rangle \neq \bot; \\ \bot & \text{altrimenti.} \end{cases} \tag{4.5}$$

Lo scopo di $\Xi^{\overline{\ell}}$ è estrarre gli stati che possono raggiungere $\overline{\ell}$: se l'esecuzione raggiunge tale indirizzo, viene restituito lo stato corrente; se invece il programma termina, con o senza errore, prima di raggiungerlo, viene restituito $\bot$, a indicare che $\overline{\ell}$ non è stato raggiunto in quella traiettoria di esecuzione. Questa funzione è impiegata direttamente nella definizione di CFG *sound* introdotta nella Sezione 3.3, e congiunge sul piano formale la semantica concreta e la costruzione del grafo.

Esempio 4.1.2 (Comportamento di Ξ e $\Xi^{\overline{\ell}}$)**.** Si consideri il frammento di programma:

$$\mathtt{P} = {}^{0}\mathtt{PUSH1\ 0x01}^{1}_{0} \quad {}^{1}\mathtt{PUSH1\ 0x02}^{1}_{0} \quad {}^{2}\mathtt{ADD}^{1}_{2} \quad {}^{3}\mathtt{JUMP}^{0}_{1}$$

L'esecuzione di $\Xi(\mathtt{P}, \langle[], 0\rangle)$ e di $\Xi^3(\mathtt{P}, \langle[], 0\rangle)$ procede in modo identico nei primi tre passi: l'*opcode* in posizione 0 inserisce 1 sullo *stack*, che diventa [1] con contatore $\mathtt{next}(0) = 1$; l'*opcode* in posizione 1 inserisce 2, portando lo *stack* a $[1, 2]$ con contatore $\mathtt{next}(1) = 2$; l'*opcode* in posizione 2 esegue la somma, lasciando lo *stack* a [3] con contatore $\mathtt{next}(2) = 3$. A questo punto le due funzioni si distinguono: Ξ^3 restituisce $\langle[3], 3\rangle$, poiché l'indirizzo $\overline{\ell} = 3$ è stato raggiunto; Ξ tenta invece di eseguire l'*opcode* `JUMP` in posizione 3 e, poiché $\Pi(3)$ non è un `JUMPDEST`, restituisce $\bot$.

4.2 Dominio astratto per gli stack EVM

Come stabilito dalla Definizione 3.3.2 e dalla condizione (3.4), la costruzione di un CFG *sound* per un programma EVM richiede di determinare, per ogni *opcode* `op` collocato all'indirizzo ℓ, l'insieme di tutti i successori raggiungibili a tempo di esecuzione. Occorre quindi approssimare l'insieme degli *stack* operandi che possono raggiungere ℓ durante una qualunque esecuzione del programma, poiché le istruzioni `JUMP` e `JUMPI` calcolano la destinazione a partire da un valore estratto dallo *stack*, non determinabile staticamente.

La procedura, descritta nelle sottosezioni seguenti, calcola una sovra approssimazione degli *stack* che raggiungono ogni *opcode*, impiega tali *stack* per

risolvere le destinazioni dei salti aggiungendo i corrispondenti archi al grafo, e ripete il procedimento fino al raggiungimento di un punto fisso. La correttezza della procedura, ossia la garanzia che nessun arco dell'esecuzione concreta venga omesso dal CFG prodotto, segue dalle proprietà del framework dell'interpretazione astratta descritte nella Sezione 3.2: la monotonia del dominio astratto e la convergenza al punto fisso garantiscono che la sovra-approssimazione calcolata includa tutti gli *stack* effettivamente raggiungibili a *runtime*.

Il dominio astratto è organizzato in una gerarchia a tre livelli, presentati nelle sottosezioni che seguono insieme alla semantica astratta delle istruzioni che opera su di essi.

4.2.1 k-interi

Il primo livello della gerarchia astratta introduce un insieme ampliato di valori per rappresentare i singoli elementi presenti sullo *stack*. Nella semantica concreta ogni elemento è un intero $z \in \mathbb{Z}$; nell'analisi astratta occorre poter rappresentare anche situazioni di incertezza parziale o totale sul valore di un elemento, nonché la situazione in cui uno slot dello *stack* astratto non corrisponde ad alcun elemento concreto.

Definizione 4.2.1 (k-interi)**.** L'insieme dei *k-interi* è

$$\mathbb{Z}^\sharp \triangleq \mathbb{Z} \cup \{\varnothing, \top_{\overline{\mathbb{Z}}}, \top_{\mathbb{Z}^\sharp}\},$$

dove i tre simboli speciali hanno il seguente significato:

- $\varnothing$ denota uno slot non inizializzato, ossia un elemento che non corrisponde ad alcun valore concreto sullo *stack*;
- $\top_{\overline{\mathbb{Z}}}$ denota un valore di cui si garantisce che non corrisponde all'indirizzo di un *opcode* `JUMPDEST` valido, pur essendo il valore numerico esatto sconosciuto;
- $\top_{\mathbb{Z}^\sharp}$ denota un valore completamente sconosciuto, che potrebbe o meno essere l'indirizzo di un `JUMPDEST`.

L'elemento $\top_{\overline{\mathbb{Z}}}$ è decisivo per la precisione dell'analisi: molti *opcode* producono valori che dipendono da quantità note solo a tempo di esecuzione, come l'orario del blocco o la quantità di gas disponibile, e che nella pratica non vengono mai usati come destinazioni di salto.[3] Distinguere $\top_{\overline{\mathbb{Z}}}$ da $\top_{\mathbb{Z}^\sharp}$ permet-

[3]Gli *opcode* che producono $\top_{\overline{\mathbb{Z}}}$ sono: `GAS`, `KECCAK256`, `CALLCODE`, `DIFFICULTY`, `ORIGIN`, `CALLER`, `CALLVALUE`, `CALLDATASIZE`, `CODESIZE`, `GASPRICE`, `RETURNDATASIZE`, `COINBASE`, `TIMESTAMP`, `NUMBER`, `GASLIMIT`, `CHAINID`, `SELFBALANCE`, `MSIZE`, `BASEFEE`, `BALANCE`, `CALLDATALOAD`, `EXTCODESIZE`, `EXTCODEHASH`, `BLOCKHASH`, `CREATE`, `CREATE2`, `CALL`, `DELEGATECALL`, `STATICCALL`.

te all'algoritmo di evitare l'aggiunta di archi spuri al grafo ogni volta che un valore prodotto da tali *opcode* raggiunge un'istruzione di salto.

Esempio 4.2.1 (Sequenza `TIMESTAMP JUMP`)**.** Si consideri il frammento di bytecode `TIMESTAMP JUMP`. L'*opcode* `TIMESTAMP` inserisce sullo *stack* il *timestamp* del blocco corrente: poiché tale valore dipende dal contesto di esecuzione e non può in alcun modo coincidere con un indirizzo di `JUMPDEST` significativo, la semantica astratta assegna a esso il valore $\top_{\overline{\mathbb{Z}}}$. Quando l'algoritmo incontra successivamente `JUMP`, il valore in cima allo *stack* astratto è $\top_{\overline{\mathbb{Z}}}$; grazie alla definizione di concretizzazione che segue, ciò implica che nessun indirizzo di `JUMPDEST` valido è raggiungibile, e pertanto nessun arco spurio viene aggiunto al CFG.

La funzione di concretizzazione $\dot{\gamma}$ mappa ciascun *k-intero* all'insieme degli interi concreti che esso rappresenta.

Definizione 4.2.2 (Concretizzazione dei *k-interi*)**.** Sia $\mathbb{J} = \{\ell \in \mathbb{L} \mid \Pi(\ell) = \texttt{JUMPDEST}\}$ l'insieme di tutti gli indirizzi del programma `P` che ospitano un *opcode* `JUMPDEST`. La funzione $\dot{\gamma} \colon \mathbb{Z}^\sharp \to \wp(\mathbb{Z})$ è definita come:

$$\dot{\gamma}(v) \triangleq \begin{cases} \emptyset, & \text{se } v = \varnothing; \\ \mathbb{Z} \setminus \mathbb{J}, & \text{se } v = \top_{\overline{\mathbb{Z}}}; \\ \mathbb{Z}, & \text{se } v = \top_{\mathbb{Z}^\sharp}; \\ \{v\}, & \text{altrimenti } (v \in \mathbb{Z}). \end{cases} \tag{4.6}$$

Il primo caso riflette il fatto che $\varnothing$ non rappresenta alcun valore concreto: lo slot è vuoto e nessun intero gli corrisponde. Il secondo caso cattura la garanzia offerta da $\top_{\overline{\mathbb{Z}}}$: il valore concreto esiste ed è un qualunque intero, ma si esclude che sia l'indirizzo di un `JUMPDEST`. Il terzo caso esprime l'assenza totale di informazione propria di $\top_{\mathbb{Z}^\sharp}$: il valore concreto può essere qualsiasi intero, inclusi gli indirizzi di `JUMPDEST`. Il quarto caso modella i valori concreti noti: l'unico intero rappresentato è v stesso.

4.2.2 Stack astratti di altezza h

Un singolo *k-intero* astrae un valore sullo *stack*, ma per ragionare sul comportamento di un *opcode* occorre astrarre l'intero *stack* operandi. Il secondo livello della gerarchia introduce gli *stack* astratti di altezza h, che mantengono traccia di al più h elementi in cima allo *stack*.

Definizione 4.2.3 (*Stack* astratti di altezza h)**.** Per ogni $h > 0$, l'insieme degli *stack* astratti di altezza h è:

$$\Sigma^h \triangleq \{[v_0, \ldots, v_{h-1}] \mid \forall i \in [0, h-1],\ v_i \in \mathbb{Z}^\sharp\} \cup \{\bot_{\Sigma^h}\},$$

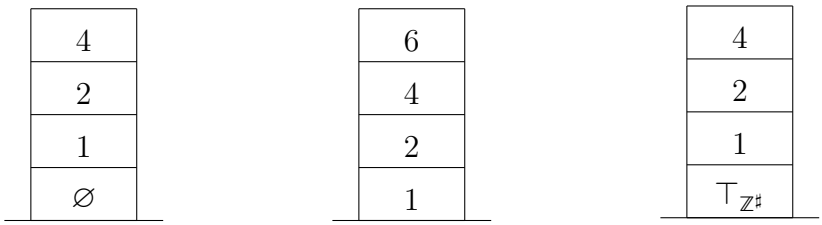


Figura 4.1: Esempi di *stack* astratti, elementi di Σ^4.

dove l'elemento più a destra, v_{h-1}, rappresenta la cima dello *stack* e $\bot_{\Sigma^h}$ è lo *stack* astratto invalido, che modella stati di errore.

Gli *stack* concreti con meno di h elementi vengono rappresentati riempiendo le posizioni inferiori con $\varnothing$: ad esempio, lo *stack* concreto $[1, 2, 3]$ in Σ^4 diventa $[\varnothing, 1, 2, 3]$, dove $\varnothing$ in posizione 0 segnala che lo *stack* concreto ha esattamente 3 elementi.

La Figura 4.1 illustra tre esempi di *stack* astratti in Σ^4. Lo *stack* in Figura 4.1a, pari a $[\varnothing, 1, 2, 4]$, rappresenta lo *stack* concreto di esattamente tre elementi la cui cima vale 4 e i cui due elementi inferiori valgono 2 e 1: la presenza di $\varnothing$ in fondo esclude l'esistenza di ulteriori elementi al di sotto. Lo *stack* in Figura 4.1b, pari a $[1, 2, 4, 6]$, descrive *stack* concreti di almeno quattro elementi, poiché nessun $\varnothing$ compare nella rappresentazione astratta: i quattro elementi tracciati sono quelli in cima, ma potrebbero esisterne altri non visibili nel dominio. Lo *stack* in Figura 4.1c, pari a $[\top_{\mathbb{Z}^\sharp}, 1, 2, 4]$, descrive *stack* concreti di almeno tre elementi: $\top_{\mathbb{Z}^\sharp}$ in fondo indica che possono esistere elementi aggiuntivi di valore ignoto, ma non si pone alcun vincolo su di essi. Il secondo *stack* è ottenuto eseguendo astrattamente `PUSH1 0x06` a partire dal primo: l'operazione inserisce il valore 6 in cima, causando lo scivolamento verso il basso di tutti gli elementi e la perdita dell'informazione sulla profondità precisa dello *stack*.

La funzione di concretizzazione $\overline{\gamma}$ trasforma uno *stack* astratto nell'insieme degli *stack* concreti che esso sovra-approssima. Per definirla formalmente si introduce il concetto di indice di fondo.

Definizione 4.2.4 (Concretizzazione degli *stack* astratti)**.** Dato uno *stack* astratto $\hat{\sigma} \in \Sigma^h$, $\hat{\sigma} = [v_0, \ldots, v_{h-1}]$, si definiscono:

$$k_\varnothing \triangleq \max\{k \in [0, h] \mid \forall i \in [0, k-1],\ v_i = \varnothing\}; \tag{4.7}$$

$$k_\top \triangleq \max\{k \in [0, h] \mid \forall i \in [0, k-1],\ v_i = \top_{\mathbb{Z}^\sharp}\}. \tag{4.8}$$

La funzione ausiliaria $\overline{\gamma}' \colon \Sigma^h \to \wp(\mathbb{S})$ è definita come:

$$\overline{\gamma}'(\hat{\sigma}) \triangleq \begin{cases} \{[z_{k_\varnothing}, \ldots, z_{h-1}] \mid \forall i \in [k_\varnothing, h{-}1],\ z_i \in \dot{\gamma}(v_i)\}, & \text{se } k_\varnothing > 0; \\ \{s \circ [z_{k_\top}, \ldots, z_{h-1}] \mid s \in \mathbb{S},\ \forall i \in [k_\top, h{-}1],\ z_i \in \dot{\gamma}(v_i)\}, & \text{se } k_\varnothing = 0, \end{cases} \tag{4.9}$$

dove $s \circ [z_{k_\top}, \ldots, z_{h-1}]$ denota la concatenazione dello *stack* s con la lista $[z_{k_\top}, \ldots, z_{h-1}]$. La funzione di concretizzazione $\overline{\gamma} \colon \Sigma^h \to \wp(\mathbb{S})$ è infine:

$$\overline{\gamma}(\hat{\sigma}) \triangleq \begin{cases} \{\bot_{\mathbb{S}}\}, & \text{se } \hat{\sigma} = \bot_{\Sigma^h}; \\ \overline{\gamma}'(\hat{\sigma}), & \text{altrimenti.} \end{cases} \tag{4.10}$$

Quando $k_\varnothing > 0$, il prefisso di $\varnothing$ garantisce che lo *stack* concreto abbia esattamente $h - k_\varnothing$ elementi: in quel caso la concretizzazione produce tutti gli *stack* concreti con quella precisa dimensione, con valori determinati da $\dot{\gamma}$ applicata alle posizioni $k_\varnothing, \ldots, h-1$. Quando invece $k_\varnothing = 0$, non si può porre un limite inferiore alla dimensione concreta dello *stack*: la concretizzazione comprende pertanto tutti i possibili *stack* con almeno $h{-}k_\top$ elementi, prefissati da un qualunque *stack* residuo s.

4.2.3 Insiemi di stack astratti di dimensione l

Un singolo *stack* astratto in Σ^h approssima un singolo percorso di esecuzione. Per ragionare sull'insieme di tutti i percorsi che possono raggiungere un dato *opcode* occorre astrarre insiemi di *stack* concreti, ossia elementi di $\wp(\mathbb{S})$. Il terzo livello della gerarchia è il *dominio insieme di stack astratti*, che raccoglie insiemi di al più l *stack* astratti di altezza h.

Definizione 4.2.5 (Dominio insieme di *stack* astratti)**.** Per ogni $h, l > 0$, il dominio insieme di *stack* astratti è:

$$\Sigma^{h,l} \triangleq \wp_{\leq l}(\Sigma^h),$$

dove $\wp_{\leq l}(\Sigma^h) \subseteq \wp(\Sigma^h)$ è la famiglia di tutti i sottoinsiemi di Σ^h con cardinalità $\leq l$, aumentata da un elemento massimo $\top_{\Sigma^{h,l}} = \Sigma^h$ che rappresenta la perdita totale di informazione. L'ordine parziale su $\Sigma^{h,l}$ è l'inclusione insiemistica $\subseteq$.

La presenza del limite l sulla cardinalità è essenziale per garantire la terminazione dell'analisi: senza di esso il dominio conterrebbe insiemi arbitrariamente grandi e il calcolo di punto fisso potrebbe non convergere. L'operatore di unione ordinaria $\cup$ non è adatto come *least upper bound* in $\Sigma^{h,l}$, perché può produrre insiemi di cardinalità $> l$; si ridefinisce pertanto l'operatore come segue.

Definizione 4.2.6 (Operatore di join in $\Sigma^{h,l}$)**.** Dati $\widehat{S}_1, \widehat{S}_2 \in \Sigma^{h,l}$, il *least upper bound* è:

$$\widehat{S}_1 \sqcup_{\Sigma^{h,l}} \widehat{S}_2 \triangleq \begin{cases} \widehat{S}_1 \cup \widehat{S}_2, & \text{se } |\widehat{S}_1 \cup \widehat{S}_2| \leq l; \\ \top_{\Sigma^{h,l}}, & \text{altrimenti.} \end{cases} \tag{4.11}$$

Il join di due insiemi viene calcolato come unione ordinaria fintanto che il risultato non supera il limite l; superato tale limite, si perde traccia dei singoli *stack* e si collassa al massimo $\top_{\Sigma^{h,l}}$. Questa strategia preserva la precisione per le coppie di insiemi la cui unione è ancora sufficientemente piccola, e garantisce la terminazione per qualsiasi catena ascendente.

Proposizione 4.2.1 (Struttura reticolare di $\Sigma^{h,l}$)**.** *La sestupla*

$$\langle \Sigma^{h,l},\ \subseteq,\ \sqcup_{\Sigma^{h,l}},\ \cap,\ \emptyset,\ \top_{\Sigma^{h,l}} \rangle$$

è un reticolo completo che soddisfa la condizione di catene ascendenti e ha altezza finita $l + 2$.

La dimostrazione di questa proprietà è riportata nella Sezione 4.4, dove si verifica formalmente che ogni sottoinsieme di $\Sigma^{h,l}$ ammette estremo superiore e inferiore, e che ogni catena ascendente è di lunghezza al più $l + 2$, il che garantisce la convergenza del calcolo di punto fisso dell'algoritmo di costruzione del CFG.

La concretizzazione di un elemento di $\Sigma^{h,l}$ si ottiene semplicemente unendo le concretizzazioni di tutti gli *stack* astratti che lo compongono.

Definizione 4.2.7 (Concretizzazione di $\Sigma^{h,l}$)**.** La funzione $\gamma \colon \Sigma^{h,l} \to \wp(\mathbb{S})$ è:

$$\gamma(\widehat{S}) \triangleq \bigcup_{\hat{\sigma} \in \widehat{S}} \overline{\gamma}(\hat{\sigma}). \tag{4.12}$$

Intuitivamente, un elemento $\widehat{S} \in \Sigma^{h,l}$ è un insieme di *stack* astratti, ciascuno dei quali sovra-approssima un insieme di *stack* concreti tramite la funzione $\overline{\gamma}$ definita nella Definizione 4.2.4. La concretizzazione dell'intero insieme si ottiene raccogliendo, con l'operatore di unione insiemistica, tutti gli *stack* concreti rappresentati dai singoli elementi di $\widehat{S}$: ogni $\hat{\sigma} \in \widehat{S}$ contribuisce con il proprio insieme $\overline{\gamma}(\hat{\sigma})$, e $\gamma(\widehat{S})$ è l'unione di tali contributi.

La concretizzazione γ è monotona: se $\widehat{S}_1 \subseteq \widehat{S}_2$, allora $\gamma(\widehat{S}_1) \subseteq \gamma(\widehat{S}_2)$. Questa proprietà è indispensabile per inquadrare il dominio astratto nel framework dell'interpretazione astratta della Sezione 3.2 e per ricavare la correttezza della semantica astratta rispetto a quella concreta.

4.2.4 Semantica astratta delle istruzioni

Definito il dominio $\Sigma^{h,l}$, si può ora specificare come ogni *opcode* trasformi gli *stack* astratti che lo raggiungono. Poiché l'evoluzione del *program counter* è già codificata negli archi del CFG, la semantica astratta deve soltanto modellare la trasformazione dello *stack* operandi, astraendo la componente ℓ dello stato di programma. Si introduce pertanto la funzione $[\![\mathsf{op}]\!]^\sharp \colon \Sigma^{h,l} \to \Sigma^{h,l}$, che dovrà essere una sovra-approssimazione della semantica concreta elevata ai powerset, $[\![\mathsf{op}]\!] \colon \wp(\mathbb{S}) \to \wp(\mathbb{S})$.

Prima di definire $[\![\cdot]\!]^\sharp$ sugli insiemi, si definiscono alcune funzioni ausiliarie che operano su singoli *stack* astratti $\hat{\sigma} \in \Sigma^h$.

Funzioni ausiliarie su Σ^h. La funzione $\mathsf{top} \colon \Sigma^h \to \mathbb{Z}^\sharp$ restituisce l'elemento in cima allo *stack* astratto senza rimuoverlo:

$$\mathsf{top}(\hat{\sigma}) \triangleq \begin{cases} v_{h-1}, & \text{se } \hat{\sigma} = [v_0, \ldots, v_{h-1}]; \\ \varnothing, & \text{se } \hat{\sigma} = \bot_{\Sigma^h}. \end{cases} \tag{4.13}$$

La funzione $\mathsf{push} \colon \Sigma^h \times \mathbb{Z}^\sharp \to \Sigma^h$ inserisce un nuovo elemento $v \in \mathbb{Z}^\sharp$ in cima allo *stack* astratto. Poiché la dimensione è fissa ad h, l'inserimento causa uno scorrimento verso sinistra di tutti gli elementi e la perdita dell'elemento più in basso:

$$\mathsf{push}(\hat{\sigma}, v) \triangleq [v_1, v_2, \ldots, v_{h-1}, v] \quad \text{per } \hat{\sigma} = [v_0, v_1, \ldots, v_{h-1}], \tag{4.14}$$

con $\mathsf{push}(\bot_{\Sigma^h}, v) \triangleq \bot_{\Sigma^h}$. La versione iterata $\mathsf{push}^n \colon \Sigma^h \times \mathbb{Z}^{\sharp n} \to \Sigma^h$, che inserisce una sequenza di n elementi, è definita per ricorsione:

$$\mathsf{push}^n(\hat{\sigma}, v_1, \ldots, v_n) \triangleq \begin{cases} \hat{\sigma}, & \text{se } n = 0; \\ \mathsf{push}^{n-1}(\mathsf{push}(\hat{\sigma}, v_1),\ v_2, \ldots, v_n), & \text{altrimenti.} \end{cases} \tag{4.15}$$

La funzione $\mathsf{pop} \colon \Sigma^h \to \Sigma^h$ rimuove l'elemento in cima allo *stack* astratto. Per preservare la corretta informazione sulla dimensione concreta, la posizione lasciata libera in fondo viene riempita con $\varnothing$ se l'attuale elemento in fondo è già $\varnothing$, indicando che lo *stack* concreto non contiene altri elementi; altrimenti viene riempita con $\top_{\mathbb{Z}^\sharp}$, a segnalare che potrebbero esistere elementi aggiuntivi di valore ignoto. Per $\hat{\sigma} = [v_0, \ldots, v_{h-2}, v_{h-1}]$:

$$\mathsf{pop}(\hat{\sigma}) \triangleq \begin{cases} [\varnothing, v_0, \ldots, v_{h-2}], & \text{se } v_0 = \varnothing; \\ [\top_{\mathbb{Z}^\sharp}, v_0, \ldots, v_{h-2}], & \text{altrimenti,} \end{cases} \tag{4.16}$$

con $\mathsf{pop}(\bot_{\Sigma^h}) \triangleq \bot_{\Sigma^h}$. La versione iterata $\mathsf{pop}^n \colon \Sigma^h \to \Sigma^h$ è analogamente:

$$\mathsf{pop}^n(\hat{\sigma}) \triangleq \begin{cases} \hat{\sigma}, & \text{se } n = 0; \\ \mathsf{pop}^{n-1}(\mathsf{pop}(\hat{\sigma})), & \text{altrimenti.} \end{cases} \tag{4.17}$$

La funzione $\mathsf{height}\colon \Sigma^h \to \mathbb{N}$ conta il numero di elementi non $\varnothing$ presenti nello *stack* astratto, corrispondente alla dimensione minima dello *stack* concreto rappresentato:

$$\mathsf{height}(\hat{\sigma}) \triangleq \begin{cases} h - k_{\varnothing}, & \text{se } \hat{\sigma} \neq \bot_{\Sigma^h}; \\ 0, & \text{altrimenti,} \end{cases} \tag{4.18}$$

dove $k_{\varnothing}$ è l'indice definito nella Definizione 4.2.4. Poiché la semantica astratta non inserisce mai $\varnothing$ direttamente sullo *stack*, tutti gli eventuali $\varnothing$ si trovano sempre in fondo, e height è quindi ben definita.

Semantica astratta su singolo *stack*. Ricordando la notazione $\mathtt{op}^{\rho}_{\delta}$ introdotta nella Sezione 4.1 per un *opcode* che preleva δ valori e ne inserisce ρ, la semantica astratta su singolo *stack* è:

$$[\![\mathtt{op}]\!]^{\sharp}\,\hat{\sigma} \triangleq \begin{cases} \bot_{\Sigma^h}, & \text{se } \hat{\sigma} = \bot_{\Sigma^h} \text{ oppure } \mathsf{height}(\hat{\sigma}) < \delta; \\ \mathsf{push}^{\rho}(\mathsf{pop}^{\delta}(\hat{\sigma}),\, \dot{v}_0, \ldots, \dot{v}_{\rho-1}), & \text{altrimenti,} \end{cases} \tag{4.19}$$

dove $\dot{v}_0, \ldots, \dot{v}_{\rho-1} \in \mathbb{Z}^{\sharp}$ sono i valori astratti prodotti dall'*opcode* in funzione dei suoi operandi. La semantica restituisce $\bot_{\Sigma^h}$ in due situazioni: quando lo *stack* di ingresso è già invalido, o quando il numero di elementi disponibili è inferiore a δ, che corrisponde astrattamente alla condizione di *stack underflow* della semantica concreta. Negli altri casi, si prelevano δ elementi tramite pop^{δ} e si inseriscono ρ nuovi elementi con push^{ρ}. Il calcolo specifico dei valori $\dot{v}_i$ dipende dall'*opcode* e non influisce sul flusso di controllo.

Semantica astratta su insiemi di *stack*. La semantica astratta si estende agli elementi di $\Sigma^{h,l}$ applicando $[\![\mathtt{op}]\!]^{\sharp}$ a ogni singolo *stack* astratto dell'insieme:

$$[\![\mathtt{op}]\!]^{\sharp}\,\widehat{S} \triangleq \begin{cases} \{[\![\mathtt{op}]\!]^{\sharp}\,\hat{\sigma} \mid \hat{\sigma} \in \widehat{S}\}, & \text{se } \left|\{[\![\mathtt{op}]\!]^{\sharp}\,\hat{\sigma} \mid \hat{\sigma} \in \widehat{S}\}\right| \leq l; \\ \top_{\Sigma^{h,l}}, & \text{altrimenti.} \end{cases} \tag{4.20}$$

Se l'applicazione puntuale produce al più l *stack* astratti distinti, questi vengono raccolti in un elemento di $\Sigma^{h,l}$; in caso contrario si ricade al massimo $\top_{\Sigma^{h,l}}$, preservando la correttezza a scapito della precisione.

Osservazione 1. La semantica $[\![\cdot]\!]^{\sharp}$ non include la trattazione specifica di `JUMP` e `JUMPI`. Questi *opcode* si distinguono dagli altri perché la loro esecuzione modifica il *program counter* in modo dipendente dai valori sullo *stack*, e la determinazione dei successori è dunque parte integrante della costruzione del CFG piuttosto che della sola trasformazione dello *stack*. La gestione di `JUMP` e `JUMPI` viene affrontata dall'algoritmo descritto nella Sezione 4.3, che impiega top e $\dot{\gamma}$ per estrarre le possibili destinazioni di salto dallo *stack* astratto e aggiornare il grafo di conseguenza.

4.3 Algoritmi di costruzione del CFG

Il dominio astratto $\Sigma^{h,l}$ introdotto nella Sezione 4.2 fornisce, per ciascun *opcode* del programma, una sovra-approssimazione dell'insieme degli *stack* operandi che possono raggiungerlo durante l'esecuzione. Questa sovra-approssimazione è il punto di partenza della strategia di costruzione del CFG presentata di seguito: le destinazioni degli *opcode* `JUMP` e `JUMPI` vengono determinate ispezionando il valore in cima agli *stack* astratti che raggiungono tali istruzioni, e i corrispondenti archi vengono aggiunti al grafo. Poiché l'aggiunta di archi modifica la struttura del CFG e quindi il risultato dell'analisi, il procedimento viene iterato fino al raggiungimento di un punto fisso, ossia finché non si aggiungono più archi nuovi.

La procedura si articola in due funzioni: BUILDCFG, che coordina la costruzione iterativa, e JUMPSOLVER, che in ciascuna iterazione esegue l'analisi astratta e aggiorna il grafo.

4.3.1 Algoritmo buildCFG

La funzione BUILDCFG, riportata nell'Algoritmo 3, riceve in ingresso un programma EVM `P`, i parametri interi positivi h e l che configurano il dominio astratto $\Sigma^{h,l}$, e un flag booleano *conservative* il cui significato verrà precisato nella Sezione 4.3.2. La funzione restituisce un CFG $G_P = (N, E)$, dove N è l'insieme degli indirizzi di tutti gli *opcode* del programma e E è l'insieme degli archi che ne descrivono il flusso di controllo.

Algoritmo 3 Costruzione del CFG di un programma EVM.

```
1: function buildCFG(P, h, l, conservative)
2:     G_P ← partialCFG(P)
3:     do
4:         changed ← jumpSolver(G_P, h, l, conservative)
5:     while changed
6:     return G_P
```

La costruzione inizia alla riga 1 invocando PARTIALCFG sul programma `P` per ottenere un CFG parziale, in cui nessuna destinazione di salto è ancora risolta. In questo grafo iniziale i nodi coincidono con gli indirizzi di tutti gli *opcode* e gli archi sono limitati a due categorie: il collegamento lineare tra ciascun *opcode* non terminale e il proprio successore naturale $\texttt{next}(\ell)$, e il ramo *false* di ogni istruzione `JUMPI` verso l'indirizzo $\texttt{next}(\ell)$. Quest'ultimo arco è aggiunto staticamente perché il successore naturale di `JUMPI` è determinabile sintatticamente, indipendentemente dal valore dello *stack*: quando la condizio-

ne è zero, il salto non viene eseguito e il *program counter* avanza all'istruzione successiva.

A partire da questo grafo parziale, il ciclo do-while (righe 3–5) invoca ripetutamente JUMPSOLVER (riga 4), che a ogni iterazione esegue l'interpretazione astratta sul CFG corrente e aggiunge gli archi corrispondenti alle destinazioni di salto identificate. La funzione JUMPSOLVER restituisce *true* se e solo se almeno un arco è stato aggiunto; il ciclo prosegue finché *changed* rimane *true* (riga 5), condizione che segnala il raggiungimento del punto fisso. La correttezza della terminazione è garantita dalla struttura a catene ascendenti finite del reticolo $\Sigma^{h,l}$, come stabilito dalla Proposizione 4.2.1: il numero di archi aggiungibili è limitato superiormente da $n \times m$, con n il numero di istruzioni di salto e m il numero di *opcode* `JUMPDEST` nel programma, quindi il ciclo termina in un numero finito di iterazioni. Infine la riga 6 restituisce il CFG costruito.

4.3.2 Algoritmo jumpSolver

La funzione JUMPSOLVER, riportata nell'Algoritmo 4, riceve in ingresso il CFG corrente $G_P = (N, E)$, i parametri h e l e il flag *conservative*. Il suo compito è eseguire l'analisi astratta sul CFG e, in base ai risultati, aggiungere al grafo gli archi corrispondenti alle destinazioni di salto risolte.

Algoritmo 4 Risoluzione iterativa delle destinazioni di salto.

```
 1: function JUMPSOLVER(G_P = (N, E), h, l, conservative)
 2:     A ← RUNANALYSIS(G_P, h, l)
 3:     E' ← E
 4:     for all (Ŝ_in, ℓ, Ŝ_out) ∈ A  con  Π(ℓ) ∈ {JUMP, JUMPI} do
 5:         if Ŝ_in = ⊤_{Σ^{h,l}} then
 6:             if conservative then
 7:                 E ← E ∪ { BUILDEDGE(ℓ, ℓ') | ℓ' ∈ 𝕁 }
 8:         else
 9:             for all σ̂ ∈ Ŝ_in do
10:                 v ← top(σ̂)
11:                 if v ∈ 𝕁 then
12:                     E ← E ∪ { BUILDEDGE(ℓ, v) }
13:                 else if v = ⊤_{ℤ♯}  e  conservative then
14:                     E ← E ∪ { BUILDEDGE(ℓ, ℓ') | ℓ' ∈ 𝕁 }
15:     return E ≠ E'
```

Analisi astratta sul CFG corrente. La prima operazione (riga 2) consiste nell'invocare RUNANALYSIS, che esegue l'interpretazione astratta sul CFG

corrente G_P utilizzando il dominio $\Sigma^{h,l}$ della Sezione 4.2. Il risultato $\mathcal{A}$ (riga 4) associa a ciascun nodo ℓ del grafo una tripla contenente lo *stack* astratto di ingresso $\widehat{S}_{in}$, l'indirizzo ℓ e lo *stack* astratto di uscita $\widehat{S}_{out}$. L'invariante $\widehat{S}_{in}$ raccoglie la sovra-approssimazione di tutti gli *stack* concreti che possono raggiungere ℓ percorrendo un qualunque cammino nel grafo; $\widehat{S}_{out}$ sovra-approssima gli *stack* prodotti dall'esecuzione di $\Pi(\ell)$ a partire da $\widehat{S}_{in}$. I nodi non raggiungibili, per i quali non esiste alcun cammino dal nodo iniziale, sono associati allo stato $\bot_{\Sigma^{h,l}}$ di $\Sigma^{h,l}$. Infine, la riga 15 restituisce il valore booleano che indica se il grafo è stato modificato.

Aggiornamento degli archi. Le righe successive ispezionano i risultati dell'analisi per ciascun nodo ℓ la cui istruzione è `JUMP` o `JUMPI`. La logica di aggiornamento degli archi distingue due situazioni principali.

Il primo caso si verifica quando $\widehat{S}_{in} = \top_{\Sigma^{h,l}}$ (righe 5–7), ossia quando l'analisi ha perso completamente traccia degli *stack* che raggiungono il salto, a causa di un numero di percorsi distinti superiore al limite l del dominio. In questa condizione non è possibile ricavare alcuna informazione utile sulla destinazione del salto. Se il flag *conservative* è attivo (riga 6), si ricorre alla strategia più conservativa aggiungendo un arco dal nodo ℓ verso ogni possibile *opcode* `JUMPDEST` del programma. In caso contrario, nessun arco viene aggiunto, accettando consapevolmente una sotto-approssimazione del grafo.

Il secondo caso (righe 8–14) tratta la situazione in cui $\widehat{S}_{in} \neq \top_{\Sigma^{h,l}}$: si itera su ogni *stack* astratto $\hat{\sigma} \in \widehat{S}_{in}$ (riga 9) e si estrae il valore in cima tramite $\mathsf{top}(\hat{\sigma})$ (riga 10). Se tale valore v appartiene a $\mathbb{J}$ (riga 11), ossia coincide con l'indirizzo di un `JUMPDEST` valido, viene aggiunto l'arco $\ell \to v$ (riga 12). La funzione ausiliaria BUILDEDGE(ℓ, v) costruisce un arco *true* quando $\Pi(\ell) =$ `JUMPI`, oppure un arco ordinario quando $\Pi(\ell) =$ `JUMP`. Se invece $v = \top_{\mathbb{Z}^\sharp}$ (riga 13), il valore è numericamente sconosciuto e potrebbe o meno corrispondere a un `JUMPDEST`: in questo caso, se *conservative* è attivo, si aggiungono archi verso tutti gli *opcode* `JUMPDEST` (riga 14); altrimenti il salto viene sotto-approssimato.

Complessità e terminazione. Il numero massimo di archi aggiungibili durante l'intera esecuzione di BUILDCFG è $n \times m$, dove n è il numero di istruzioni `JUMP` e `JUMPI` e m è il numero di *opcode* `JUMPDEST` nel programma. Tale limite costituisce un maggiorante molto conservativo del numero di chiamate a JUMPSOLVER: nella pratica la maggior parte dei salti si risolve verso un singolo *opcode* `JUMPDEST`, e molti archi vengono aggiunti in un'unica chiamata, rendendo il numero effettivo di iterazioni molto inferiore al caso peggiore.

4.3.3 Esempio guida

Per illustrare il funzionamento concreto degli algoritmi, si considera il frammento di bytecode riportato nel Codice 5, che contiene un *salto orfano* all'indirizzo 10. Un salto è detto orfano quando la sua destinazione non è ricavabile direttamente dall'istruzione di `PUSH` che la precede, bensì viene calcolata a partire da valori intermedi presenti sullo *stack* [3]. L'istruzione `JUMPI` all'indirizzo

Algoritmo 5 Frammento di bytecode EVM con salto orfano all'indirizzo 10.

```
 0:  PUSH1 0x05
 2:  PUSH1 0x05
 4:  EQ
 5:  PUSH1 0x08
 7:  PUSH1 0x04
 9:  ADD
10:  JUMPI       <- salto orfano: destinazione = 0x08 + 0x04 = 12
11:  INVALID
12:  JUMPDEST
13:  PUSH1 0x01
15:  JUMPDEST

```

10 è orfana perché la destinazione non è il valore immediato di una singola `PUSH`, bensì il risultato dell'operazione aritmetica `ADD` applicata alle due costanti `0x08` e `0x04` caricate in precedenza. Per determinare staticamente che la destinazione è 12, l'analisi deve quindi propagare i valori concreti attraverso `ADD` e riconoscere che $8 + 4 = 12$ è l'indirizzo di un `JUMPDEST` valido.

Stack astratto al punto di programma 10. Si fissa $h = 4$ e $l = 1$. L'interpretazione astratta del frammento, eseguita sul CFG parziale iniziale $G_\perp$ in cui il ramo *true* di `JUMPI` non è ancora risolto, calcola gli *stack* astratti a ciascun *opcode*. Al punto di programma $\ell = 10$, ossia subito prima dell'esecuzione di `JUMPI`, lo *stack* astratto di ingresso appartiene a $\Sigma^{4,1}$ ed è composto dall'unico elemento:

$$\hat{\sigma}_{10} = [\varnothing,\ \varnothing,\ 1,\ 12].$$

In questa rappresentazione la cima dello *stack* è l'elemento più a destra: $\mathsf{top}(\hat{\sigma}_{10}) = 12$ è la destinazione del salto, e il valore immediatamente sotto, 1, è la condizione di ramificazione. I due $\varnothing$ in fondo indicano che lo *stack* concreto contiene esattamente due elementi, senza ulteriori elementi al di sotto. La Figura 4.2 illustra graficamente questo *stack* astratto.

CFG parziale iniziale e CFG finale. La Figura 4.3 mostra i due CFG prodotti rispettivamente prima e dopo l'esecuzione di JUMPSOLVER.

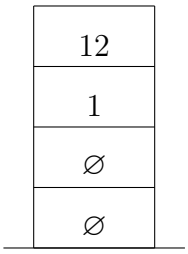


Figura 4.2: *Stack* astratto $\hat{\sigma}_{10} \in \Sigma^{4,1}$ al punto di programma 10, prima di `JUMPI`.

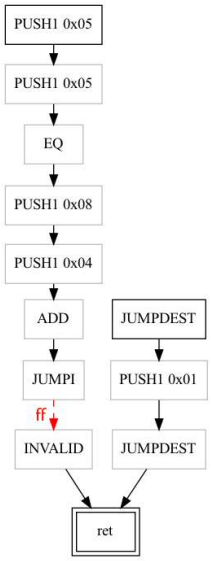
Traccia di esecuzione passo-passo. L'esecuzione di BUILDCFG sul frammento di bytecode si svolge come segue.

1. PARTIALCFG costruisce il CFG parziale $G_\perp$, rappresentato nella Figura 4.3a. Gli archi presenti sono i collegamenti lineari tra *opcode* consecutivi e il ramo *false* di `JUMPI` verso `INVALID` all'indirizzo 11. Il ramo *true* è assente perché la destinazione del salto non è ancora stata determinata.

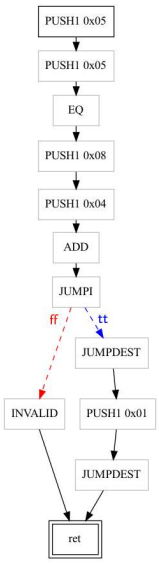
2. Prima iterazione: JUMPSOLVER invoca RUNANALYSIS su $G_\perp$ con $h = 4$ e $l = 1$. L'analisi propaga gli *stack* astratti lungo il grafo e calcola, all'indirizzo 10, lo *stack* di ingresso $\widehat{S}_{in} = \{\hat{\sigma}_{10}\}$, dove $\hat{\sigma}_{10} = [\varnothing, \varnothing, 1, 12]$ come illustrato nella Figura 4.2. Poiché $\widehat{S}_{in} \neq \top_{\Sigma^{h,l}}$, si itera su $\hat{\sigma}_{10}$: si calcola $v = \mathsf{top}(\hat{\sigma}_{10}) = 12$ e si verifica che $12 \in \mathbb{J}$, in quanto all'indirizzo 12 si trova un `JUMPDEST`. Viene quindi aggiunto l'arco *true* $10 \to 12$ tramite BUILDEDGE$(10, 12)$. La funzione restituisce *true*, segnalando che il grafo è stato modificato.

3. Seconda iterazione: JUMPSOLVER viene chiamata sul CFG aggiornato, che ora include l'arco $10 \to 12$. L'analisi ricalcola gli invarianti tenendo conto del nuovo arco: per l'`JUMPI` all'indirizzo 10 lo *stack* di ingresso rimane invariato, poiché il ramo aggiunto non altera il flusso di dati che precede il salto. Nessun arco nuovo viene identificato e la funzione restituisce *false*.

4. BUILDCFG rileva che *changed* = *false* e termina il ciclo. Il CFG finale G_P, rappresentato nella Figura 4.3b, contiene tutti gli archi che modellano correttamente il flusso di controllo del frammento.



(a) CFG parziale $G_\perp$ prima di JUMPSOLVER: il ramo *true* di `JUMPI` non è risolto.



(b) CFG finale G_P con l'arco $10 \to 12$ che risolve il ramo *true* di `JUMPI`.

Figura 4.3: Evoluzione del CFG per il Codice 5: stato iniziale (sinistra) e finale dopo la risoluzione del salto orfano (destra).

4.3.4 Classificazione dei salti e *soundness*

Una volta terminata l'esecuzione di BUILDCFG, è possibile classificare ciascuna istruzione di salto in base agli *stack* astratti che la raggiungono nel CFG prodotto. Questa classificazione fornisce, per ciascun salto, un'indicazione precisa sul tipo di informazione disponibile e consente di stabilire in quali condizioni il CFG generato sia *sound* nel senso della Definizione 3.3.2 [3].

Sia $\widehat{S} = \{\hat{\sigma}_0, \ldots, \hat{\sigma}_{n-1}\}$, con $n \in \mathbb{N}$, l'insieme degli *stack* astratti che raggiunge un nodo di salto ℓ nel CFG finale. Si dice che uno *stack* astratto $\hat{\sigma}_i \in \widehat{S}$ è *erroneo* per l'istruzione $\Pi(\ell)$ se $\mathsf{height}(\hat{\sigma}_i) = 0$, oppure se $\Pi(\ell) =$ `JUMPI` e $\mathsf{height}(\hat{\sigma}_i) = 1$: in questi casi lo *stack* descrive un'esecuzione già erronea, o un'esecuzione che produrrebbe immediatamente uno *stack underflow* tentando di eseguire il salto. Con questa nozione, i salti vengono classificati nelle quattro categorie seguenti.

Unreachable. Il salto è non raggiungibile quando $\widehat{S} = \emptyset$: nessuno *stack* astratto raggiunge il nodo di salto e, poiché l'analisi è una sovra-approssimazione della semantica concreta, nessuna esecuzione concreta può attraversare quel nodo. Questa categoria è utile per identificare codice

morto, ma la sua correttezza dipende dall'assenza di salti di categoria *unknown*, come discusso nel seguito.

Erroneous. Il salto è erroneo quando tutti gli *stack* astratti $\hat{\sigma}_i \in \widehat{S}$ sono erronei. In questa situazione, ogni esecuzione concreta che raggiunge ℓ produce necessariamente un'eccezione di *stack underflow* prima o durante l'esecuzione del salto. Il CFG non include archi in uscita da tale nodo perché nessuna esecuzione lo supera correttamente.

Resolved. Il salto è risolto quando tutti gli *stack* astratti non erronei $\hat{\sigma}_i \in \widehat{S}$ soddisfano $\mathsf{top}(\hat{\sigma}_i) \neq \top_{\mathbb{Z}^\sharp}$: il valore in cima allo *stack* è quindi un intero concreto o il simbolo $\top_{\overline{\mathbb{Z}}}$. Nel primo caso la destinazione del salto è nota con precisione; nel secondo si garantisce che il valore non corrisponde a nessun `JUMPDEST`. Per i salti di questa categoria, la *soundness* del CFG è garantita: tutti gli archi corrispondenti a destinazioni concrete raggiungibili sono stati aggiunti da JUMPSOLVER.

Unknown. Il salto è sconosciuto in due situazioni distinte: quando $\widehat{S} = \top_{\Sigma^{h,l}}$, il che indica che l'analisi ha perso completamente l'informazione sugli *stack* in ingresso, oppure quando almeno uno *stack* astratto non erroneo $\hat{\sigma}_i \in \widehat{S}$ presenta $\mathsf{top}(\hat{\sigma}_i) = \top_{\mathbb{Z}^\sharp}$, ossia il valore in cima è numericamente sconosciuto e potrebbe corrispondere a un indirizzo di `JUMPDEST`. In entrambi i casi la *soundness* del CFG non è garantita in modo incondizionato.

Condizioni di *soundness*. Per i salti della categoria *resolved*, la *soundness* è garantita indipendentemente dal valore del flag *conservative*: l'analisi ha determinato con sufficiente precisione le possibili destinazioni concrete e gli archi corrispondenti sono stati inseriti correttamente nel grafo.

Per i salti della categoria *unknown*, la *soundness* dipende invece dal valore di *conservative*. Se *conservative* = *true*, JUMPSOLVER aggiunge archi verso tutti gli *opcode* `JUMPDEST` del programma, ottenendo così una sovra-approssimazione conservativa che preserva la correttezza. Se *conservative* = *false*, i salti di categoria *unknown* vengono sotto-approssimati: nel caso $\widehat{S} = \top_{\Sigma^{h,l}}$ nessun arco viene aggiunto, mentre nel caso $\mathsf{top}(\hat{\sigma}_i) = \top_{\mathbb{Z}^\sharp}$ vengono aggiunti solo gli archi verso le destinazioni note con certezza. Questa scelta è intenzionale: la presenza di salti *unknown* segnala che i parametri h e l sono probabilmente troppo restrittivi, invitando a esplorare combinazioni più generose finché quei salti non diventino *resolved*.

La condizione necessaria e sufficiente per la *soundness* del CFG prodotto da BUILDCFG si esprime quindi in modo compatto: il grafo è *sound* se e solo se *conservative* = *true*, oppure non esistono salti di categoria *unknown*. La dimostrazione formale di questo risultato è presentata nella Sezione 4.4.

4.4 Proprietà formali

Si dimostrano qui le proprietà formali del dominio $\Sigma^{h,l}$ e degli algoritmi BUILDCFG e JUMPSOLVER. In particolare: $\Sigma^{h,l}$ è un reticolo completo con la condizione di catene ascendenti (Teorema 4.4.1); la funzione di concretizzazione γ è monotona (Teorema 4.4.2); la semantica astratta $[\![\cdot]\!]^\sharp$ è una sovra-approssimazione *sound* della semantica concreta $[\![\cdot]\!]$ (Teorema 4.4.3); JUMPSOLVER è estensivo e monotono sul reticolo dei CFG (Lemmi 4.4.4–4.4.6); BUILDCFG termina in un numero finito di iterazioni (Teorema 4.4.7); il CFG prodotto con *conservative* = *true* è *sound* nel senso della Definizione 3.3.2 del Capitolo 3 (Teorema 4.4.8). Quest'ultimo risultato chiude la condizione di correttezza anticipata nella Sezione 4.3.4 e costituisce la base su cui l'analisi taint del Capitolo 5 si innesta. Le dimostrazioni complete sono nell'Appendice A.

4.4.1 Struttura reticolare del dominio astratto

Teorema 4.4.1 (Reticolo completo ACC di $\Sigma^{h,l}$)**.** *La sestupla*

$$\langle \Sigma^{h,l},\ \subseteq,\ \sqcup_{\Sigma^{h,l}},\ \cap,\ \emptyset,\ \top_{\Sigma^{h,l}} \rangle$$

è un reticolo completo che soddisfa la condizione di catene ascendenti, con altezza finita pari a $l + 2$*.*

4.4.2 Monotonia della funzione di concretizzazione

Teorema 4.4.2 (Monotonia di γ)**.** *La funzione di concretizzazione* $\gamma \colon \Sigma^{h,l} \to \wp(\mathbb{S})$ *è monotona rispetto all'inclusione insiemistica, ossia per ogni* $\widehat{S}_1, \widehat{S}_2 \in \Sigma^{h,l}$*:*

$$\widehat{S}_1 \subseteq \widehat{S}_2 \implies \gamma(\widehat{S}_1) \subseteq \gamma(\widehat{S}_2).$$

4.4.3 Correttezza della semantica astratta

Teorema 4.4.3 (Correttezza di $[\![\cdot]\!]^\sharp$)**.** *Per ogni opcode* `op` *e per ogni* $\widehat{S} \in \Sigma^{h,l}$ *vale:*

$$[\![\texttt{op}]\!]\,\gamma(\widehat{S}) \ \subseteq\ \gamma([\![\texttt{op}]\!]^\sharp\,\widehat{S}),$$

dove $[\![\texttt{op}]\!]$ *applicato a un insieme di stati è inteso come estensione puntuale:* $[\![\texttt{op}]\!]\,A = \bigcup_{s \in A} \{[\![\texttt{op}]\!]\,s\}$*.*

4.4.4 Struttura reticolare dei CFG e proprietà di jumpSolver

La terminazione di buildCFG si fonda su una struttura reticolare sull'insieme dei CFG per un programma fissato. Si mostra che jumpSolver è estensivo e monotono su tale reticolo.

Definizione 4.4.1 (CFG per il programma P)**.** Dato un programma EVM P, siano N l'insieme degli indirizzi di tutti i suoi *opcode* e $\mathbb{J}$ l'insieme degli indirizzi di JUMPDEST. Si definiscono i CFG estremi:

$$G_\bot = (N,\, E_\bot), \qquad E_\bot = \{\ell \to \mathsf{next}(\ell) \mid \ell \in N\};$$
$$G_\top = (N,\, E_\top), \qquad E_\top = E_\bot \cup \{\ell \to \ell' \mid \Pi(\ell) \in \{\texttt{JUMP}, \texttt{JUMPI}\},\ \ell' \in \mathbb{J}\}.$$

Un CFG per P è qualunque $G = (N, E)$ con $E_\bot \subseteq E \subseteq E_\top$.

Il grafo $G_\bot$ coincide con il risultato di partialCFG: contiene solo i collegamenti lineari e i rami *false* di JUMPI. Il grafo $G_\top$ contiene anche tutti i possibili archi di salto verso ogni JUMPDEST ed è il CFG più conservativo ammissibile. L'insieme $\mathbb{G}_\mathtt{P}$ di tutti i CFG per P ha una naturale struttura di reticolo.

Lemma 4.4.4 (Reticolo dei CFG)**.** *L'insieme $\mathbb{G}_P$ con l'ordine $(N, E_1) \sqsubseteq_E (N, E_2)$ se e solo se $E_1 \subseteq E_2$ forma un reticolo finito, completo e con la condizione di catene ascendenti, con minimo $G_\bot$ e massimo $G_\top$.*

Lemma 4.4.5 (jumpSolver è estensivo)**.** *Per ogni $G \in \mathbb{G}_P$, sia G' il CFG prodotto da* jumpSolver$(G, h, l, \mathit{conservative})$*. Allora $G \sqsubseteq_E G'$.*

Lemma 4.4.6 (jumpSolver è monotono)**.** *Dati $G_1, G_2 \in \mathbb{G}_P$ con $G_1 \sqsubseteq_E G_2$, siano G_1' e G_2' i CFG prodotti da* jumpSolver *applicato rispettivamente a G_1 e G_2 con gli stessi parametri h, l, conservative. Allora $G_1' \sqsubseteq_E G_2'$.*

4.4.5 Terminazione e correttezza di buildCFG

Teorema 4.4.7 (Terminazione di buildCFG)**.** *Per ogni programma EVM P e per ogni scelta dei parametri h, l e conservative, la procedura*

$$\text{buildCFG}(P, h, l, \mathit{conservative})$$

termina in un numero finito di iterazioni, calcolando il minimo punto fisso di jumpSolver *sul reticolo $\mathbb{G}_P$.*

Teorema 4.4.8 (Correttezza di buildCFG)**.** *Sia P un programma EVM con primo indirizzo ℓ_0. Allora* buildCFG(P, h, l, true) *restituisce un CFG $G = (N, E)$ sound per P nel senso della Definizione 3.3.2, ossia soddisfa l'Equazione* (3.4)*:*

$$\{\ell \to \ell' \mid \exists s, s' \in \mathbb{S}\ .\ \Xi^\ell(P, \langle [], \ell_0\rangle) = \langle s, \ell\rangle \wedge [\![\Pi(\ell)]\!]\langle s, \ell\rangle = \langle s', \ell'\rangle\} \ \subseteq\ E.$$

Nel caso *conservative* = *false*, la correttezza del CFG è garantita esclusivamente in assenza di salti di categoria *unknown*: se tale condizione è soddisfatta, le categorie *unreachable*, *erroneous* e *resolved* coprono tutti i salti del programma e la dimostrazione del Teorema 4.4.8 si applica invariata. Il CFG prodotto da BUILDCFG con *conservative* = *true* è quindi un CFG *sound* nel senso della Definizione 3.3.2, costituendo la base rigorosa su cui l'analisi taint del Capitolo 5 si innesta.

Capitolo 5

Rilevamento di Access Control Incompleteness nei Bridge Cross-Chain

L'*Access Control Incompleteness* è la classe di vulnerabilità con le conseguenze economiche più gravi tra gli *smart contract* distribuiti su blockchain [33]. Nei *cross-chain bridge* la gravità è maggiore: le architetture di interoperabilità coordinano stato e asset su catene eterogenee attraverso percorsi di esecuzione asincroni, e un singolo punto di ingresso non adeguatamente protetto può causare perdite irreversibili su più catene contemporaneamente. Tre casi reali illustrano le modalità concrete di sfruttamento della vulnerabilità. Il dominio *taint* relazionale introdotto nel Capitolo 3 è poi specializzato all'architettura a *stack* dell'EVM: si definiscono le *source*, i *sink* e i *sanitizer* specifici del modello di minaccia dei *bridge*, si formalizza la proprietà di sicurezza *all-paths*, e si descrive il *checker* implementato in EVMLiSA, comprensivo dell'interfaccia di annotazione dei *modifier* Solidity. I risultati sperimentali sono nel Capitolo 6.

5.1 Esempi motivanti

L'*Access Control Incompleteness* si manifesta in configurazioni distinte: l'assenza di qualsiasi controllo sul chiamante, la propagazione di uno stato permissivo attraverso un percorso di inizializzazione non protetto, l'insufficienza di un controllo aggregato che non ragiona sulla provenienza dei singoli input. I frammenti di codice sono ricostruzioni minimali della logica vulnerabile, progettate per isolare le dipendenze di dati e i flussi di controllo rilevanti ai fini della verifica formale; l'ambiente di riferimento è quello dei contratti Solidity dispiegati su Ethereum e sulle blockchain compatibili con l'EVM.

La proprietà che il *checker* verifica è la seguente: ogni *entry point* raggiungibile dall'esterno deve validare tutti gli input controllati da un attaccante che influenzano operazioni privilegiate, prima che tali input raggiungano un *sink*. I tre casi mostrano altrettante modalità in cui questa proprietà viene violata.

5.1.1 Parity Multisig (2017)

Il caso Parity Multisig del 2017, introdotto nella Sezione 2.5.2, è il riferimento classico per le vulnerabilità di *Access Control Incompleteness*: una funzione di inizializzazione pubblica, priva di qualsiasi verifica sull'identità del chiamante, consentiva a qualunque indirizzo di assumere la proprietà della libreria e di invocare successivamente `selfdestruct` [34, 20]. Lo Snippet di Codice 6 riporta il frammento di libreria in cui la vulnerabilità è localizzata; nel seguito si analizza la corrispondente compilazione in bytecode EVM per mostrare che nessun percorso di esecuzione contiene un *sanitizer* per il flusso che porta dai parametri di inizializzazione alla scrittura dello storage, e che l'analisi *taint* relazionale segnala quindi la violazione della proprietà *all-paths*.

Algoritmo 6 Frammento della libreria Parity Multisig con la funzione di inizializzazione non protetta.

```
contract WalletLibrary {
 address[] public owners;
 uint public required;

 function initWallet(address[] _owners, uint _required) public {
  // VULN: callable by anyone when library is uninitialized
  owners = _owners;
  required = _required;
 }

 function kill(address _to) public onlyOwner {
  selfdestruct(_to);
 }

 modifier onlyOwner() {
  require(isOwner(msg.sender));
  _;
 }

 function isOwner(address _addr) public view returns (bool) {
  for (uint i = 0; i < owners.length; i++) {
   if (owners[i] == _addr) return true;
  }
  return false;
 }
}
```

A livello di bytecode EVM, la funzione `initWallet` alla riga 5 corrisponde a un blocco di istruzioni in cui i parametri `_owners` e `_required` sono caricati dal *calldata*, l'area di sola lettura che contiene gli argomenti della transazione corrente, mediante l'opcode `CALLDATALOAD` e scritti sullo storage persistente del contratto mediante l'opcode `SSTORE` alle righe 7–8. Dal punto di vista dell'analisi *taint*, non esiste alcuna istruzione `JUMPI` la cui condizione dipenda dall'opcode `CALLER` prima che i valori caricati da `CALLDATALOAD` raggiungano l'`SSTORE`: non vi è quindi alcuna diramazione di controllo condizionata sull'identità del chiamante lungo il percorso che conduce dalla *source* al *sink*. La proprietà *all-paths* è violata perché nessun percorso di esecuzione contiene un *sanitizer* per quel flusso. La funzione `kill` alla riga 11, invece, è protetta dal *modifier* `onlyOwner` alle righe 15–18, che verifica l'identità del chiamante prima di consentire l'invocazione di `SELFDESTRUCT` alla riga 12. Poiché però l'inizializzazione non è protetta, l'attaccante può prima richiamare `initWallet` per inserire se stesso come proprietario e poi invocare `kill`.

5.1.2 Nomad Bridge (2022)

L'exploit del bridge Nomad del luglio 2022 [16, 17], descritto nella Sezione 2.5.2, ha causato perdite per circa 190 milioni di dollari sfruttando una funzione di inizializzazione del contratto `Replica` che non vincolava il chiamante a nessuna identità fidata: a seguito di un aggiornamento che aveva inserito `bytes32(0)` come radice valida, qualunque partecipante poteva sottomettere messaggi arbitrari. Il frammento riportato nello Snippet di Codice 7 isola i flussi non sanitizzati alla base della vulnerabilità e mostra come l'analisi *taint* relazionale li identifichi.

Dal punto di vista dell'analisi *taint*, i parametri `_updater` e `_root` della funzione `initialize` alla riga 6 sono caricati tramite `CALLDATALOAD` e propagati fino all'opcode `SSTORE` che scrive i campi `updater` e `committedRoot` alle righe 10–11. Il controllo `require(!initialized)` alla riga 8 si traduce a livello bytecode in una `JUMPI` la cui condizione dipende dal valore letto dallo storage tramite `SLOAD`, non dall'identità del chiamante: è dunque un controllo sullo stato del contratto, non un controllo di autorizzazione sulla provenienza dei dati. L'analisi relazionale lo riconosce come non pertinente perché la condizione della `JUMPI` non è influenzata da `CALLER`, e pertanto non costituisce un *sanitizer* per i flussi provenienti da tale *source*. Il flusso da `CALLDATALOAD` a `SSTORE` rimane quindi non sanitizzato su tutti i percorsi di esecuzione.

Algoritmo 7 Frammento del contratto Replica di Nomad Bridge.

```
1  contract Replica {
2   bytes32 public committedRoot;
3   address public updater;
4   bool public initialized;
5
6   function initialize(address _updater, bytes32 _root) public {
7    // VULN: missing access control on initialize
8    require(!initialized);
9    initialized = true;
10   updater = _updater;
11   committedRoot = _root;
12  }
13
14  function acceptMessage(bytes calldata message,
15              bytes calldata proof) external {
16   // VULN: accepts messages without proper updater authorization
17   require(isProven(message, proof, committedRoot));
18   process(message);
19  }
20
21  function isProven(bytes calldata, bytes calldata,
22           bytes32) internal pure returns (bool) {
23   return true; // simplified placeholder for proof verification
24  }
25
26  function process(bytes calldata) internal { }
27 }
```

5.1.3 Ronin Bridge (2022)

Il bridge Ronin [40, 6], introdotto anch'esso nella Sezione 2.5.2, evidenzia un pattern distinto: un controllo aggregato sulle firme dei validatori che non ragiona sulla provenienza dei singoli input. Il contratto `Bridge`, riportato nello Snippet di Codice 8, verifica che un numero minimo di validatori abbia firmato il messaggio, ma il payload è estratto direttamente dai dati di chiamata e propagato alla funzione `executeWithdrawal` senza che la provenienza di ciascun parametro sia verificata individualmente. La funzione `verifySignatures` controlla che le firme raggiungano una soglia numerica, ma non certifica che il payload corrisponda a un'operazione legittima rispetto allo stato on-chain: questo è il difetto che l'analisi relazionale rileva.

Questo caso mostra perché il ragionamento relazionale per provenienza è necessario. Un'analisi non relazionale potrebbe interpretare il `require` sulla soglia alla riga 8 come un *sanitizer* generico per tutti i dati del messaggio, mascherando la vulnerabilità. L'analisi relazionale, al contrario, traccia la provenienza di ciascun valore *tainted*: il payload è un parametro della funzione `withdraw` alle righe 5–6 e raggiunge la chiamata esterna all'interno di `executeWithdrawal` alla riga 9. L'istruzione `JUMPI` associata al `require` sulla

Algoritmo 8 Frammento del contratto Bridge di Ronin.

```
1  contract Bridge {
2   uint public constant THRESHOLD = 5;
3   mapping(address => bool) public validators;
4
5   function withdraw(bytes calldata payload,
6            bytes[] calldata sigs) external {
7    bytes32 h = keccak256(payload);
8    require(verifySignatures(h, sigs) >= THRESHOLD);
9    executeWithdrawal(payload);
10  }
11
12  function verifySignatures(bytes32, bytes[] calldata)
13                  internal view returns (uint) {
14   // simplified signature count logic
15   return 0;
16  }
17
18  function executeWithdrawal(bytes calldata) internal { }
19 }
```

soglia alla riga 8 ha come condizione il risultato di `verifySignatures`, il quale dipende dall'hash `h` alla riga 7 e dalle firme `sigs`, non direttamente dal payload: il *program point* corrispondente al parametro del payload non compare nell'insieme di provenienza della condizione, pertanto la `JUMPI` non è classificata come *sanitizer* per quel flusso specifico. Il flusso dal payload ai *sink* rimane non sanitizzato.

5.2 Istanziazione del dominio taint all'architettura EVM

Il dominio $\mathcal{T}^{\mathsf{pp}}$, introdotto nella Definizione 3.4.2, è il substrato formale del *checker*. Tale dominio viene specializzato all'architettura a *stack* dell'EVM; le *source* e i *sink* sono istanziati per il modello di minaccia dei *cross-chain bridge*, e si precisa la nozione di *sanitizer* con la proprietà di sicurezza verificata dal *checker*.

L'EVM è una macchina virtuale basata su *stack* senza registri nominati: ogni istruzione agisce sui valori in cima allo *stack* operando in modo puramente posizionale. L'analisi *taint* non può quindi ragionare su variabili con un nome, ma deve tracciare lo stato *taint* di ciascuna posizione dello *stack* in ogni punto del programma. A questo fine, si introduce il dominio degli *stack* astratti *taint*, che astrae le configurazioni concrete di *stack* tracciando lo stato *taint* dei valori nelle prime h posizioni dalla cima. Sorgenti, *sink* e *sanitizer* sono poi istanziati agli opcode EVM pertinenti al modello di minaccia considerato.

5.2.1 Stack astratto taint per l'EVM

Poiché l'EVM esegue le istruzioni operando su uno *stack* di valori a 256 bit, l'analisi *taint* deve mantenere, per ciascun punto del programma, un'approssimazione dello stato *taint* di ogni posizione dello *stack*. In EVMLiSA, lo *stack* astratto introdotto per la costruzione del CFG è specializzato sostituendo i valori numerici con elementi del dominio $\mathcal{T}^{\mathsf{pp}}$. La finestra tracciata è limitata alle h posizioni più alte dello *stack*, dove h è un parametro configurabile: accedere a posizioni al di sotto della finestra produce conservativamente $\top_{\mathcal{T}}$, indicando che la provenienza di quei valori è sconosciuta.

L'insieme degli *stack* astratti di altezza h con elementi del dominio $\mathcal{T}^{\mathsf{pp}}$ è definito come

$$\Sigma^{h,\mathsf{pp}} = \{[t_0, t_1, \ldots, t_{h-1}] \mid \forall i \in [0, h-1] \mid t_i \in \mathcal{T}^{\mathsf{pp}}\}, \tag{5.1}$$

ossia l'insieme degli *stack* formati esattamente da h elementi di $\mathcal{T}^{\mathsf{pp}}$, indicizzati da sinistra, dove l'elemento più a destra t_{h-1} rappresenta la cima dello *stack*. Gli *stack* di dimensione inferiore ad h sono modellati riempiendo le posizioni mancanti con $\bot_{\mathcal{T}}$ in fondo allo *stack*, ovvero nella posizione più a sinistra.

Le operazioni di base sugli *stack* astratti sono la proiezione dell'opcode `PUSH` e quella di `POP`, e si definiscono come segue. La funzione push inserisce un nuovo elemento $t \in \mathcal{T}^{\mathsf{pp}}$ in cima allo *stack*: dato $\widehat{s} = [t_0, t_1, \ldots, t_{h-1}] \in \Sigma^{h,\mathsf{pp}}$,

$$\mathsf{push}(\widehat{s}, t) \triangleq [t_1, t_2, \ldots, t_{h-1}, t], \tag{5.2}$$

ovvero tutti gli elementi vengono traslati verso sinistra di una posizione, l'elemento in fondo t_0 viene scartato, e il nuovo elemento t viene collocato in cima. Il significato intuitivo è che l'analisi mantiene traccia delle sole h posizioni più alte e, quando lo *stack* cresce oltre tale limite, le informazioni sugli elementi più in profondità vengono perse in modo conservativo.

La funzione pop rimuove l'elemento in cima allo *stack*. Dato $\widehat{s} \in \Sigma^{h,\mathsf{pp}}$ con $\widehat{s} = [t_0, t_1, \ldots, t_{h-1}]$, si definisce

$$\mathsf{pop}(\widehat{s}) \triangleq \begin{cases} [\bot_{\mathcal{T}},\, t_0, t_1, \ldots, t_{h-2}] & \text{se } t_0 = \bot_{\mathcal{T}}; \\ [\top_{\mathcal{T}},\, t_0, t_1, \ldots, t_{h-2}] & \text{altrimenti.} \end{cases} \tag{5.3}$$

L'elemento rimosso è t_{h-1}; tutti gli altri vengono traslati verso destra di una posizione. La posizione in fondo è riempita con $\bot_{\mathcal{T}}$ nel caso in cui $t_0 = \bot_{\mathcal{T}}$, indicando che lo *stack* concreto aveva altezza inferiore ad h e la posizione appena scoperta è vuota. In tutti gli altri casi è riempita con $\top_{\mathcal{T}}$, a indicare che la posizione era occupata da un valore non tracciato, la cui provenienza è dunque sconosciuta. Questo comportamento conservativo garantisce che l'analisi non possa falsamente concludere che un valore sia *clean* solo perché la sua posizione sfugge alla finestra di tracciamento.

Le operazioni `DUP`i e `SWAP`i sono modellate coerentemente: `DUP`i duplica l'elemento alla profondità i portandolo in cima; se $i > h$, la posizione è al di fuori della finestra e l'elemento risultante è conservativamente $\top_{\mathcal{T}}$. `SWAP`i scambia la cima con l'elemento alla profondità i; analogamente, se $i > h$, si conserva $\top_{\mathcal{T}}$. Per le operazioni aritmetiche e logiche, la funzione di trasferimento astratta consuma gli operandi dallo *stack* e produce un risultato il cui stato *taint* è calcolato dalla funzione taint definita nell'Equazione (3.10): il risultato è *tainted* se almeno un operando lo è, e il suo insieme di *program point* è l'unione degli insiemi di provenienza degli operandi *tainted*.

L'analisi sugli *stack* astratti *taint* opera in parallelo con il dominio del Capitolo 4: il primo dominio risolve le destinazioni di salto per la costruzione del CFG e traccia i valori concreti necessari a quel fine, mentre il secondo dominio traccia la provenienza dei valori per l'identificazione delle vulnerabilità. I due stadi operano sul medesimo CFG, prodotto nella prima fase di analisi.

5.2.2 Source e sink nell'EVM

Le *source* e i *sink*, introdotti nella Sezione 3.4.1 come componenti generici del framework di analisi *taint*, sono qui istanziati agli opcode EVM pertinenti al modello di minaccia dell'*Access Control Incompleteness* nei *cross-chain bridge*. La scelta è guidata da due criteri: le *source* includono tutti gli opcode che caricano dati controllabili da un attaccante esterno, mentre i *sink* includono le operazioni che modificano lo stato persistente del contratto o trasferiscono valore e controllo a indirizzi esterni, perché sono queste le operazioni attraverso cui una mancanza di controllo sull'accesso produce un danno concreto.

Le *source* comprendono tutti gli opcode che materializzano sullo *stack* dati provenienti dall'ambiente di chiamata. `CALLDATALOAD` e `CALLDATACOPY` caricano i parametri della chiamata; `CALLER` e `ORIGIN` restituiscono l'identità del chiamante diretto e dell'indirizzo che ha originato la transazione; `CALLVALUE` espone il quantitativo di Ether allegato alla chiamata; `CALLDATASIZE` la dimensione del payload. Tutti questi valori sono sotto il controllo diretto di un attaccante che interagisce con il contratto e non devono essere considerati fidati in assenza di un controllo esplicito.

I *sink* comprendono gli opcode attraverso cui un valore *tainted* può causare un danno materiale. `CALL` e `CALLCODE` trasferiscono Ether e invocano funzioni esterne su indirizzi arbitrari; `DELEGATECALL` esegue codice esterno nel contesto di storage del contratto chiamante, con effetti che ne alterano lo stato in modo permanente; `STATICCALL` legge stato da contratti remoti e può essere sfruttato per condizionare il flusso di controllo su dati non fidati. Quando un argomento di destinazione o un dato di payload raggiunge questi opcode senza essere stato verificato, un attaccante può dirigere il trasferimento verso un indirizzo

Tabella 5.1: *Source* e *sink* dell'analisi *taint* per il rilevamento di *Access Control Incompleteness*.

Categoria	**Opcode**	**Significato**
Source	`CALLDATALOAD`	Carica una parola dai dati della chiamata corrente.
	`CALLDATACOPY`	Copia una porzione dei dati della chiamata in memoria.
	`CALLER`	Restituisce l'indirizzo del chiamante diretto.
	`ORIGIN`	Restituisce l'indirizzo che ha originato la transazione.
	`CALLVALUE`	Restituisce la quantità di Ether trasferita con la chiamata.
	`CALLDATASIZE`	Restituisce la dimensione dei dati della chiamata.
Sink	`CALL`	Chiamata esterna con possibile trasferimento di Ether.
	`CALLCODE`	Chiamata che esegue codice esterno preservando lo storage.
	`DELEGATECALL`	Chiamata che esegue codice esterno nel contesto del chiamante.
	`STATICCALL`	Chiamata in sola lettura senza modifiche di stato.
	`SSTORE`	Scrittura sullo storage persistente del contratto.

arbitrario o invocare funzioni arbitrarie su contratti esterni. `SSTORE` scrive direttamente sullo storage persistente del contratto: se un parametro *tainted* lo raggiunge, l'attaccante può sovrascrivere strutture dati critiche, come l'insieme dei proprietari nel caso Parity discusso nella Sezione 5.1.1.

La scelta di escludere gli opcode di sola lettura dalle *source* e gli opcode di controllo di flusso `JUMP` e `JUMPI` dall'insieme dei *sink* è intenzionale: l'obiettivo del *checker* non è rilevare ogni uso di dati non fidati, ma identificare i flussi che, in assenza di una guardia di autorizzazione, producono effetti privilegiati sul contratto o su contratti correlati. I *sanitizer* sono trattati nella Sezione 5.2.3.

5.2.3 Identificazione dei sanitizer

Nell'EVM, ogni diramazione di controllo è implementata dall'opcode `JUMPI`, che preleva dalla cima dello *stack* una destinazione di salto e una condizione booleana. Se la condizione è diversa da zero, il flusso di esecuzione prosegue alla destinazione indicata; altrimenti, prosegue all'istruzione successiva. Un controllo di autorizzazione tipico, come il confronto `msg.sender == owner`, viene compilato in una sequenza che carica `CALLER`, confronta il valore con un dato di storage tramite `EQ` o una funzione equivalente, e passa il risultato come condizione a `JUMPI`, che salta alla sequenza di ripristino se il confronto fallisce.

Un'istruzione `JUMPI` è un *sanitizer* per un determinato flusso se e soltanto se la sua condizione è un valore *tainted* in $\mathcal{T}^{\mathsf{pp}}$: solo in questo caso la diramazione condiziona effettivamente il proseguimento dell'esecuzione sulla base di un input proveniente da una *source* controllata dall'attaccante.

Definizione 5.2.1 (Sanitizer per program point)**.** Un'istruzione `JUMPI` al *program point* pp_j, la cui condizione è $(\mathsf{T}, P) \in \mathcal{T}^{\mathsf{pp}}$, è un *sanitizer* per i flussi la cui *source* appartiene all'insieme P.

Il ragionamento relazionale rende questa identificazione precisa: una `JUMPI` che controlla una condizione derivata da `CALLER` è un *sanitizer* esclusivamente per i flussi provenienti dall'opcode `CALLER`, identificato dal suo *program point* specifico nell'insieme P. Non sanitizza i flussi provenienti da `CALLDATALOAD`, perché il *program point* di questi ultimi non compare nell'insieme P della condizione della `JUMPI`.

Tornando agli esempi motivanti, nel caso Parity la funzione `initWallet` non contiene alcuna `JUMPI` la cui condizione sia influenzata da `CALLER`: l'unico controllo presente è `require(!initialized)`, che dipende da uno stato di storage e non dall'identità del chiamante. Pertanto, nessun *sanitizer* esiste per il flusso che porta da `CALLDATALOAD` a `SSTORE`, e la vulnerabilità è correttamente rilevata. Una situazione analoga si verifica nel caso Nomad: la funzione `initialize` dello Snippet di Codice 7 contiene il solo `require(!initialized)` (riga 8), la cui condizione, come discusso, dipende dal valore letto da `SLOAD` e non da `CALLER`; di conseguenza la `JUMPI` corrispondente non sanitizza alcun flusso proveniente da `CALLDATALOAD`, e la scrittura in storage di `updater` e `committedRoot` è correttamente classificata come violazione della proprietà *all-paths*. Nel caso Ronin, il `require` sulla soglia delle firme condiziona il salto sulla base di un valore derivato dall'hash del payload e dalle firme, i cui *program point* di provenienza non coincidono con il *program point* del `CALLDATALOAD` del payload stesso: la `JUMPI` non è pertanto un *sanitizer* per quel flusso.

5.2.4 Proprietà di sicurezza all-paths

La proprietà di sicurezza verificata dal *checker* è una condizione *all-paths*: un flusso da una *source* al *program point src* fino a un *sink* è sicuro se e soltanto se ogni percorso di esecuzione che collega *src* al *sink* attraversa almeno un *sanitizer* identificato per quel flusso. Un flusso è non sanitizzato se esiste almeno un percorso di esecuzione che conduce da *src* al *sink* senza transitare per alcun `JUMPI` classificato come *sanitizer* per il *program point src*.

La condizione *all-paths* corrisponde alla semantica della sicurezza: un sistema di autorizzazione è efficace soltanto quando non esiste alcun percorso di aggiramento. Se una guardia protegge alcune traiettorie di esecuzione ma non tutte, un attaccante sceglie il percorso non protetto, quello abilitato da condizioni di stato che aprono sequenze di salto alternative. Una verifica esistenziale, che dichiarasse sicuro un flusso in presenza di almeno un percorso protetto, produrrebbe falsi negativi per tutti i *bypass* raggiungibili attraverso cammini alternativi.

Nei *cross-chain bridge* questo aspetto è particolarmente esposto: l'esecuzione asincrona su più catene introduce percorsi di controllo con guardie differenti a seconda dello stato del contratto al momento dell'invocazione. Un messaggio ricevuto quando una guardia non è ancora attiva, o attraverso codice di inizializzazione raggiungibile prima della configurazione completa del sistema, può aggirare controlli che sarebbero altrimenti applicati. La condizione *all-paths* cattura questi casi: richiede che ogni percorso di esecuzione presenti una guardia adeguata, non soltanto i percorsi più frequenti o quelli testati in condizioni ordinarie.

Il CFG *sound* prodotto dalla procedura di costruzione presentata nel Capitolo 4 include ogni percorso di esecuzione realmente raggiungibile: il controllo *all-paths* su tale CFG è quindi equivalente alla verifica sulla semantica concreta del programma [2, 3]. Un percorso di esecuzione reale che aggirasse un controllo è necessariamente presente nel CFG *sound* e viene rilevato dall'analisi. L'assenza di falsi negativi del *checker* dipende direttamente dalla *soundness* del CFG.

5.3 Implementazione

Il *checker* è integrato in EVMLiSA come secondo stadio di analisi, che opera sul CFG *sound* prodotto dalla procedura di costruzione presentata nel Capitolo 4. Le sue componenti sono il CFG, i risultati dell'analisi $\mathcal{T}^{\mathsf{pp}}$ calcolata sul CFG, e una mappa opzionale di annotazioni dei *modifier*. Il sistema riceve in ingresso il bytecode EVM di un singolo contratto, l'ABI del contratto in analisi e,

facoltativamente, la mappa dei *modifier* Solidity; produce in uscita l'insieme delle vulnerabilità di *Access Control Incompleteness* rilevate nel contratto.

5.3.1 Integrazione con EVMLiSA

L'architettura prevede due stadi in sequenza, come illustrato nella Figura 5.1. Il primo stadio è la costruzione del CFG descritta nel Capitolo 4: EVMLiSA esegue l'analisi astratta sul bytecode del contratto con il dominio parametrico degli *stack* astratti a valori in $\mathbb{Z}^\sharp$, risolve le destinazioni di salto dinamiche e produce il CFG *sound* [3]. Il secondo stadio prende in ingresso quel CFG ed esegue l'analisi *taint* relazionale con il dominio $\mathcal{T}^{\mathsf{pp}}$: propaga le etichette di provenienza attraverso le istruzioni del contratto e individua i flussi non sanitizzati verso i *sink*.

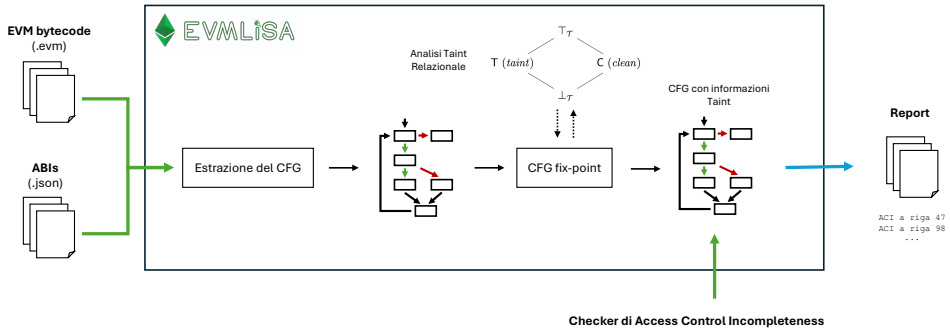


Figura 5.1: Architettura di EVMLiSA per la rilevazione di *Access Control Incompleteness*.

I due stadi condividono la struttura del CFG, ma operano su domini astratti distinti. Il primo utilizza il dominio degli *stack* astratti a valori in $\mathbb{Z}^\sharp$ per approssimare i valori concreti necessari a risolvere le destinazioni di salto; il secondo utilizza il dominio $\mathcal{T}^{\mathsf{pp}}$ per approssimare la provenienza dei valori. Tenendo i domini separati, le funzioni di trasferimento del secondo stadio si occupano esclusivamente del tracciamento della provenienza e non interferiscono con la logica di costruzione del CFG. Il *checker* produce in uscita un insieme di coppie $(src, sink)$, dove src è il *program point* della *source* e $sink$ è il *program point* dell'istruzione *sink* raggiunta da un flusso non sanitizzato.

5.3.2 Recupero degli entry point basato su ABI

Un contratto EVM espone le proprie funzioni al mondo esterno attraverso un *dispatcher*: una sequenza di diramazioni che, all'inizio di ogni transazione,

esamina i primi quattro byte del *calldata* e seleziona il blocco di codice corrispondente alla funzione richiesta. Questi quattro byte costituiscono il *selector* della funzione, calcolato come i primi quattro byte del keccak-256 della firma della funzione stessa. Il bytecode del contratto non contiene informazioni esplicite sui confini tra le diverse funzioni: la struttura del *dispatcher* è l'unica traccia della separazione tra i punti di ingresso, e la sua decodifica richiede la conoscenza delle firme delle funzioni, che non sono incluse nel bytecode ma nell'ABI.

L'ABI fornisce la mappatura tra i *selector* e le firme delle funzioni, con i relativi tipi dei parametri. EVMLiSA utilizza l'ABI per due scopi: identificare gli indirizzi di ingresso nel bytecode corrispondenti a ciascuna funzione esterna del *dispatcher*, e determinare gli *offset* di `CALLDATALOAD` che caricano i singoli parametri di ciascuna funzione. Questa fase di inizializzazione è necessaria affinché l'analisi *taint* contrassegni esattamente le istruzioni `CALLDATALOAD` corrette come *source* all'inizio del corpo di ciascuna funzione, senza perdere *source* pertinenti né contrassegnare come *tainted* dati che non sono parametri della funzione.

L'ABI è generalmente disponibile in due situazioni: per i contratti verificati e pubblicati su blockchain explorer come Etherscan, è scaricabile tramite API pubbliche; per i contratti compilati da sorgente nell'ambito della pipeline di valutazione, è prodotta automaticamente dal compilatore Solidity come artefatto di compilazione. Quando l'ABI non è disponibile, il *checker* opera in modalità degradata, identificando le *source* senza disaggregazione per funzione. Nella configurazione di riferimento adottata per la valutazione sperimentale, l'ABI è sempre fornita come input obbligatorio.

5.3.3 Interfaccia di annotazione dei modifier

Solidity consente di incapsulare le guardie di autorizzazione in costrutti dichiarativi chiamati *modifier*: un *modifier* è una funzione speciale annotata con il simbolo `modifier`, il cui corpo contiene la logica di controllo e che viene applicata a una o più funzioni del contratto mediante una dichiarazione esplicita nella firma. Lo Snippet di Codice 9 riporta il *modifier* `only_owner` del contratto `PublicResolver`: la funzione `setContent` delega il controllo di accesso al *modifier*, che verifica l'identità del chiamante prima di permettere l'esecuzione del corpo della funzione.

Il compilatore Solidity inserisce il corpo del *modifier* all'inizio della funzione durante la compilazione: nel bytecode prodotto, la logica di guardia è iniettata nel corpo della funzione senza alcuna delimitazione. Il confine tra codice del *modifier* e codice della funzione non è preservato, e l'analisi bytecode non

Algoritmo 9 Esempio di *modifier* Solidity: `only_owner`.

```
function setContent(bytes32 node, bytes32 hash)
  only_owner(node) public {
  records[node].content = hash;
  ContentChanged(node, hash);
}

modifier only_owner(bytes32 node) {
  if (ens.owner(node) != msg.sender) throw;
  _;
}
```

dispone di alcuna marcatura esplicita che distingua le istruzioni di guardia dalle istruzioni funzionali.

Il *checker* a livello bytecode non può quindi distinguere una `JUMPI` che implementa una guardia di autorizzazione iniettata mediante *modifier* da qualsiasi altra diramazione condizionale nella logica del contratto. Come mostrato nel Capitolo 6, questa opacità spiega una quota rilevante dei falsi positivi nella configurazione senza annotazioni: il *checker* segnala vulnerabilità in funzioni protette da un *modifier* Solidity la cui guardia è presente nel bytecode ma non è riconoscibile come tale a livello di opcode.

Per affrontare questo problema, si introduce un'interfaccia di annotazione dei *modifier* opzionale: il *checker* accetta in ingresso, oltre al bytecode e all'ABI, una mappa che associa ciascun *entry point* di funzione ai *modifier* dichiarati su di essa a livello sorgente Solidity. Nell'implementazione di riferimento, la mappa è estratta automaticamente dall'*Abstract Syntax Tree* (AST) prodotto dal compilatore Solidity `solc` come artefatto standard della compilazione.

Il criterio di soppressione adottato è conservativo: una vulnerabilità è soppressa soltanto quando la mappa certifica che la funzione è decorata con un *modifier* corrispondente a una guardia di autorizzazione che domina il *sink* nel CFG e che abortisce l'esecuzione in caso di fallimento del controllo. Quando l'analisi incontra guardie che non seguono questo schema, ad esempio *modifier* il cui comportamento in caso di fallimento non sia l'aborto o annotazioni la cui interpretazione non sia univoca, la soppressione non viene attivata: il *checker* segnala la vulnerabilità, preferendo un eventuale falso positivo alla perdita di un falso negativo.

Le annotazioni eliminano la classe di falsi positivi causati dalla perdita di semantica dei *modifier* durante la compilazione: la precisione aumenta senza scartare vulnerabilità reali. I dati sperimentali completi, il confronto con SmartAxe [19] e l'analisi per ciascun *bridge* analizzato sono nel Capitolo 6.

5.3.4 Algoritmo di rilevamento a due fasi

Il *checker* implementa un algoritmo a due fasi che combina la propagazione *taint* relazionale con la raggiungibilità nel CFG, come mostrato nell'Algoritmo 10. La prima fase identifica le istruzioni `JUMPI` nel CFG che possono agire come *sanitizer*, determinando per ciascuna l'insieme dei *program point* delle *source* che essa controlla. La seconda fase esamina ogni istruzione *sink* nel CFG e verifica che ogni flusso *tainted* che la raggiunge attraversi, su ogni percorso di esecuzione, almeno un *sanitizer* pertinente. La struttura relazionale di $\mathcal{T}^{\mathsf{pp}}$ permette questa distinzione, poiché associa a ciascun valore *tainted* l'insieme preciso dei *program point* di provenienza.

Algoritmo 10 Checker di *Access Control Incompleteness*

Input: CFG G, risultati analisi *taint* relazionale $\mathcal{T}^{\mathsf{pp}}$, mappa *modifier* opzionale $\mathcal{M}$
Output: insieme di vulnerabilità V

$sanitizers \leftarrow \emptyset$
$V \leftarrow \emptyset$

Fase 1: Identificazione dei *sanitizer*
for all $stmt \in G$ tale che $stmt$ è `JUMPI` **do**
    $stack \leftarrow \mathcal{T}^{\mathsf{pp}}$.`getStackBefore`($stmt$)
    $e_{dst}, e_{cond} \leftarrow stack$.`getJUMPIOperands`()
    if e_{cond} è *tainted*, cioè $\pi_1(e_{cond}) = \mathsf{T}$ **then**
        $sanitizers[stmt] \leftarrow \pi_2(e_{cond})$

Fase 2: Verifica dell'isolamento di ogni *sink*
for all $sink \in G$ **do**
    $stack \leftarrow \mathcal{T}^{\mathsf{pp}}$.`getStackBefore`($sink$)
    if almeno un argomento di $sink$ è *tainted* in $stack$ **then**
        $sources \leftarrow$ istruzioni *source* da cui $sink$ è raggiungibile in G
        for all $src \in sources$ **do**
            $srcPP \leftarrow src$.`programCounter`()
            $domJumpis \leftarrow G$.`getDominatingJUMPIs`($src, sink$)
            $guardedOnAllPaths \leftarrow$ **false**
            if $\exists\, j \in domJumpis : j \in sanitizers \wedge srcPP \in sanitizers[j]$ **then**
                $hasBypass \leftarrow G$.`existsBypassPath`($src, sink, srcPP, sanitizers$)
                $guardedOnAllPaths \leftarrow \neg\, hasBypass$
            $modGuard \leftarrow \mathcal{M}$.`validatedGuard`($sink.function, sink, srcPP$)
            if $\neg\, guardedOnAllPaths \wedge \neg\, modGuard$ **then**
                $V \leftarrow V \cup \{(src, sink)\}$
return V

Inizializzazione. L'algoritmo inizializza due strutture vuote (righe 4–5): *sanitizers*, una mappa che associerà a ogni istruzione `JUMPI` l'insieme dei *program point* la cui provenienza essa controlla, e V, l'insieme delle vulnerabilità rilevate.

Fase 1: identificazione dei sanitizer. La prima fase (righe 7–12) itera su tutti gli opcode del CFG che corrispondono a un'istruzione `JUMPI` (riga 8). Per ciascuna istruzione si recupera lo stato astratto dello *stack* nell'istante immediatamente precedente la sua esecuzione interrogando i risultati dell'analisi $\mathcal{T}^{\mathsf{pp}}$ (riga 9), quindi si estraggono i due operandi della `JUMPI`: la destinazione di salto e_{dst} e la condizione booleana e_{cond} (riga 10). Si verifica poi se e_{cond} sia *tainted*, ovvero se $\pi_1(e_{cond}) = \mathsf{T}$ (riga 11): solo quando il salto è controllato da dati di provenienza esterna la diramazione costituisce un potenziale controllo di autorizzazione. La condizione di *taint* è verificata sul solo operando di condizione e non su quello di destinazione: una `JUMPI` la cui destinazione dipende da dati *tainted* ma la cui condizione è *clean* non è un controllo di autorizzazione, bensì un salto il cui target è determinato da input esterni. Quando la condizione è soddisfatta, l'insieme $\pi_2(e_{cond})$ dei *program point* che hanno contribuito alla condizione è registrato in $sanitizers[stmt]$ (riga 12): la `JUMPI` è dunque nota come *sanitizer* per i flussi provenienti da quei *program point* specifici.

Fase 2: verifica di ogni sink. La seconda fase (righe 14–28) itera su ogni opcode del CFG classificato come *sink* secondo la Tabella 5.1 (riga 15). Per ciascun *sink* si recupera lo stato astratto dello *stack* prima della sua esecuzione (riga 16) e si verifica se almeno un argomento sia *tainted* (riga 17): in caso negativo il *sink* non è raggiunto da dati di provenienza esterna e si passa al successivo. Quando la condizione è soddisfatta, si raccolgono le istruzioni *source*, ovvero gli opcode classificati come tali nella Tabella 5.1, da cui il *sink* è raggiungibile nel CFG secondo i risultati di $\mathcal{T}^{\mathsf{pp}}$ (riga 18). Per ciascuna *source src* (riga 19) si registra il suo *program counter* come *srcPP* (riga 20), si recuperano le istruzioni `JUMPI` che dominano il *sink* su tutti i percorsi a partire da *src* (riga 21) e si inizializza *guardedOnAllPaths* a **false** (riga 22). Se esiste almeno una `JUMPI` dominante che sia un *sanitizer* per *srcPP* (riga 23), si esegue la verifica esplicita di *bypass* controllando se esista un percorso da *src* al *sink* che eviti tutti i *sanitizer* pertinenti (riga 24): *guardedOnAllPaths* è posto a **true** soltanto se non esiste alcun percorso di *bypass* (riga 25), realizzando così la proprietà *all-paths* descritta nella Sezione 5.2.4. Si interroga infine la mappa $\mathcal{M}$ per verificare se un *modifier* Solidity fornisca una guardia valida per quel *sink* rispetto a *srcPP* (riga 26). Se il flusso non è protetto né da *sanitizer* su tutti i percorsi né da un *modifier* validato (riga 27), la coppia (*src*, *sink*) è

aggiunta all'insieme delle vulnerabilità V (riga 28). Dopo aver elaborato tutti i *sink*, l'algoritmo restituisce V (riga 29).

Il controllo esplicito sul *bypass* realizza la proprietà *all-paths* descritta nella Sezione 5.2.4: non è sufficiente che esista un *sanitizer* dominante su alcuni percorsi; occorre che nessun percorso di esecuzione dalla *source* al *sink* possa evitarlo. La presenza di un CFG *sound* garantisce che ogni percorso di esecuzione realmente percorribile dal bytecode sia rappresentato nel CFG, e il controllo *all-paths* su quel CFG è quindi conservativo rispetto alla semantica concreta.

Soundness e assunzioni. L'assenza di falsi negativi del *checker*, intesa come conservatività rispetto alla semantica astratta adottata, vale sotto tre assunzioni.

1. Il CFG ricostruito da EVMLiSA sovra-approssima tutti i percorsi di esecuzione intra-contratto realmente eseguibili dal bytecode.

2. L'insieme di *source* e *sink* selezionato è *sound* rispetto al modello di minaccia dell'*Access Control Incompleteness* considerato.

3. Quando fornite, le annotazioni dei *modifier* sono corrette e corrispondono a guardie che dominano il *sink* e abortiscono l'esecuzione in caso di fallimento del controllo.

La prima assunzione è garantita dal teorema di *soundness* della costruzione del CFG dimostrato per EVMLiSA [2, 3]. La seconda è una precondizione del modello: se esistono opcode *source* o *sink* rilevanti per il modello di minaccia non inclusi negli insiemi della Tabella 5.1, il *checker* non rileva le vulnerabilità che coinvolgono tali opcode; l'insieme adottato è definito per coprire la classe di vulnerabilità considerata nei *cross-chain bridge*. La terza assunzione riguarda esclusivamente la componente opzionale di annotazione dei *modifier* descritta nella Sezione 5.3.3.

Capitolo 6

Valutazione Sperimentale

In questo capitolo si valuta sperimentalmente il *checker* per la rilevazione di Access Control Incompleteness implementato in EVMLiSA e descritto nel Capitolo 5. L'obiettivo è misurare l'efficacia dello strumento su applicazioni *bridge* reali rispetto a due aspetti: la capacità di rilevare tutte le istanze di vulnerabilità note e la *precision* dei *warning* emessi rispetto a vulnerabilità validate manualmente.

Gli incidenti discussi nella Sezione 5.1 mostrano il peso economico di questa classe di vulnerabilità: Nomad e Ronin hanno causato perdite superiori a 190 milioni e 625 milioni di dollari, rispettivamente [16, 17, 40, 6].

6.1 Risultati del Checker sugli Esempi Motivanti

Prima di presentare i risultati aggregati sull'intero benchmark, si riportano gli esiti del *checker* sui tre esempi motivanti illustrati nella Sezione 5.1. Questo collegamento consente di verificare che i criteri formali adottati nelle sezioni successive producano il comportamento atteso sui casi concreti.

Parity Multisig. Per il frammento di codice Parity Multisig mostrato nella Sezione 5.1.1, EVMLiSA classifica come contaminati i parametri `_owners` e `_required` della funzione `initWallet`: entrambi originano da istruzioni `CALLDATALOAD` e il loro flusso è tracciato fino alle istruzioni `SSTORE` che scrivono rispettivamente `owners` e `required`. Lungo questo percorso non compare alcuna istruzione `JUMPI` la cui condizione dipenda da `CALLER` e che possa agire da *sanitizer* per i flussi considerati. La funzione viene quindi segnalata come affetta da Access Control Incompleteness, in accordo con la proprietà *all-paths* definita nella Sezione 5.2.4.

Nomad Replica. Per il frammento del bridge Nomad presentato nella Sezione 5.1.2, gli input `_updater` e `_root` della funzione `initialize` scorrono verso le istruzioni `SSTORE` che aggiornano `updater` e `committedRoot`. L'unica diramazione condizionale presente verifica il flag `initialized`, non l'identità del chiamante: il flusso rimane quindi non sanitizzato e viene segnalato. Nella funzione `acceptMessage`, l'input `message` raggiunge il *sink* interno di elaborazione senza che alcun `JUMPI` pertinente sulla catena di dipendenze di controllo imponga una verifica sull'autorizzazione del chiamante: anche questa funzione viene segnalata come vulnerabile.

Ronin Bridge. Per l'esempio del Ronin presentato nella Sezione 5.1.3, l'input `payload` della funzione `withdraw` origina da un'istruzione `CALLDATALOAD` e scorre verso `executeWithdrawal`. Il percorso di controllo include un `require` su `verifySignatures`, ma tale verifica vincola `sigs` e non `payload`. La semantica *taint* relazionale tiene conto di questa distinzione: il *sanitizer* individuato non è contestualmente rilevante per il flusso verso il *sink*, e il *checker* segnala correttamente che `payload` raggiunge un *sink* sensibile senza un controllo di autorizzazione dedicato.

6.2 Il Benchmark di Valutazione

Non esiste un benchmark pubblico dedicato specificamente alla rilevazione di Access Control Incompleteness nei contratti *bridge cross-chain*. Il riferimento più vicino è SmartAxe [19], l'unico lavoro precedente noto a fornire un dataset sistematico etichettato per vulnerabilità *cross-chain*. Si adotta quindi il benchmark SmartAxe come suite di valutazione e l'analisi si concentra sulla sola categoria Access Control Incompleteness.

6.2.1 Il Dataset SmartAxe

SmartAxe [19] valuta un framework di analisi statica su 16 applicazioni *bridge* che comprendono complessivamente 1.003 *smart contract*, con 88 vulnerabilità *cross-chain* etichettate manualmente. Le etichette coprono Access Control Incompleteness, assenza di controlli, mancanza di validazione della ripetitività, inconsistenze semantiche e altre categorie di problemi di sicurezza.

La Tabella 6.1 riassume le statistiche strutturali del benchmark a livello di applicazione. Il corpus contiene 1.003 file Solidity e 109.952 linee di codice (LoC); le singole applicazioni variano da 1 a 305 file e da 585 a 18.865 LoC [19].

Tabella 6.1: Statistiche descrittive del benchmark: numero di file Solidity e distribuzione delle LoC per applicazione.

Applicazione	**# File (Solidity)**	**LoC Totali**	**LoC Media**	**LoC Min**	**LoC Max**
ChainSwap20210711	1	1.008	1.008	1.008	1.008
EvoDeFi20220607	15	733	49	4	289
LiFinance20220320	305	17.154	56	2	1.410
MeterPassPort20220206	32	8.050	252	10	1.315
Multichain20220118	117	18.865	161	4	1.049
Multichain20230215	117	18.714	160	4	1.049
Nerve20211115	28	5.739	205	10	1.003
pNetwork20210920	99	7.602	77	4	1.090
PolyNetwork20210810	80	8.639	108	0	985
QANXBridge20220518	7	585	84	7	191
QBridge20220128	20	1.181	59	11	194
Rubic20221225	30	1.841	61	4	400
Synapse20211106	108	15.533	144	4	1.003
THORChain20210629	14	1.252	89	17	167
THORChain20210716	15	1.378	92	17	167
THORChain20210723	15	1.378	92	17	167
Totale	**1.003**	**109.952**	**110**	–	–

6.2.2 Ricostruzione del Ground Truth

Una valutazione attendibile richiede una *ground truth* affidabile per le vulnerabilità di Access Control Incompleteness. Il tentativo di riprodurre le annotazioni originali di SmartAxe ha tuttavia incontrato un ostacolo: il lavoro pubblicato non documenta né la metodologia di etichettatura né la corrispondenza tra i singoli *warning* e i *smart contract* con un livello di dettaglio sufficiente a una replicazione diretta. Una verifica diretta è stata condotta sul repository GitHub del progetto,[1] da cui sono stati estratti 19 casi candidati di Access Control Incompleteness: l'ispezione manuale ha mostrato che soltanto 11 di questi corrispondono a vulnerabilità genuine, mentre i restanti 8 sono falsi positivi. Questa discrepanza segnala l'inaffidabilità delle annotazioni originariamente pubblicate; un confronto diretto con lo strumento SmartAxe eseguito sul medesimo benchmark è riportato nella Sezione 6.5.

Date queste inconsistenze, si è proceduto a una nuova validazione manuale dell'intero benchmark, ricostruendo la *ground truth* per le vulnerabilità di Access Control Incompleteness a partire dall'analisi del codice sorgente. Ogni applicazione *bridge* è stata ispezionata contratto per contratto, e ogni istanza inclusa nella *ground truth* finale è stata verificata con un'ispezione indipendente a livello di *sink* e un'analisi del flusso di controllo. Il processo ha individuato 301 vulnerabilità di Access Control Incompleteness.[2] Nella presente valutazio-

[1] `https://github.com/InPlusLab/FSE24-SmartAxe`.

[2] `https://github.com/lisa-analyzer/evm-lisa/blob/cross-chain/`

ne, un'istanza di vulnerabilità corrisponde a una funzione `public` o `external` per la quale il *checker* segnala almeno un flusso da una *source* non fidata a un *sink* sensibile non sanitizzato, in violazione della condizione di Access Control Incompleteness.

6.2.3 Policy di Source e Sink

La *policy* a livello di opcode adottata dal *checker*, definita nella Sezione 5.2.2 e riportata nella Tabella 5.1, distingue le istruzioni EVM che leggono input esterni, modellate come *source* di contaminazione, da quelle che effettuano chiamate esterne o modificano lo stato persistente del contratto, modellate come *sink* sensibili. A livello di funzione, lo scope dell'analisi è limitato alle funzioni `public` ed `external`, poiché soltanto questi punti di ingresso sono direttamente raggiungibili da chiamanti esterni non fidati.

6.3 Configurazione Sperimentale

Le 16 applicazioni *bridge* del dataset SmartAxe sono state analizzate con il *checker* implementato in EVMLiSA e descritto nella Sezione 5.3. L'analisi è stata configurata con le *source* e i *sink* elencati nella Tabella 5.1. Ogni *warning* emesso da EVMLiSA è stato ispezionato manualmente per classificarlo come *true positive*, corrispondente a una vulnerabilità reale di Access Control Incompleteness, o come *false positive*, corrispondente a un *warning* spurio. Per ogni *true positive* è stato verificato che un input non fidato raggiunga un *sink* sensibile senza attraversare un controllo di autorizzazione adeguato.

Tutte le analisi sono state eseguite sull'infrastruttura HPC dell'Università di Parma, su un nodo con 32 core CPU e 48 GB di RAM. La stessa configurazione hardware è stata impiegata per gli esperimenti in modalità bytecode + ABI e in modalità *modifier-aware*, così da rendere i risultati comparabili.

6.4 Risultati

I risultati complessivi sono riportati nella Tabella 6.2.

6.4.1 Configurazione Bytecode + ABI

In assenza di informazioni aggiuntive rispetto al bytecode EVM e all'ABI, EVMLiSA ha prodotto 411 *warning* sulle 16 applicazioni *bridge*: 301 *true positive* e 110 *false positive*, con una *precision* del 73,24% e una *recall* del

`datasets/cross-chain/tops/ground-truth.csv`.

100%. La *recall* del 100% indica che tutte le 301 vulnerabilità di Access Control Incompleteness presenti nella *ground truth* vengono segnalate, senza che alcuna istanza etichettata sfugga alla rilevazione. Una *precision* del 73,24% significa che circa tre quarti dei *warning* sono effettivamente rilevanti.

L'ispezione manuale dei 110 *false positive* ha mostrato che 92 di essi, corrispondenti all'83,6% del totale, condividono la medesima causa radice: la perdita di informazioni indotta dal compilatore descritta nella Sezione 5.3.3, per cui i *modifier* Solidity vengono espansi *inline* nel bytecode e la logica condizionale che essi introducono risulta indistinguibile, a livello di bytecode, da un qualunque ramo arbitrario. La configurazione bytecode + ABI, non disponendo delle annotazioni originali dei *modifier*, segnala in modo conservativo quelle funzioni che non può dimostrare protette.

6.4.2 Configurazione Modifier-Aware

L'abilitazione dell'interfaccia di annotazione dei *modifier*, popolata nell'implementazione a partire dall'output AST del compilatore come descritto nella Sezione 5.3.3, riduce i *false positive* da 110 a 18. La *precision* sale al 94,36% mentre la *recall* rimane al 100%, con un guadagno di oltre 21 punti percentuali rispetto alla configurazione bytecode + ABI. I 18 *false positive* residui corrispondono a casi limite: pattern di controllo degli accessi non standard e dipendenze inter-contrattuali complesse che la logica di soppressione conservativa non riesce a escludere.

Tabella 6.2: Risultati del *checker* di Access Control Incompleteness sulle 16 applicazioni *bridge*.

Approccio	**Warning**	**True Positive**	**False Positive**	**Precision**	**Recall**
EVMLiSA (bytecode + ABI)	411	301	110	73,24%	100%
EVMLiSA (+ modifier-aware)	319	301	18	94,36%	100%

6.5 Confronto con SmartAxe

A complemento dell'etichettatura manuale descritta nella Sezione 6.2.2, lo strumento SmartAxe [19] è stato eseguito sullo stesso benchmark per ottenere un confronto diretto.

La riproduzione dello strumento ha richiesto un lavoro considerevole. Il repository pubblico non fornisce una specifica delle dipendenze funzionante: diversi requisiti elencati non sono portabili tra ambienti differenti, l'implementazione si appoggia ad API Python rimosse nelle *release* recenti e numerosi *script* contengono *path* assoluti *hard-coded*. Una *build* funzionante è stata ottenuta

soltanto dopo aver ricostruito le versioni originali delle dipendenze, sostituito le chiamate ad API deprecate e adattato manualmente la configurazione dei *path*.

Sul benchmark dei *bridge*, SmartAxe non segnala alcuna vulnerabilità di Access Control Incompleteness, mentre EVMLiSA individua tutte le 301 istanze validate manualmente. Per escludere che questo esito nullo sia un artefatto della configurazione sperimentale, lo stesso binario di SmartAxe è stato valutato anche sui contratti bytecode forniti nel repository ufficiale dello strumento. In questo contesto, SmartAxe rileva correttamente 10 vulnerabilità legate al controllo degli accessi: lo strumento è compilato e operativo. L'assenza di risultati sul benchmark non è quindi riconducibile a problemi di configurazione o di esecuzione: essa indica un limite intrinseco nella capacità di rilevazione di SmartAxe su questo tipo di contratti.

6.6 Performance

La Tabella 6.3 riporta le statistiche di tempo di esecuzione per *bridge* di EVMLiSA sul benchmark considerato. Il tempo di esecuzione include la costruzione del *control-flow graph*, l'analisi *taint* relazionale e l'esecuzione del *checker*.

Il tempo mediano di esecuzione per *smart contract* è di 5,79 secondi; il valore medio è di 28,70 secondi. Lo scarto tra le due misure è dovuto a un numero limitato di contratti computazionalmente onerosi che alzano la media, mentre la maggior parte viene elaborata in pochi secondi. I tempi sono compatibili con un uso orientato all'*audit*.

I tempi di esecuzione variano sensibilmente tra le applicazioni *bridge*, in funzione della dimensione del codice e della complessità del flusso di controllo. I casi più costosi sono i due corpus Multichain, che richiedono rispettivamente 6.242 e 5.679 secondi in totale, con tempi medi per contratto di 104 e 96 secondi e tempi nel caso peggiore di 597 e 476 secondi. Applicazioni più leggere come Nerve20211115 e le varianti THORChain mostrano tempi medi per contratto compresi tra 4,62 e 6,87 secondi. Il divario tra media e mediana in applicazioni come LiFinance20220320 e pNetwork20210920 segnala una distribuzione asimmetrica, in cui un numero ridotto di contratti domina il tempo complessivo.

L'analisi scala ad applicazioni *bridge cross-chain* di dimensioni reali.

Tabella 6.3: Statistiche di tempo di esecuzione per *bridge*, espresse in secondi.

Bridge	**Totale App.**	**Min App.**	**Max App.**	**Media per SC**	**Mediana**
ChainSwap20210711	4,12	4,12	4,12	4,12	4,12
EvoDeFi20220607	133,38	0,09	71,16	12,13	8,69
LiFinance20220320	542,93	0,003	429,09	11,08	0,005
MeterPassPort20220206	2.100,46	0,01	349,44	40,39	3,57
Multichain20220118	6.242,57	0,04	596,72	104,04	59,24
Multichain20230215	5.679,29	0,04	475,76	96,26	57,51
Nerve20211115	263,31	0,03	12,88	4,62	3,67
pNetwork20210920	5.336,77	0,002	442,58	13,05	0,12
PolyNetwork20210810	317,92	0,01	85,82	13,25	2,57
QANXBridge20220518	79,15	0,03	33,76	11,31	4,20
QBridge20220128	305,37	15,40	206,57	61,07	17,87
Rubic20221225	349,34	39,62	117,95	58,22	48,60
Synapse20211106	331,87	0,01	136,97	9,22	3,85
THORChain20210629	116,34	0,004	14,95	6,84	8,88
THORChain20210716	116,80	3,45	11,66	6,87	7,38
THORChain20210723	115,40	0,004	14,06	6,79	8,37
Totale	**22.035,02**	–	–	**28,70**	**5,79**

Capitolo 7

Limitazioni e Discussione

Il *checker* per la rilevazione di Access Control Incompleteness presentato nel Capitolo 5 e valutato nel Capitolo 6 è uno strumento di analisi statica *sound* applicabile a *smart contract* reali in scenari di *audit* della sicurezza. Come ogni analisi statica basata su interpretazione astratta, opera su un'approssimazione del comportamento concreto del programma e presenta margini di imprecisione che questo capitolo discute esplicitamente.

Le sezioni seguenti trattano: l'opacità dei *modifier* di Solidity nella configurazione senza annotazioni, le assunzioni sull'atomicità delle transazioni EVM, la dipendenza da metadati verificati off-chain e il costo computazionale dell'analisi su corpus di grandi dimensioni.

7.1 Opacità dei modifier

La prima limitazione riguarda la configurazione dell'analisi che non ricostruisce esplicitamente i *guard* introdotti dai *modifier* di Solidity nella configurazione *senza annotazioni*. In questa modalità, i controlli di autorizzazione implementati tramite *modifier* possono diventare opachi a livello di *bytecode* e risultare invisibili al *checker* nel punto in cui si osservano le operazioni di aggiornamento dello stato.

Un caso rappresentativo è la funzione `setContent`, definita nel contratto `PublicResolver` del *bridge pNetwork20210920*. Come illustrato nello Snippet di Codice 11, la funzione accetta i parametri `node` e `hash` e aggiorna il campo `records[node].content` tramite un'assegnazione diretta. La guardia di autorizzazione è espressa dal *modifier* `only_owner(node)`, che verifica che `ens.owner(node)` corrisponda a `msg.sender` e in caso contrario solleva un'eccezione.

Il compilatore Solidity risolve i *modifier* mediante *inlining*: il corpo del *modifier* viene espanso *inline* nel corpo della funzione prima della compilazio-

Algoritmo 11 `setContent` di `PublicResolver`: il *guard* `only_owner(node)` non è riconoscibile come *sanitizer* a livello di *bytecode* senza annotazioni dei *modifier*.

```
1 function setContent(bytes32 node, bytes32 hash) only_owner(node) public {
2     records[node].content = hash;
3     ContentChanged(node, hash);
4 }
5
6 modifier only_owner(bytes32 node) {
7     if (ens.owner(node) != msg.sender) throw;
8     _;
9 }
```

ne in *bytecode*. Di conseguenza, la verifica `ens.owner(node) != msg.sender` è presente nel *bytecode* generato, ma la sua relazione semantica con il ruolo di guardia di accesso non è esplicitamente etichettata come tale a livello di istruzioni EVM. Nella configurazione senza annotazioni dei *modifier*, il *checker* osserva un'istruzione `SSTORE` raggiunta da un flusso di dati originato da `CALLDATALOAD`, senza che lungo tale percorso sia presente un `JUMPI` riconosciuto come *sanitizer* pertinente per quel flusso. Il *warning* viene pertanto emesso, e si tratta di un falso positivo.

La configurazione con annotazioni dei *modifier*, descritta nella Sezione 5.3.3 del Capitolo 5, risolve questo problema mediante un controllo di soppressione esterno al flusso di analisi: prima di emettere un *warning* per un dato *sink*, il *checker* consulta la mappa $\mathcal{M}$ delle annotazioni dei *modifier* e, se la funzione contenente il *sink* risulta decorata con un *modifier* validato che agisce come guardia di autorizzazione, il *warning* non viene sollevato. La logica di soppressione, formalizzata nelle righe 26–27 dell'Algoritmo 10 e descritta in dettaglio nella Sezione 5.3.3, è conservativa: la soppressione si attiva soltanto quando l'annotazione certifica un *modifier* che domina il *sink* e che abortisce l'esecuzione in caso di fallimento del controllo. I risultati sperimentali del Capitolo 6 mostrano che questa configurazione raggiunge una *precision* del 94,36%, confermando l'efficacia dell'interfaccia di annotazione nel contenere questa classe di falsi positivi.

7.2 Assunzioni sull'atomicità delle transazioni

Una seconda fonte residua di falsi positivi emerge dalla natura dell'analisi statica a livello di *bytecode*, che non incorpora un modello dell'atomicità delle transazioni EVM. Nell'EVM, ogni transazione è atomica: se una transazione non termina con successo, tutte le modifiche di stato prodotte durante la

sua esecuzione vengono annullate tramite il meccanismo di *revert*. Il *checker*, operando a livello di *bytecode* senza tale modello, tratta in modo conservativo ogni istruzione `SSTORE` non sanitizzata come un'istanza di Access Control Incompleteness, indipendentemente dal fatto che una verifica di autorizzazione successiva nel flusso di esecuzione possa interrompere la transazione e annullare la scrittura.

Un caso illustrativo è la funzione `_permit` nel contratto `FiatTokenV2.sol`, appartenente al corpus del *bridge PolyNetwork 2021-08-10*, riportata nello Snippet di Codice 12. Il parametro `owner` è fornito dal chiamante e viene classificato come contaminato dall'analisi *taint*.

Algoritmo 12 `_permit` in `FiatTokenV2.sol`: l'`SSTORE` sul *nonce* precede la verifica della firma, ma il *revert* in caso di firma non valida annulla atomicamente la scrittura.

```
1  function _permit(
2      address owner, address spender,
3      uint256 value, uint256 deadline,
4      uint8 v, bytes32 r, bytes32 s
5  ) internal {
6      require(deadline >= now, "FiatTokenV2: permit is expired");
7      bytes memory data = abi.encode(
8          PERMIT_TYPEHASH, owner, spender, value,
9          _permitNonces[owner]++,  // SSTORE on unvalidated owner
10         deadline
11     );
12     require(  // signature check
13         EIP712.recover(DOMAIN_SEPARATOR, v, r, s, data) == owner,
14         "EIP2612: invalid signature"
15     );
16     _approve(owner, spender, value);
17 }
```

All'interno della chiamata `abi.encode` alla riga 7 del codice, la sottoespressione `_permitNonces[owner]++` emette un'istruzione `SSTORE` su uno slot di *storage* indicizzato da `owner` prima che tale valore sia stato validato. In quel punto del *bytecode*, nessuna verifica di autorizzazione è ancora stata eseguita. Il *checker* segnala pertanto questo `SSTORE` come un'istanza di Access Control Incompleteness, poiché `owner` raggiunge un *sink* di modifica dello stato senza che un ramo condizionale pertinente abbia prima verificato l'identità del chiamante. Il *warning* è corretto rispetto alla semantica dell'analisi.

In pratica, tuttavia, questa specifica istanza non è sfruttabile. La verifica della firma alla riga 12 costituisce una guardia effettiva: se la firma non è valida, la transazione fallisce e il meccanismo di *revert* dell'EVM annulla l'incremento del *nonce* insieme a ogni altra modifica di stato prodotta durante l'esecuzione. Nessuna modifica permanente sopravvive a una transazione

fallita. Il *checker*, non disponendo di un modello dell'atomicità a livello di *bytecode*, applica la regola conservativa per cui qualsiasi `SSTORE` non sanitizzato costituisce una vulnerabilità potenziale: si tratta di un compromesso intrinseco all'analisi statica a questo livello di astrazione. Risolvere questa classe di falsi positivi richiederebbe l'integrazione di un modello di raggiungibilità a livello di transazione o di una semantica dell'intera esecuzione; tale integrazione si individua come direzione di lavoro futuro.

7.3 Disponibilità dei metadati

La terza limitazione riguarda la disponibilità di artefatti off-chain necessari per la configurazione con annotazioni dei *modifier*. Come descritto nella Sezione 5.3.3, tale configurazione richiede l'accesso a sorgenti Solidity verificate, agli output di compilazione, o all'AST prodotto dal compilatore, per recuperare la semantica dei *modifier* di autorizzazione non rappresentata esplicitamente nel *bytecode* compilato.

Questi metadati sono reperibili per i contratti il cui codice sorgente è stato verificato e pubblicato su registri pubblici come Etherscan; per tali contratti, la corrispondenza tra *bytecode* distribuito e sorgente Solidity è attestata da un processo di verifica formale della compilazione. Tuttavia, non tutti i contratti distribuiti sulla blockchain hanno il sorgente verificato. Nei casi di *deployment* non verificati, o in presenza di metadati inaffidabili, la configurazione con annotazioni non può essere applicata in modo sicuro.

In queste circostanze, l'analisi ricade sulla configurazione *bytecode*+ABI di base, che non dipende da artefatti aggiuntivi oltre all'interfaccia binaria del contratto. Questa configurazione preserva le garanzie di *soundness* dell'analisi, nel senso che non introduce falsi negativi rispetto al modello adottato, ma opera con una *precision* inferiore rispetto alla configurazione con annotazioni, come mostrato quantitativamente nei risultati del Capitolo 6.

7.4 Considerazioni sulla scalabilità

Una quarta considerazione riguarda la scalabilità dell'analisi in funzione della complessità dei contratti e della dimensione dei corpus applicativi. Il tempo di esecuzione di EVMLiSA dipende dalla complessità del *control-flow graph* costruito e dai parametri configurabili del dominio astratto, in particolare dall'altezza massima degli *stack* astratti h e dal numero massimo di *stack* astratti per punto di programma l, descritti nel Capitolo 4.

Come documentato nella Sezione 6.6, la maggior parte dei contratti viene analizzata in pochi secondi: il tempo mediano di esecuzione sull'intero *bench-*

mark è di 5,79 secondi per contratto, un valore compatibile con l'impiego dello strumento in contesti di *audit* della sicurezza. La media, pari a 28,70 secondi per contratto, è sensibilmente più alta della mediana per effetto di un numero limitato di contratti computazionalmente onerosi, a indicare una distribuzione dei costi asimmetrica con una coda destra pronunciata.

Come riportato nella Sezione 6.6 e nella Tabella 6.3, i corpus più grandi del *benchmark* dominano il costo complessivo dell'analisi: la complessità non cresce in modo uniforme con la dimensione del corpus e un sottoinsieme ristretto di contratti può determinare la quasi totalità del tempo di esecuzione.

I risultati sperimentali confermano che l'approccio è praticabile su *bridge cross-chain* reali in contesti di *audit* della sicurezza, dove il tempo di analisi non è determinante. Per scenari di integrazione continua che richiedano un *feedback* rapido, il costo computazionale dei contratti più complessi può costituire un limite operativo. Due direzioni di lavoro concorrono a ridurre questo costo: ottimizzare la costruzione del CFG per i contratti più densi e individuare configurazioni parametriche del dominio astratto che bilancino *precision* e velocità in contesti di analisi incrementale.

Capitolo 8

Conclusione

8.1 Riepilogo dei contributi

La presente tesi affronta il problema del rilevamento statico di vulnerabilità di Access Control Incompleteness nei contratti che compongono le applicazioni di *bridge cross-chain* su Ethereum. Il contributo principale è un *checker* che, a partire dal *bytecode* EVM e dall'*ABI* del contratto, identifica i flussi di dati non fidati che raggiungono *sink* sensibili senza attraversare un controllo di autorizzazione adeguato.

L'analisi richiede un *Control-Flow Graph sound* del contratto: senza una rappresentazione completa del flusso di controllo, un singolo cammino non analizzato potrebbe nascondere una scrittura non autorizzata sullo stato persistente, e le garanzie di copertura del *checker* verrebbero meno. Per questa ragione, il Capitolo 4 richiama la costruzione *sound* dei CFG da *bytecode* EVM tramite EVMLiSA, basata sul dominio astratto parametrico Σ_l^h, che combina interi k-bounded, *stack* astratti di altezza massima h e insiemi di al più l *stack* astratti, e che consente di risolvere anche le destinazioni dei cosiddetti *orphan jump*. Questa infrastruttura è il prerequisito tecnico su cui si basa l'analisi sviluppata in questa tesi.

Il contributo metodologico, presentato nel Capitolo 5, consiste in un'analisi *taint* relazionale con tracciamento della provenienza a livello di *program point*. Sopra di essa si articola un *checker* a due fasi: nella prima si identificano le `JUMPI` la cui condizione dipende da valori *tainted*, interpretandole come *sanitizer* di autorizzazione associati ai *program point* delle *source* che esse controllano; nella seconda si verifica, per ogni *sink*, che ogni cammino di esecuzione dalla *source* attraversi almeno uno di questi *sanitizer*. A questa logica di flusso si affianca un'interfaccia di annotazione opzionale per i *modifier* di Solidity, che consente di sopprimere in modo conservativo i *warning* sulle funzioni decorate con guardie validate, contenendo la classe di falsi positivi

prodotti dall'*inlining* dei *modifier* durante la compilazione.

La valutazione sperimentale, riportata nel Capitolo 6, è stata condotta su 16 applicazioni di *bridge cross-chain* reali, per un totale di 1.003 contratti e 109.952 righe di *bytecode*, con una *ground truth* di 301 vulnerabilità validate manualmente. Il *checker* raggiunge una *recall* del 100% in entrambe le configurazioni e una *precision* che sale dal 73,24% nella configurazione *bytecode+ABI* al 94,36% quando si abilita l'interfaccia di annotazione dei *modifier*. Il ragionamento sulla provenienza dei valori a livello di *bytecode* risulta dunque sufficiente per identificare in modo *sound* l'assenza di controlli di autorizzazione, e l'integrazione di annotazioni leggere sui *modifier* riduce i falsi positivi senza intaccare la copertura. L'analisi è quindi impiegabile in flussi di *auditing* della sicurezza in cui la minimizzazione dei falsi negativi è il requisito prioritario.

8.2 Lavori futuri

La prima direzione riguarda l'estensione dell'analisi al contesto inter-contratto e *cross-chain*. La costruzione attuale dei CFG opera all'interno dei confini di un singolo contratto, lasciando irrisolte le destinazioni delle istruzioni parametriche di invocazione esterna, come `CALL`, `STATICCALL` e `DELEGATECALL`. Risolvere questi *target* richiederebbe di approssimare i valori dei relativi argomenti a *runtime* e di collegare il CFG del contratto chiamante con quello del contratto chiamato, estendendo l'analisi oltre i confini del singolo *deployment*. Analogamente, nel contesto *cross-chain*, la risoluzione dei *target* delle invocazioni tra catene diverse richiede un'approssimazione dei valori degli eventi e dei messaggi scambiati tra i *layer* di interoperabilità [14]. Una tale estensione generalizzerebbe il modello attuale alle architetture di *bridge* studiate in questa tesi.

Sul versante del supporto multi-linguaggio, il *framework* LiSA [24] è progettato per supportare *frontend* multipli per linguaggi eterogenei, il che rende possibile estendere EVMLiSA all'analisi di *smart contract* scritti in linguaggi non basati su EVM. Tra gli ambienti già esplorati nell'ecosistema LiSA figurano Hyperledger Fabric e Cosmos [31] e la macchina virtuale Michelson di Tezos [29]. In uno scenario di interoperabilità *cross-chain*, dove catene diverse adottano macchine virtuali diverse, un'analisi uniforme su linguaggi eterogenei coprirebbe una superficie di verifica molto più ampia rispetto agli approcci focalizzati su una singola piattaforma.

Un terzo sviluppo riguarda l'introduzione di nuovi *checker* di vulnerabilità all'interno di EVMLiSA. La stessa infrastruttura di analisi è applicabile ad

altre classi di vulnerabilità degli *smart contract*, fra cui la dipendenza dal *timestamp* del blocco e la dipendenza da valori pseudo-casuali generati *on-chain*, che espongono i contratti a manipolazioni da parte dei miner [4]. L'architettura del *checker* è inoltre estendibile a vulnerabilità specifiche dei *bridge* al di là dell'Access Control Incompleteness, verso un *framework* statico di sicurezza più ampio per i sistemi *cross-chain*.

Una quarta direzione riguarda la modellazione dell'atomicità delle transazioni. Come discusso nel Capitolo 7, tra i falsi positivi residui figurano quei percorsi di esecuzione in cui una modifica di stato, segnalata come potenzialmente non autorizzata, viene di fatto annullata atomicamente da un fallimento di autorizzazione che si verifica in una fase successiva della stessa transazione. Integrare nel *checker* un modello di raggiungibilità a livello di transazione eliminerebbe questa classe di falsi positivi senza ridurre la *recall*.

Disponibilità dei materiali. Il codice sorgente di EVMLiSA è disponibile pubblicamente nel *repository* GitHub ufficiale dello strumento. La suite di *benchmark* utilizzata nella valutazione sperimentale, insieme alla *ground truth* di 301 vulnerabilità validate manualmente, è accessibile nel *branch* `cross-chain` del medesimo *repository*.[1]

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

Appendice A

Dimostrazioni

Le dimostrazioni dei teoremi e lemmi presentati nella Sezione 4.4 del Capitolo 4 sono raccolte in questa appendice.

A.1 Struttura reticolare del dominio astratto

Dimostrazione del Teorema 4.4.1. Il dominio $\Sigma^{h,l}$ è il sottoinsieme del reticolo delle parti $\langle \wp(\Sigma^h), \subseteq, \cup, \cap, \emptyset, \Sigma^h \rangle$ che esclude tutti gli elementi $X \in \wp(\Sigma^h)$ tali che $X \neq \Sigma^h$ e $|X| > l$. L'elemento $\top_{\Sigma^{h,l}}$ coincide con Σ^h e tiene luogo di tutti gli insiemi esclusi.

La chiusura rispetto a $\sqcup_{\Sigma^{h,l}}$ è garantita per costruzione: l'operatore restituisce $\widehat{S}_1 \cup \widehat{S}_2$ se la cardinalità dell'unione non supera l, e $\top_{\Sigma^{h,l}}$ altrimenti; in entrambi i casi il risultato appartiene a $\Sigma^{h,l}$. La chiusura rispetto a $\cap$ discende dal fatto che l'intersezione di due insiemi di cardinalità $\leq l$ ha cardinalità $\leq l$.

Per quanto riguarda la completezza, l'elemento $\emptyset$ è il minimo e $\top_{\Sigma^{h,l}}$ è il massimo; ogni sottoinsieme di $\Sigma^{h,l}$ ha pertanto estremo superiore, ottenuto applicando iterativamente $\sqcup_{\Sigma^{h,l}}$, ed estremo inferiore, ottenuto come intersezione.

La condizione di catene ascendenti segue dall'altezza finita del reticolo: l'elemento minimo $\emptyset$ ha cardinalità zero; ogni elemento successivo in una catena crescente aggiunge almeno un elemento, incrementando la cardinalità di almeno uno; la cardinalità è limitata superiormente da l; al di sopra vi è solo $\top_{\Sigma^{h,l}}$. La lunghezza massima di una catena ascendente è dunque $l + 2$, il che garantisce che ogni iterazione di punto fisso converge in un numero finito di passi. □

A.2 Monotonia della funzione di concretizzazione

Dimostrazione del Teorema 4.4.2. Si supponga $\widehat{S}_1 \subseteq \widehat{S}_2$. Senza perdita di generalità si scrive $\widehat{S}_2 = \widehat{S}_1 \cup \widehat{S}'$ per qualche insieme $\widehat{S}'$ non vuoto. Espandendo la definizione di γ fornita nell'Equazione (4.12):

$$\begin{aligned}\gamma(\widehat{S}_2) &= \gamma(\widehat{S}_1 \cup \widehat{S}') \\ &= \bigcup_{\hat{\sigma} \in \widehat{S}_1 \cup \widehat{S}'} \overline{\gamma}(\hat{\sigma}) \\ &= \Big(\bigcup_{\hat{\sigma} \in \widehat{S}_1} \overline{\gamma}(\hat{\sigma})\Big) \cup \Big(\bigcup_{\hat{\sigma} \in \widehat{S}'} \overline{\gamma}(\hat{\sigma})\Big) \\ &= \gamma(\widehat{S}_1) \cup \gamma(\widehat{S}').\end{aligned}$$

Poiché $\gamma(\widehat{S}') \supseteq \emptyset$, si conclude che $\gamma(\widehat{S}_2) \supseteq \gamma(\widehat{S}_1)$. □

A.3 Correttezza della semantica astratta

Dimostrazione del Teorema 4.4.3. La dimostrazione procede in due fasi: si verifica l'inclusione per il singolo *stack* astratto, quindi la si estende all'intero insieme.

Caso del singolo *stack* astratto. Si vuole mostrare che per ogni $\hat{\sigma} \in \Sigma^h$:

$$[\![\mathsf{op}]\!]\, \overline{\gamma}(\hat{\sigma}) \;\subseteq\; \overline{\gamma}([\![\mathsf{op}]\!]^\sharp\, \hat{\sigma}).$$

Si escludono i casi banali: se $\hat{\sigma} = \bot_{\Sigma^h}$, allora $\overline{\gamma}(\bot_{\Sigma^h}) = \{\bot_{\mathbb{S}}\}$ e $[\![\mathsf{op}]\!]^\sharp \bot_{\Sigma^h} = \bot_{\Sigma^h}$, per cui l'inclusione vale. Se $\mathsf{height}(\hat{\sigma}) < \delta$, la semantica concreta restituisce $\bot_{\mathbb{S}}$ per tutti gli *stack* in $\overline{\gamma}(\hat{\sigma})$ per *stack underflow*, mentre la semantica astratta produce $\bot_{\Sigma^h}$; l'inclusione è quindi soddisfatta.

Nel caso generale si supponga $\delta = \rho < h$ per semplicità, notando che le altre combinazioni si trattano in modo analogo. Sia $\hat{\sigma} = [v_0, \ldots, v_{h-1-\delta}, \ldots, v_{h-1}]$. Per definizione di $\overline{\gamma}$, ogni *stack* concreto $s \in \overline{\gamma}(\hat{\sigma})$ ha la forma

$$s = [\ldots, z_0, \ldots, z_{h-1-\delta}, \ldots, z_{h-1}], \quad z_i \in \dot{\gamma}(v_i).$$

L'applicazione di $[\![\mathsf{op}]\!]$ rimuove i δ elementi in cima e inserisce ρ nuovi valori $\dot{z}_0, \ldots, \dot{z}_{\rho-1}$, ottenendo uno *stack* della forma $[\ldots, z_0, \ldots, z_{h-1-\delta}, \dot{z}_0, \ldots, \dot{z}_{\rho-1}]$. Per la definizione di $\overline{\gamma}$ applicata a $\mathsf{push}^\rho(\mathsf{pop}^\delta(\hat{\sigma}), \dot{v}_0, \ldots, \dot{v}_{\rho-1})$, tale *stack* concreto appartiene alla concretizzazione del risultato astratto, poiché le posizioni non modificate conservano i medesimi vincoli di $\dot{\gamma}$. Si conclude quindi:

$$[\![\mathsf{op}]\!]\, \overline{\gamma}(\hat{\sigma}) \;\subseteq\; \overline{\gamma}(\mathsf{push}^\rho(\mathsf{pop}^\delta(\hat{\sigma}), \dot{v}_0, \ldots, \dot{v}_{\rho-1})) = \overline{\gamma}([\![\mathsf{op}]\!]^\sharp\, \hat{\sigma}).$$

Estensione all'insieme. La correttezza per $\Sigma^{h,l}$ segue direttamente:

$$\begin{aligned}\llbracket \mathtt{op} \rrbracket\, \gamma(\widehat{S}) &= \bigcup_{\hat{\sigma} \in \widehat{S}} \llbracket \mathtt{op} \rrbracket\, \overline{\gamma}(\hat{\sigma}) \\ &\subseteq \bigcup_{\hat{\sigma} \in \widehat{S}} \overline{\gamma}(\llbracket \mathtt{op} \rrbracket^\sharp\, \hat{\sigma}) \\ &= \gamma(\llbracket \mathtt{op} \rrbracket^\sharp\, \widehat{S}),\end{aligned}$$

dove il secondo passo usa il risultato appena dimostrato e il terzo applica la definizione di γ. □

A.4 Struttura reticolare dei CFG e proprietà di jumpSolver

Dimostrazione del Lemma 4.4.4. $\mathbb{G}_{\mathtt{P}}$ è finito perché $E_\top$ è finito e i CFG per `P` corrispondono ai sottoinsiemi di $E_\top$ che contengono $E_\bot$. Un insieme finito con un ordine parziale è sempre un reticolo completo e soddisfa la condizione di catene ascendenti. □

Dimostrazione del Lemma 4.4.5. L'algoritmo JUMPSOLVER non rimuove mai archi dal grafo in ingresso, ma aggiunge soltanto nuovi archi verso *opcode* `JUMPDEST`. Pertanto $E \subseteq E'$, da cui $G \sqsubseteq_E G'$. □

Dimostrazione del Lemma 4.4.6. L'analisi astratta eseguita da RUNANALYSIS implementa un operatore monotono sul reticolo $\Sigma^{h,l}$: poiché il dominio soddisfa la condizione di catene ascendenti (Teorema 4.4.1), non è necessario alcun operatore di allargamento e il punto fisso è calcolato in modo esatto. Poiché $G_1 \sqsubseteq_E G_2$, ogni arco di G_1 è presente anche in G_2, quindi l'informazione disponibile per l'analisi su G_2 è almeno pari a quella su G_1. Ogni arco aggiunto a partire dall'analisi di G_1 sarà pertanto aggiunto anche a partire dall'analisi di G_2, da cui $E'_1 \subseteq E'_2$ e quindi $G'_1 \sqsubseteq_E G'_2$. □

A.5 Terminazione e correttezza di buildCFG

Dimostrazione del Teorema 4.4.7. Il reticolo $\mathbb{G}_{\mathtt{P}}$ è finito e soddisfa la condizione di catene ascendenti (Lemma 4.4.4). L'operatore JUMPSOLVER è estensivo (Lemma 4.4.5) e monotono (Lemma 4.4.6). Per il teorema di Knaster-Tarski, ogni operatore monotono su un reticolo completo ammette minimo punto fisso; la condizione di catene ascendenti garantisce che tale punto fisso è raggiunto in un numero finito di applicazioni dell'operatore a partire da $G_\bot$. Il

ciclo `do-while` di BUILDCFG applica iterativamente JUMPSOLVER e si arresta quando nessun arco viene aggiunto, ossia quando si raggiunge il punto fisso. □

Dimostrazione del Teorema 4.4.8. Per il Teorema 4.4.7, BUILDCFG termina producendo un CFG $G = (N, E)$ per P. Dai Teoremi 4.4.1, 4.4.2 e 4.4.3 segue che gli *stack* astratti calcolati da RUNANALYSIS sono una sovra-approssimazione di quelli che occorrono nelle esecuzioni concrete: formalmente, per ogni indirizzo ℓ raggiungibile, ogni *stack* concreto s tale che $\Xi^{\ell}(\mathtt{P}, \langle [], \ell_0 \rangle) = \langle s, \ell \rangle$ soddisfa $s \in \gamma(\widehat{S}_{in})$.

L'analisi della correttezza procede per casi sulla categoria del salto all'indirizzo ℓ:

Unreachable. Nessuna esecuzione concreta raggiunge ℓ, quindi l'insieme degli archi concreti è vuoto e l'inclusione è banalmente soddisfatta.

Erroneous. Ogni esecuzione concreta che raggiunge ℓ produce $\bot_{\mathbb{S}}$ per *stack underflow* prima o durante l'esecuzione del salto; pertanto non esiste alcun arco concreto della forma $\ell \to \ell'$ con $\ell' \neq \epsilon$, e l'inclusione è soddisfatta.

Resolved. Per ogni $\hat{\sigma} \in \widehat{S}_{in}$ non erroneo, $\mathsf{top}(\hat{\sigma}) = v \in \mathbb{Z}$. Se $v \in \mathbb{J}$, la destinazione concreta appartiene a $\dot{\gamma}(v) = \{v\}$ (quarta riga dell'Equazione (4.6)), quindi l'unica destinazione concreta raggiungibile è v stesso; JUMPSOLVER aggiunge l'arco $\ell \to v$, garantendo l'inclusione. Se invece $v = \top_{\overline{\mathbb{Z}}}$, allora $\dot{\gamma}(\top_{\overline{\mathbb{Z}}}) = \mathbb{Z} \setminus \mathbb{J}$, da cui nessuna destinazione concreta è un `JUMPDEST` valido e la semantica concreta restituisce $\bot_{\mathbb{S}}$; nessun arco è necessario.

Unknown. Poiché *conservative* = *true*, JUMPSOLVER aggiunge archi verso tutti gli elementi di $\mathbb{J}$ sia nel caso $\widehat{S}_{in} = \top_{\Sigma^{h,l}}$ sia nel caso $\mathsf{top}(\hat{\sigma}) = \top_{\mathbb{Z}^\sharp}$ per qualche $\hat{\sigma} \in \widehat{S}_{in}$. Poiché qualunque destinazione concreta di un salto deve essere un `JUMPDEST` (altrimenti la semantica concreta restituisce $\bot_{\mathbb{S}}$), l'insieme degli archi concreti è un sottoinsieme di $\{\ell \to \ell' \mid \ell' \in \mathbb{J}\} \subseteq E$, e l'inclusione è soddisfatta.

In tutti i casi l'inclusione richiesta dalla Definizione 3.3.2 è garantita, quindi G è un CFG *sound* per P. □

Bibliografia

[1] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[2] Vincenzo Arceri, Saverio Mattia Merenda, Greta Dolcetti, Luca Negrini, Luca Olivieri, and Enea Zaffanella. Towards a sound construction of EVM bytecode control-flow graphs. In Luca Di Stefano, editor, *Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2024, Vienna, Austria, 20 September 2024*, pages 11–16. ACM, 2024.

[3] Vincenzo Arceri, Saverio Mattia Merenda, Luca Negrini, Luca Olivieri, and Enea Zaffanella. EVMLiSA: Sound static control-flow graph construction for evm bytecode. *Blockchain: Research and Applications*, page 100384, 2025.

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[5] Vitalik Buterin. Ethereum white paper. *GitHub repository*, 1:22–23, 2013.

[6] Chainalysis. The ronin bridge hack investigation, 2022. `https://www.chainalysis.com/blog/axie-infinity-ronin-bridge-dprk-hack-seizure/` Accessed 03/2026.

[7] Gianluca Chiap. Blockchain: tecnologia e applicazioni per il business. 2019.

[8] Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.

[9] Patrick Cousot and Radhia 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*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery. https://doi.org/10.1145/512950.512973.

[10] Patrick Cousot and Radhia Cousot. Constructive versions of tarski's fixed point theorems. *Pacific journal of Mathematics*, 82(1):43–57, 1979.

[11] criptoinvestire.com. Crittografia: la sicurezza delle blockchain. 2018.

[12] DeFiLlama. DeFiLlama: Open-Source DeFi Data Aggregator. https://defillama.com, 2026. Accessed: 2026-03-04.

[13] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, 1976.

[14] Ghareeb Falazi, Uwe Breitenbücher, Frank Leymann, and Stefan Schulte. Cross-chain smart contract invocations: A systematic multi-vocal literature review. 56(6), January 2024.

[15] Gate Ventures. Review of Cross-Chain Bridge Hacks, 2022. https://medium.com/@gate_ventures/review-of-cross-chain-bridge-hacks-f70df59d5204 Accessed 04/2025.

[16] Google Cloud. Dissecting the nomad bridge hack, 2022. https://cloud.google.com/blog/topics/threat-intelligence/dissecting-nomad-bridge-hack Accessed 02/2026.

[17] Halborn. Explained: The Nomad hack (August 2022), 2022. https://www.halborn.com/blog/post/explained-the-nomad-hack-august-2022 Accessed 03/2026.

[18] Herbie et al. Smart contract security. 2023.

[19] Zeqin Liao, Yuhong Nan, Henglong Liang, Sicheng Hao, Juan Zhai, Jiajing Wu, and Zibin Zheng. SmartAxe: Detecting cross-chain vulnerabilities in bridge smart contracts via fine-grained static analysis. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[20] Charlie Miller et al. A deep dive into the parity multi-sig bug, 2017. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/ Accessed 02/2026.

[21] Antoine Miné et al. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends® in Programming Languages*, 4(3-4):120–372, 2017.

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

[23] Luca Negrini, Vincenzo Arceri, Luca Olivieri, Agostino Cortesi, and Pietro Ferrara. Teaching through practice: Advanced static analysis with lisa. In Emil Sekerinski and Leila Ribeiro, editors, *Formal Methods Teaching*, pages 43–57, Cham, 2024. Springer Nature Switzerland. `https://doi.org/10.1007/978-3-031-71379-8_3`.

[24] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. Lisa: a generic framework for multilanguage static analysis. In *Challenges of Software Verification*, pages 19–42. Springer, 2023.

[25] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. LiSA: A generic framework for multilanguage static analysis. In Vincenzo Arceri, Agostino Cortesi, Pietro Ferrara, and Martina Olliaro, editors, *Challenges of Software Verification*, pages 19–42. Springer Nature Singapore, Singapore, 2023. `https://doi.org/10.1007/978-981-19-9601-6_2`.

[26] Nico et al. Ethereum accounts. 2023.

[27] Nico et al. Ethereum smart contracts. 2023.

[28] Nico et al. Ethereum virtual machine. 2023.

[29] Luca Olivieri, Thomas Jensen, Luca Negrini, and Fausto Spoto. Michelsonlisa: A static analyzer for tezos. In *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 80–85, 2023. `https://doi.org/10.1109/PerComWorkshops56833.2023.10150247`.

[30] Luca Olivieri, Aradhita Mukherjee, Nabendu Chaki, and Agostino Cortesi. Blockchain interoperability through bridges: A token transfer perspective. In *2024 6th International Conference on Blockchain Computing and Applications (BCCA)*, pages 742–748, 2024.

[31] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, and Fausto Spoto. Information Flow Analysis for Detecting Non-Determinism in Blockchain. In Karim Ali

and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–25, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2023.23.

[32] Konashevych Olwksii. Takeaways: 5 years after the dao crisis and ethereum hard fork. 2021.

[33] OWASP Foundation. OWASP Smart Contract Top 10, 2024.

[34] Parity Technologies. The multi-sig hack: A postmortem, 2017. https://web.archive.org/web/20210227150251/https://www.parity.io/the-multi-sig-hack-a-postmortem/ Accessed 02/2026.

[35] Michele Pasqua, Andrea Benini, Filippo Contro, Marco Crosara, Mila Dalla Preda, and Mariano Ceccato. Enhancing ethereum smart-contracts static analysis by computing a precise control-flow graph of ethereum bytecode. *J. Syst. Softw.*, 200:111653, 2023. https://doi.org/10.1016/J.JSS.2023.111653.

[36] Permisison.io. Permissioned vs. permissionless blockchains explained. 2021.

[37] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[38] Toshendra Kumar Sharma. Types of blockchains explained - public vs. private vs. consortium. 2023.

[39] David Siegel. Understanding the dao attack. 2023.

[40] Sky Mavis. The ronin security breach, 2022. https://roninchain.com/blog/5-the-ronin-security-breach Accessed 03/2026.

[41] Nick Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)*, 18(2):28, 1996.

[42] Wikipedia. Crittografia asimmetrica. 2024.

[43] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014. https://cryptodeep.ru/doc/paper.pdf, Accessed: 12-02-2024.

Ringraziamenti

Il primo ringraziamento va al mio professore Vincenzo Arceri, che mi ha accompagnato in questo lungo cammino durato diversi anni. Mi ha dato fiducia quando ancora non sapevo bene dove stessi andando e mi ha fatto scoprire un modo di lavorare e di pensare che porto con me ben oltre questa tesi. Senza di lui non sarei arrivato fino a qui e non avrei vissuto le esperienze che mi hanno formato, dentro e fuori dall'università.

Ringrazio i miei colleghi, per gli infiniti aperitivi al Bar Navetta e per tutti i momenti di divertimento e di risate che hanno reso questo percorso molto più bello. A loro devo anche la scoperta di angoli sperduti della Svezia che da solo non avrei mai raggiunto: tre ore di volo da Milano per finire in posti che non sapevo nemmeno esistessero, e che oggi rifarei senza pensarci.

Ringrazio infine chi c'è sempre stato: la mia famiglia, i miei amici e la mia ragazza. A loro va il grazie più grande.