Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Analisi Statica di Smart Contract in Ethereum: Stimare i Consumi di Gas

Relatore:
Chiar.mo Prof.
Ugo Dal Lago

Presentata da:
Melania Ghelli

Sessione II
Anno Accademico 2018-2019

“ Perché nessuno possa dimenticare di quanto sarebbe bello se,
per ogni mare che ci aspetta, ci fosse un fiume, per noi.
E qualcuno - un padre, un amore, qualcuno - capace
di prenderci per mano e di trovare quel fiume . . . ”
- Alessandro Baricco

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Le innovazioni tecnologiche introdotte negli ultimi decenni hanno rivoluzionato la nostra società. Il risultato che ne deriva è che numerosi settori stanno cambiando, muovendosi verso una realtà sempre più digitale. Se da una parte questo ha costituito un progresso, dall'altra ha creato nuove opportunità per i cybercriminali.

Oggi il principale obiettivo della sicurezza informatica è proprio quello di trovare soluzioni adattabili alle nuove infrastrutture, come i sistemi IoT [21] che si stanno diffondendo in modo capillare. Con l'impiego di queste nuove tecnologie nell'industria i punti di accesso alla rete aziendale sono aumentati, moltiplicando la tipologia ed il numero di minacce. In questo contesto il rischio che si corre è maggiore, poichè legato alla violazione di dati sensibili o addirittura alla compromissione dei processi di produzione.

La blockchain nasce nel 2008 assieme al Bitcoin [26], e ad oggi riveste un ruolo chiave nel settore della cybersecurity, offrendo sicurezza, anonimato e integrità dei dati senza la necessità di un ente centrale che faccia da garante [8].

Dal punto di vista informatico il termine Blockchain identifica un registro distibuito, organizzato in blocchi legati tra loro. Ciascun blocco raggruppa un numero arbitrario di *transazioni*, vale a dire l'unità di base in cui le informazioni vengono codificate nel registro. Il contenuto di ciascuna transazione varia a seconda del protocollo adottato dalla blockchain. Ad interagire con la blockchain sono i miner, coloro che effettivamente realizzano le transazioni per conto degli utenti della rete. Si occupano di raggruppare le transazioni in blocchi, per poi farli approvare dal resto della rete. Per ottenere il consenso delle altre parti è richiesta la risoluzione di un algoritmo chiamato *proof-of-work*.

Il paradigma trova applicazioni nei settori piú disparati [19], offrendo innovazione

grazie alla possibilitá di fare a meno di banche o istituzioni pubbliche. Può essere utilizzata come tecnologia di base dalla quale partire per implementare dei servizi efficienti.

Tuttavia ad oggi le applicazioni più diffuse della blockchain includono l'impiego delle crittovalute. La più celebre è sicuramente il Bitcoin, il primo esempio storico di moneta digitale che introduce i concetti di rete distibuita e algoritmo proof-of-work. Questo protocollo permette di raggiungere un consenso distribuito attraverso tutto il network. È il concetto chiave per capire come riusciamo ad evitare di affidarci ad un'autorità centrale.

Tra le principali crittovalute troviamo Ethereum, che con una capitalizzazione di circa 16 miliardi di dollari ed una quotazione attuale di circa 140€ è seconda al Bitcoin. Ethereum si differenzia dalla sua concorrente per i servizi offerti: si tratta di una piattaforma open-source che mette a disposizione un linguaggio di programmazione Turing-completo. Questo linguaggio puó essere utilizzato dagli utenti per implementare dei programmi, i così detti smart contract.

Gli smart contract possono essere eseguiti sul network peer-to-peer di Ethereum solo al fronte di un pagamento anticipato. Per ragioni di sicurezza a ciascuna istruzione di basso livello è associato un costo monetario. Dunque eseguire un programma costerá tanto quante sono le istruzioni che lo compongono. Il costo di ciascuna istruzione è espresso in termini di gas, una sorta di carburante che viene pagato in ether, la crittovaluta di Ethereum. Dal momento che il gas viene pagato in anticipo, potrebbe accadere che l'esecuzione di un programma ecceda la quantitá messa a disposizione. In questi casi la computazione non giunge al termine, risultando nella perdita delle risorse investite dall'utente. Oltre a questo comportamento indesiderato l'esaurimento del gas disponibile puó avere conseguenze pericolose. Un programma che non gestisce correttamente queste situazioni viene etichettato come vulnerabile. La conseguenza piú diretta è il blocco del contratto, che puó essere anche permanente. La pericolositá però risiede nel fatto che questo tipo di programmi diventano un bersaglio facile per attacchi malevoli. Dato il valore monetario associato agli smart contract il rischio che si corre in caso di attacchi informatici è una ingente perdita di denaro. Un caso del genere si è verificato con l'attacco DAO [9] che nel 2016 ha comportato la perdita di circa 150 milioni di

dollari.

Per queste ragioni è necessario individuare possibili criticità nel codice prima della sua esecuzione. In questo contesto l'analisi statica dei programmi costituisce un potente strumento di prevenzione.

All'interno di questo elaborato ci concentreremo solo sulle tecniche di analisi dei consumi di gas. Conoscere a priori quest'informazione permetterebbe non solo un investimento adeguato da parte degli utenti, ma anche uno strumento di difesa da eventuali attacchi. Nonostante i numerosi vantaggi che potrebbe apportare, l'utilizzo dell'analisi statica nello sviluppo degli smart contract è ancora una pratica poco diffusa. Come vedremo, una delle motivazioni più forti è l'assenza di strumenti che permettano di calcolare con precisione la quantitá di gas richiesto durante una computazione. L'obiettivo principale di questa tesi è quello di analizzare dei tool capaci di condurre questo tipo di analisi, e di verificarne la precisione.

Capitolo 1

Background

Lo scopo di questo capitolo è quello di fornire una panoramica dei concetti chiave intorno ai quali si sviluppa questo elaborato di tesi.

1.1 Blockchain

Il termine Blockchain - in italiano "catena di blocchi" - identifica un registro distribuito e sicuro. In questo senso si può pensare alla blockchain come ad una struttura di dati in continua evoluzione, dove le informazioni sono raggruppate in blocchi collegati fra loro.

Ciascun blocco codifica una sequenza di transazioni individuale, e viene concatenato a quello precedente in ordine cronologico, usando una *funzione crittografica di hash*, vale a dire un algoritmo che mappa dati di dimensione arbitraria in una stringa di dimensione finita. Per definizione le funzioni hash sono unidirezionali: dall'output è molto difficile risalire all'input. Per essere definite tali, queste funzioni devono rispettare delle specifiche proprietà di sicurezza. La concatenazione tra i blocchi è dunque irreversibile: ciascun nuovo blocco contiene l'hash del suo predecessore. In questo modo, modificare un blocco implicherebbe l'invalidazione di *tutta* la catena successiva.

La peculiarità di questa struttura risiede nel fatto che sia condivisa: ogni nodo che compone la rete mantiene una copia del registro aggiornata. Per poter aggiungere un blocco è necessario validare l'intera catena, ed ottenere un consenso da parte degli altri

nodi della rete. Una volta ottenuto, il nuovo blocco viene trasmesso agli altri componenti in modo tale da aggiornare lo stato della blockchain.

Il processo di validazione dei nuovi blocchi viene chiamato *mining*, e viene realizzato da membri del network la cui mansione è limitata a questo. Tali personaggi sono chiamati *miner*, ed il loro compito è quello di verificare le transazioni proposte per poi fare in modo che il nuovo blocco venga linkato alla blockchain. Per fare questo i miner sono chiamati a risolvere un algoritmo proof-of-work, un puzzle crittografico che richiede un significativo costo computazionale per essere risolto.

Questo sistema permette di raggiungere il consenso senza la necessità di un'autorità centrale che faccia da garante. È il concetto chiave delle tecnologie basate su blockchain: la possibilità di implementare servizi sicuri senza appoggiarsi a banche, istituzioni pubbliche, ecc.

Questa nuova tecnologia può essere integrata in diverse aree [19], trovando applicazioni che spaziano dai sistemi di voto, alla difesa della proprietà intellettuale, o addirittura ai sistemi di crowfunding. Ad oggi tuttavia l'uso della blockchain più conosciuto è quello nei sistemi di pagamento che impiegano crittovalute; il dato non è poi così sorprendente: la prima blockchain nasce grazie a Satoshi Nakamoto assieme al Bitcoin [26]. In questo senso il Bitcoin è una piattaforma di pagamenti, dove la catena di blocchi funge da storico di tutte le transazioni avvenute: una sorta di conto corrente condiviso.

1.2 Ethereum, gli Smart Contract e la EVM

All'interno di quest'elaborato verrà preso in considerazione solo il network Ethereum, una piattaforma decentralizzata basata su una blockchain, che come Bitcoin possiede una propria valuta: l'*ether*. Conosciuta anche con la sigla ETH, l'ether è suddivisa in unità di conto, tra le quali la più piccola e più conosciuta è il Wei; 1 Wei equivale a 10^{-18} Ether. Le altre sottounità sono mostrate nella Tabella 1.1.

Diversamente da quanto vale per le altre crittovalute, Ethereum non è solo un network per lo scambio di moneta, ma un framework che permette l'esecuzione di programmi. Tali programmi prendono il nome di *smart contract*, cioè "contratti intelligenti". Sebbe-

Ordine	Nome
10^0	Wei
10^{12}	Szabo
10^{15}	Finney
10^{18}	Ether

Tabella 1.1: Unità di conto di Ethereum

ne il nome possa suggerire una funzione ben precisa, questi programmi sono usati per computazioni general-purpose, permettendo quindi di realizzare un vasto numero di operazioni. All'interno di questo elaborato utilizzeremo impropriamente anche il termine *contratto*.

Gli smart contract sono scritti in linguaggi ad alto livello; fra i vari (Serpent, Viper e LLL) quello più diffuso ad oggi è Solidity [13]. Tale linguaggio object-oriented è pensato solo per lo sviluppo di smart contract che, per poter girare nella rete, vengono poi tradotti in bytecode. Ciascun nodo di Ethereum infatti esegue localmente la Ethereum Virtual Machine, anche detta EVM, una macchina a stack in grado di eseguire un linguaggio di basso livello, ossia bytecode. Questo linguaggio è non tipato, e composto da un piccolo insieme di istruzioni.

1.3 Il Ruolo del gas

Per *gas* si intende l'unità di misura dello sforzo computazionale richiesto dalla EVM per eseguire ciascuna istruzione. Ogni computazione eseguita dal network è soggetta ad una tassazione, il cui importo è espresso in unità di gas. In questo senso il gas può essere inteso come un carburante, che serve alla EVM per poter portare avanti ciascuna transazione; è necessario acquistarne una quantità adeguata *prima* che l'operazione venga eseguita, e qualora questa finisca la macchina si fermerà.

Il gas viene pagato in ether dagli utenti che intendono far eseguire una *transazione*, i quali possono essere anche esterni alla rete Ethereum. Una transazione è una semplice

istruzione che viene propagata su tutto il network, processata dai miner e, se validata da questi, aggiunta alla blockchain. Esistono due tipologie di transazioni: le *message call*, che prevedono il passaggio di un messaggio tra mittente e destinatario, e le *contract creation*, vale a dire transazioni che danno origine alla creazione di un nuovo contratto. Per ciascuna transazione il committente è tenuto a specificare alcuni parametri tra cui:

gasPrice: corrisponde al prezzo di acquisto di ciascuna unità di gas associata alla transazione; è espresso in Wei.

gasLimit: è la massima quantità di gas che dovrebbe essere utilizzata per portare a termine la transazione.

Il prezzo del gas, in modo analogo a quello dei carburanti, non ha un costo fisso. Invece che dipendere dalle oscillazioni di mercato, viene deciso direttamente dall'utente che lo acquista. Potrebbe sembrare una scelta poco intelligente, ma in realtà questo fattore incide molto sul destino delle transazioni: i miner sono liberi di scegliere se processare una transazione o meno sulla base del prezzo che i committenti sono disposti a pagare. Questo perché l'ether destinato all'acquisto di gas costituisce una delle loro fonti di guadagno. Al termine della transazione infatti l'ether destinato all'acquisto del gas che non viene né utilizzato dalla computazione, né rimborsato all'utente, può essere accreditato su di un indirizzo specifico chiamato *beneficiario*. Tale indirizzo viene specificato dal miner, e generalmente coincide con un conto sotto controllo diretto dello stesso miner. Quando coloro che propongono una transazione devono scegliere quanto pagare per il gas, devono trovare un compromesso tra la possibilità di risparmiare e quella di aumentare la probabilità che la loro transazione venga scelta in poco tempo.

La scelta di restituire il gas inutilizzato al termine delle computazioni dipende dal fatto che questo bene non abbia alcun valore al di fuori dall'esecuzione delle transazioni. Per come è stato definito, non si può *possedere* gas.

Ciascuna istruzione di basso livello ha associato un costo fisso in gas. Per calcolare quindi il consumo totale di un programma Solidity è necessario comprendere in quali istruzioni di basso livello verrà tradotto. I costi di alcune delle istruzioni EVM [28] sono riportati nella Tabella 1.2.

Istruzione	Costo	Descrizione
JUMPDEST	1	Indica la destinazione di un'istruzione JUMP
`POP`	2	Rimuove un elemento dallo stack
`PUSHn`	3	Inserisce un elemento di n byte nello stack
`ADD, SUB`	3	Operatori aritmetici
`AND, OR, NOT, XOR, ISZERO, BYTE`	3	Operatori logici
`MUL, DIV`	5	Operatori aritmetici
`JUMP`	8	Salto semplice senza condizione
`JUMP1`	10	Salto condizionale
`CALL`	700	Chiama una transazione
`CALLVALUE`	9000	Pagato per un argomento diverso da 0 dell'istruzione CALL
`SSTORE`	20000	Salva una parola in memoria. Si paga quando il valore precedente è uguale a 0

Tabella 1.2: Costi espressi in gas di alcune istruzioni della EVM

Il concetto del gas è stato introdotto in Ethereum con l'obiettivo di difendere la rete dai possibili abusi. La ragione per cui il protocollo prevede di far pagare i propri utenti non è triviale. Prima di tutto impedisce loro di sovraccaricare i miner di lavoro sfruttando il potere computazionale della rete. Inoltre scoraggia gli utenti a impiegare troppa memoria, una risorsa preziosa nelle tecnologie basate su blockchain. Infine limita il numero di computazioni eseguite dalla stessa transazione. Questo difende il network intero da attacchi malevoli come i DDoS: la finitezza delle transazioni fa sì che non si possa, ad esempio, far ciclare un programma infinitamente. Per poter disabilitare la rete anche solo per pochi minuti gli hacker dovrebbero pagare delle ingenti somme.

Capitolo 2

Analisi Statica

L'analisi statica è un processo di valutazione della correttezza dei programmi che rientra tra le tecniche di verifica del software. L'aggettivo *statica* identifica una serie di controlli che possono essere effettuati sul codice prima della sua esecuzione. In questo si differenzia dall'analisi dinamica, una tecnica complementare che comprende quei controlli che vengono invece effettuati a runtime.

L'analisi statica solitamente è il primo controllo che viene effettuato sul codice durante lo sviluppo del programma. I vantaggi apportati da questo tipo di controllo del codice sono numerosi. I bug possono essere individuati per tempo, evitando comportamenti inaspettati da parte dei programmi. Inoltre, a seconda del tool utilizzato, è possibile migliorare il codice dal punto di vista della leggibilità, della strutturazione o della performance.

Tuttavia l'analisi statica resta una pratica poco diffusa tra gli sviluppatori [20]. Le principali cause sono da ricondursi agli output prodotti dai tool di analisi: l'elevato numero di warning così come la presenza di falsi positivi rendono gli strumenti meno affidabili. Oltre a questo, gran parte degli informatici afferma di non utilizzare questi tool di verifica a causa del sovraccarico di lavoro: spesso i ritmi aziendali così come la scarsa collaborazione dei team di sviluppo fanno sì che non ci siano i presupposti per dedicare spazio sufficiente all'analisi del codice.

2.1 Analisi Statica vs. Analisi Dinamica

Per analisi statica si intende l'analisi dei programmi dal punto di vista del codice che li compone, vale a dire senza doverli eseguire. L'analisi può essere fatta sia sul codice sorgente, che sul codice oggetto, ossia il prodotto della compilazione.

L'analisi statica viene condotta su tre dimensioni: esaminando la struttura del programma, costruendo un modello che rappresenti i possibili stati del codice e ragionando sul possibile comportamento in fase di esecuzione [11]. Rientrano in questa categoria la verifica formale dei programmi e le ottimizzazioni a tempo di compilazione. L'analisi statica viene spesso implementata da tool automatici, e garantisce proprietà di correttezza. Le principali critiche mosse nei confronti di questa tecnica derivano dal fatto che possa portare a dei *falsi positivi*, cioè situazioni in cui viene segnalata una vulnerabilità nel codice sebbene non sia stata violata alcuna regola. La ragione di questo comportamento è legata alle proprietà che ci poniamo di verificare: la terminazione di un programma, ad esempio, è di per sé una proprietà indecidibile. È dimostrato matematicamente che non esiste alcun metodo di analisi statica che sia al tempo stesso *corretto* e *completo* e che non sia limitato dalle risorse che utilizza (es. memoria, tempo)[1] [5]. Dunque, per ottenere degli strumenti realmente utili, siamo costretti a rinunciare alla completezza dell'analisi, preferendo dei risultati che siano corretti. Ci accontentiamo di strumenti che producono risultati approssimati, talvolta anche diversi da quelli attesi.

L'analisi dinamica identifica quei controlli che possono essere effettuati sul programma soltanto durante la sua esecuzione, che sia su un processore reale o virtuale. Il software testing rientra in questa categoria. Per condurre questo tipo di controllo è necessario fornire un input ben preciso e analizzare poi il comportamento del programma. Occorre inoltre stabilire a priori *che cosa* si vuole misurare. Sebbene questa analisi sia più veloce rispetto alla prima, non garantisce la stessa correttezza. Per essere rigorosa infatti l'analisi dinamica dovrebbe coprire ogni possibile configurazione del programma.

[1] Questo risultato è dato dal Teorema di Rice

2.2 Tecniche di Analisi Statica in Informatica

Le tecniche di analisi possono essere suddivise in due tipologie in base ai risultati prodotti. La prima categoria comprende i tool di analisi volti a localizzare bug nel codice. La seconda identifica invece un gruppo di software con una forte base logica, che utilizzano tecniche matematiche per la verifica di specifiche prorietà del programma.

Di seguito daremo una panoramica sulle principali tecniche [27] di analisi statica.

2.2.1 La Compilazione

Tutti i compilatori per eseguire la traduzione del codice dorgente in codice oggetto applicano l'analisi statica. L'operazione di compilazione, intesa come *traduzione automatica*, può essere suddivisa in due macro-fasi: dal codice sorgente alla generazione della forma intermedia, e dalla forma intermedia al codice oggetto, cioè il prodotto finale. La prima fase è quella che fa più utilizzo di analisi statica; il compilatore esegue in sequenza delle trasformazioni sul codice, chiamate analisi lessicale, sintattica e semantica. È durante quest'ultimo passaggio che il programma viene sottoposto ai controlli relativi ai vincoli del linguaggio [14]: si controllano le dichiarazioni delle variabili, la coerenza dei tipi, il numero dei parametri delle funzioni ecc. In generale questi controlli variano a seconda del linguaggio di programmazione.

La compilazione dunque non costituisce una vera e propria tecnica, ma piuttosto un tipo di controllo sul codice che non può essere risparmiato.

2.2.2 Tecniche Manuali

Consideriamo *manuali* quelle tecniche che non possono essere automatizzate da un software, ma richiedono l'interazione umana per poter essere realizzate. Di seguito ne citiamo alcune.

Code Reading

Come suggerisce il termine stesso si tratta della rilettura del codice da parte di una persona. Sebbene i bug identificabili possono variare in base a diversi fattori (es. numero di persone, conoscenza del codice, livello di esperienza) questa operazione può portare alla luce difetti che invece il compilatore non rileva. Commenti inconsistenti con il codice, nomi di variabili errati, loop infiniti, codice non strutturato, sono solo alcuni di questi. L'efficacia di questa tecnica è limitata se colui che legge il codice è la stessa persona ad averlo sviluppato.

Code Reviews

Generalmente adottata in contesti aziendali. Identifica un controllo del codice fatto in gruppo, il quale viene costituito secondo requisiti specifici. È una riunione dove lo sviluppatore è chiamato a leggere il codice ad alta voce di fronte ad altri esperti, che possono commentare il programma con lo scopo di individuare gli errori; in questo modo possono essere rilevati dal 30 al 70% di quelli presenti nel programma.

Walktrough

Molto simile alla tecnica precedente per le modalità in cui viene effettuata, poiché prevede la riunione di un gruppo di persone. Differisce negli obiettivi: cerca di trovare dei difetti nel comportamento del programma, e per farlo simula l'esecuzione del codice a mano.

2.2.3 Tecniche Automatizzabili

Queste tecniche di revisione del codice possono essere automatizzate, al fine di implementare dei tool di analisi. Ne riportiamo alcuni esempi.

Control Flow Analysis

Prevede la rappresentazione del codice attraverso un grafo chiamato CFG (*Control Flow Graph*), dove ciascun nodo rappresenta un'istruzione o un predicato, mentre gli archi il passaggio del flusso di controllo. Successivamente il grafo viene analizzato, al fine di rilevare anomalie nel programma quali non strutturazione o iraggiungibilità del codice.

Data Flow Analysis

La tecnica di analisi del data flow solitamente rientra nella categoria dei controlli dinamici. Analizza l'evoluzione delle variabili durante il tempo di esecuzione, al fine di rilevare anomalie. Parte di questi controlli possono essere effettuati anche staticamente, permettendo di rilevare parte dei comportamenti anomali del programma, come l'uso delle variabili prima della loro dichiarazione, o l'annullamento prima dell'utilizzo.

Esecuzione Simbolica

Consiste nell'esecuzione del programma con dei valori di input simbolici (es. espressioni) piuttosto che con i valori effettivi. Può risultare molto difficile da realizzare in caso di istruzioni if, poichè rende complesso valutare la condizione. Un altro caso che viene mal gestito è quello dei cicli, sia determinati (nel caso dipendano dal valore di una variabile) che non.

2.3 Analisi Statica di Smart Contract

L'impiego delle tecniche di analisi statica per la verifica degli smart contract non è molto diffuso. Principalmente perchè data la dimensione limitata di questi programmi non si ritiene necessario il suo impiego.

In parte l'impopolarità dell'analisi statica è dovuta anche alla difficile rappresentazione del bytecode EVM. Decompilare le istruzioni di basso livello al fine di ottenere una

rappresentazione migliore che funga da base per una buona analisi richiede un notevole sforzo. Un altro fattore a rendere poco appetibile l'applicazione di queste tecniche al mondo degli smart contract è il rischio di ottenere falsi positivi.

2.4 Tool per l'Analisi

Durante questo lavoro è stato preso in considerazione un certo numero di software che implementano tecniche di analisi statica orientata alla verifica degli smart contract. Di seguito ne vedremo alcuni.

2.4.1 Verificare le Proprietà di Sicurezza

I seguenti software sono stati pensati per verificare la sicurezza dei programmi di Ethereum.

Il primo è uno strumento completo, per cui si può etichettare uno smart contract come *sicuro* o meno. Il secondo invece è in grado di indivuare dei comportamenti anomali dei programmi causati soltanto dall'esaurimento del gas. Dunque la sua verifica comprende una tipologia circoscritta di proprietà di sicurezza.

EtherTrust [16] questo framework offre la possibilità di analizzare i programmi al fine di verificarne le proprietà di sicurezza. Tali proprietà, come ad es. la *single-entrancy*, per poter essere verificate devono prima essere modellate.
Per condurre la sua analisi EtherTrust produce una rappresentazione astratta del bytecode EVM nella forma di clausole di Horn. Successivamente questa rappresentazione viene data in input ad un SMT solver, il quale verifica che siano rispettate delle proprietà di sicurezza ben precise. EtherTrust garantisce la proprietà di correttezza.

MadMax [15] attraverso la combinazione di più tecniche di analisi statica (analisi Data Flow e Control Flow) questo software è in grado di verificare smart contract

al fine di scoprire bug legati all'esaurimento del gas disponibile.
MadMax individua una serie di vulnerabilità *gas-focused* in modo da definire dei pattern da ricercare attraverso l'analisi dei programmi. Questa viene condotta a partire da una rappresentazione intermedia (IR) del codice, ottenuta tramite la decompilazione del bytecode EVM.

2.4.2 Rappresentare il Bytecode EVM

I prossimi tool utilizzano tecniche di analisi statica per fornire una miglior rappresentazione del bytecode. I risultati che si ottengono dalla loro esecuzione possono essere utilizzati per un'analisi statica volta alla verifica delle proprietà del codice.

KEVM [17] produce una semantica formale per la EVM. Gli autori del programma sottolineano che la loro rappresentazione del byteocode si presta facilmente all'applicazione di tecniche di analisi, e forniscono come esempio un tool per stimare i consumi di gas degli smart contract.

EthIR [3] è un framework di analisi del bytecode di EVM. A partire dalle istruzioni di basso livello, che vengono rappresentate tramite grafi CFG dal tool Oyente [25], EthIR produce una rappresentazione *Ruled Based* (RB). Tale modellizzazione può essere utilizzata per desumere proprietà del bytecode, applicando delle ulteriori tecniche di analisi statica.

2.4.3 Fornire un Bound al gas

L'ultima categoria di software che vedremo è la più interessante dal punto di vista della ricerca che abbiamo condotto. Si tratta di programmi che tramite la combinazione di tecniche di analisi statica rilevano e forniscono un bound ai consumi di gas dei programmi esaminati.

Li citeremo per completezza, per poi trattarli in modo più dettagliato nei capitoli successivi.

solc [12] è il compilatore ufficiale di Solidity. Tra le opzioni di utilizzo c'è la modalità *gas*, dove l'output prodotto è una stima della quantità di gas richiesto dal programma. Nella maggior parte dei casi il risultato prodotto è infinito.

GASTAP [4] è la prima piattaforma sviluppata in grado di analizzare smart contract al fine di dare un upper bound ai consumi di gas dello stesso. Questo software è ancora in via di sviluppo, perciò presenta ancora delle limitazioni. Tuttavia si distingue per la precisione nella stima dei bound, riuscendo a fornire un analisi più precisa rispetto ad altri programmi che implementano le stesse funzionalità.

Capitolo 3

Risultati Sperimentali

Lo scopo di questo capitolo è quello di riassumere gli esperimenti condotti e illustrare i relativi risultati.

Una volta individuati i tool capaci di determinare bound espliciti ai consumi di gas si è testato il loro comportamento su input diversi. Per studiare la potenza espressiva dei software, i programmi in input sono stati scelti in modo da testare diversi costrutti base messi a disposizione dal linguaggio Solidity.

I test sono stati condotti con il tool GASTAP e con il compilatore solc, ad oggi gli unici capaci di produrre dei bound ai consumi di gas per gli smart contract di Ethereum. Per quanto riguarda il primo gli sviluppatori hanno fornito i dettagli della sua implementazione, documentando in che modo venga condotta la loro analisi. Conoscere le tecniche utilizzate ha permesso di dedurre alcune proprietà del tool e dell'analisi di smart contract in generale. Dall'altro lato la documentazione del compilatore solc [12] tratta l'analisi del gas in modo superficiale. Tuttavia il suo impiego nei test condotti ci ha permesso di confrontare i risultati di GASTAP, in modo da considerare l'analisi degli smart contract in chiave critica.

3.1 Stimare i Consumi di gas

Perché stimare i consumi di gas è importante?

I consumi di gas sono associati alle operazioni di basso livello, dunque specificati solo per il bytecode EVM [28]. Dal momento che però gli smart contract sono sviluppati utilizzando linguaggi di alto livello (come ad es. Solidity [13]), è difficile per lo sviluppatore conoscere i costi del proprio programma durante la fase di sviluppo. Per di più la traduzione dei costrutti di alto livello in bytecode fa sì che stimare i constumi tramite l'analisi statica sia una sfida non triviale. L'impiego dell'analisi statica si rende indispensabile, in quanto il costo totale in gas richiesto per eseguire un programma dipende anche da altri fattori.

Semplificando potremmo dire che i costi totali dipendono da:

1. il costo intrinseco di ciascuna istruzione di basso livello; questi valori sono fissati (vedi Tabella 1.2).

2. i costi determinati dalla creazione di un contratto o dalla chiamata di un altro programma. Questi sono determinati dalle istruzioni `CREATE` , `CALL` and `CALLCODE`.

3. eventuali costi aggiunti, che vengono addebitati nel caso in cui la memoria richiesta dal programma superi una certa soglia.

Mentre alcuni di questi valori possono essere facilmente stimati, altri possono essere determinati soltanto durante l'esecuzione del contratto. Un modo per sopperire a questa difficoltà è l'analisi statica: soltanto delle tecniche precise ci permettono di stimare questi valori, poichè consentono di calcolare in anticipo quali "sorprese" riserverà il codice durante la sua esecuzione.

Dal momento che il gas viene pagato anticipatamente, può succedere che durante la sua esecuzione un programma ecceda la quantità che ha a disposizione. Come conseguenza, la EVM sollleva un'eccezione di tipo *out-of-gas* e abortisce la transazione. Un contratto che non gestisce bene l'eventuale interruzione di una transazione è soggetto ad una vulnerabilità legata al gas. Una panoramica su questo tipo di vulnerabilità viene fornita dagli autori di MadMax [15]. Generalmente questi programmi, che vengono

etichettati come rischiosi, saranno bloccati in modo permanente. Quando si eccede il gas disponibile un'altra conseguenza più immediata è il blocco della transazione: la computazione non giunge a termine, l'utente non ottiene il risultato desiderato e l'ether pagato per il gas va perso.

Un altro limite all'esecuzione dei contratti è dovuto al protocollo adottato da Ethereum. Questo infatti pone un limite superiore alla quantità di gas che ciascun blocco può consumare. Dal momento che le transazioni vengono raggruppate in blocchi, tale limite influenza anche queste: il costo di ciascuna transazione non può superare il limite superiore del blocco al quale appartiene. Può succedere infatti che se l'esecuzione di una certa funzione aumenta nel tempo, ad un certo punto non sia più possibile portarla avanti a causa del superamento del limite massimo di gas. Conoscere a priori i consumi può aiutare anche ad evitare questo tipo di errori.

Va inoltre considerato che se l'utente della rete ha modo di conoscere il costo di una computazione nel suo caso pessimo, ha anche modo di confrontare fra di loro dei programmi semanticamente equivalenti, al fine di prediligere quello che consuma meno. In questo senso la stima dei costi di gas costituirebbe l'uinica fonte di risparmio: considerato che le transazioni sotto una certa soglia di *gasPrice* rischiano di non essere accettate dai miner, i committenti non hanno margine di risparmio.

Una stima affidabile del gas aiuta un utente a stabilire un prezzo per ciascuna unità di gas in linea con l'utilità della sua transazione. Infatti una quantità insufficiente per completare la transazione comporta la perdita dei soldi investiti, senza che la transazione venga eseguita. Al tempo stesso una sovrastima fa sì che i miner assumano un atteggiamento diffidente, abbassando la probabilità che la stessa venga scelta.

Conoscere un limite ai consumi di gas del proprio programma assicura all'utente che se la quantità di gas investito supera il bound l'esecuzione verrà portata a termine enza incorrere in spiacevoli sorprese.

3.2 Caratteristiche del Tool GASTAP

GASTAP (acronimo per Gas-Aware Smart contracT Analysis Platform [4]), è un tool automatico di analisi statica per i programmi di Ethereum. La principale tecnica di analisi adottata da questo software è la Control Flow Analysis (vedi Sez. 2.2.5).

Dato in input uno smart contract scritto in Solidity, EVM bytecode oppure EVM disassemblato, GASTAP produce un upper bound in termini di gas per ciscuna delle funzioni pubbliche che lo compongono. Per produrre questo calcolo il tool effettua una serie di operazioni in sequenza: (1) costruzione dei grafi *control-flow* (CFG), (2) decompilazione del codice di basso livello in una rappresentazione di alto livello, (3) deduzione delle relazioni di grandezza, (4) generazione delle equazioni di gas, e (5) risoluzione delle equzioni fino a formare un bound.

GASTAP ha un ampio spettro di applicazioni, sia per chi sviluppa o possiede i contratti, sia per gli attaccanti, permettendo di individuare vulnerabilità nel codice e di verificare gli utilizzi di gas, eventualmente anche a scopo di debugging. Dal punto di vista degli sviluppatori e dei proprietari un buon tool di analisi serve a conoscere la quantità di gas necessaria per eseguire in modo *sicuro* il programma, garantendone la proprietà di liveness. Un altro beneficio è quello di poter determinare quante unità di gas investire per eseguire con successo una callback nei casi in cui uno smart contract si appoggi ad un servizio esterno. Dal punto di vista degli attaccanti invece è possibile stimare quanto Ether è necessario investire per eseguire un attacco DoS, sebbene tali quantità siano svantaggiose dal punto di vista economico, rendendo poco invitante l'alternativa di compromettere uno smart contract.

3.2.1 Struttura del Tool

Per implementare ciascuna delle operazioni elencate sopra GASTAP si appoggia ad altri tool open-source. Questi, grazie a degli adattamenti, vengono utilizzati in sequenza al fine di realizzare l'architettura rappresentata in Figura 3.1.

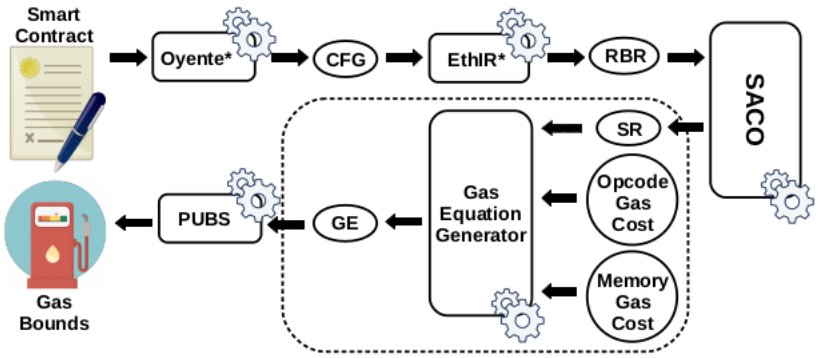

Figura 3.1: Architettura di GASTAP

1. **costruzione dei grafi control-flow (CFG)**: tale passaggio è realizzato con OYENTE*, un'estensione dell'omonimo tool OYENTE [25].

2. **decompilazione del codice di basso livello**: in questa fase il bytecode viene tradotto in una RBR (*Rule-Based Representation*) grazie ad ETHIR*, estensione del già citato ETHIR [3].

3. **deduzione delle relazioni di grandezza**: questo passaggio consiste nell'associare a ciascuna delle istruzioni in forma RBR le dimensioni dei dati con i quali interagisce. Quest'operazione è indispensabile per poter costruire le equazioni necessarie a calcolare i bound, e viene realizzata dal tool SACO [1], che produce le così dette SR (*Size Relations*).

4. **generazione delle equazioni di gas**: costituisce il core di GASTAP. Al fine di produrre le equazioni, il tool utilizza le SR insieme alla codifica dei costi delle istruzioni EVM, secondo le specifiche di [28]. Questi vengono suddivisi tra i costi richiesti dall'esecuzione del bytcode (*Opcode Gas Cost*) e quelli richiesti per l'uso della memoria (*Memory Gas Cost*).

5. **risoluzione delle equazioni fino a formare un bound**: per produrre il risultato finale GASTAP utilizza il solver PUBS [2], che risolve le GE (*Gas Equations*) producendo una formula chiusa dei costi in termini di gas.

3.2.2 Interfaccia Web

GASTAP è utilizzabile tramite un'interfaccia web, disponibile all'indirizzo https://costa.fdi.ucm.es/gastap.

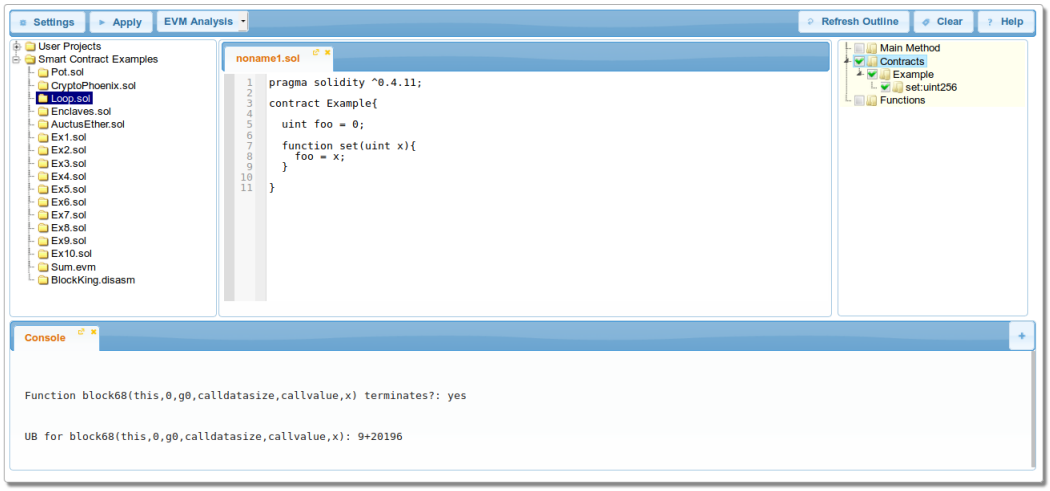

Figura 3.2: Interfaccia di GASTAP

È possibile scrivere un proprio programma in Solidity oppure scegliere uno degli esempi proposti dal menù "Smart Contract Examples".

Una volta selezionato il programma di input, cliccando su "Refresh Outline" GASTAP mappa le funzioni pubbliche del nostro programma, che vengono mostrate nella sezione di destra. Dopo aver selezionato i metodi che si vogliono analizzare, il pulsante "Apply" esegue l'analisi e produce un output nella Console.

Nell'esempio viene proposto un semplice programma che setta una variabile globale. Stando all'output prodotto, tale programma è garantito teminare, e l'*Upper Bound* (UB) ai consumi di gas è `9+20196`. Si noti che l'upper bound fornito è sempre nella forma **memory bound** + **opcode bound**, dove le cifre si riferiscono rispettivamente al bound dei costi di memorizzazione e a quello dei costi delle operazioni che compongono il programma.

3.3 Compilatore solc

Si tratta del compilatore ufficiale del linguaggio Solidity, utilizzabile da linea di comando. Il comando `solc --help` fornisce la spiegazione di ciascuna delle opzioni con cui può essere lanciato.

Per condurre i nostri test abbiamo utilizzato il compilatore con l'opzione `--gas`. In questa modalità solc è in grado di determinare soltato dei bound costanti; in tutti gli altri casi produce `infinite` come valore di output. Questo comportamento resta tutt'oggi una forte limitazione nell'utilizzo del compilatore, come provano le discussioni presenti nelle community degli utenti di Ethereum [10, 18, 23]. In molti casi questo limite viene attribuito ad una cattiva gestione di solc dei `JUMP` all'indietro nel bytecode EVM.

Ecco un esempio dell'utilizzo del compilatore sul programma raffigurato in Figura 3.2.

```
$~solc --gas example.sol

======= example.sol:Example =======
Gas estimation:
construction:
   5093 + 32800 = 37893
external:
   set(uint256):        20205
```

Confrontando i risultati ottenuti con quelli prodotti da GASTAP si evince che la stima del gas consumato dalla funzione `set()` corrisponde a quella determinata da solc. Dunque l'uso dell'analisi statica non fa sì che si perda accuratezza nel calcolo.

Nella Sezione 4.3.2 l'esempio verrà ripreso al fine di comprendere il bound.

3.4 Test Condotti

La nostra analisi è stata condotta su un insieme di programmi scritti in Solidity, disponibili nella repository [24].

Ad eccezione del contratto `four-function.sol`, proveniente dalla repository [7], gli altri smart contract sono stati sviluppati seguendo la documentazione ufficiale del linguaggio Solidity [12]. Si tratta di semplici programmi ad hoc per il testing dei costrutti di base del linguaggio di programmazione. Di seguito i casi che abbiamo trattato.

3.4.1 Operazioni di Assegnamento

I programmi `assignment*.sol` implementano dei contratti che contengono un numero arbitrario di operazioni di assegnamento.

```
//assignment2.sol

pragma solidity ^0.4.11;

contract B{

    function init(){
        uint number = 1;
    }

}
```

```
//assignment4.sol

pragma solidity ^0.4.11;

contract D{

    uint foo;

    function reset(){
        uint foo = 0;
    }

}
```

Abbiamo potuto verificare come l'operazione di assegnare un valore ad una variabile locale (`assignment2.sol`) o globale (`assignment3.sol`, `assignment4.sol`) costa una quantità di gas relativamente bassa, in media 140 unità. Aggiungendo un'operazione di assegnamento in più, che va dunque ad incrementare il valore precedente della variabile, questa stima in alcuni casi subisce una crescita notevole: passiamo da 140 unità a circa 20000.

```
//assignment1.sol

pragma solidity ^0.4.11;

contract A{

    uint number = 0;

    function init(){
        number = 1;
    }

}
```

Tale incremento è dato dalla presenza nel codice EVM dell'istruzione `SSTORE` (vedi Tabella 1.2). Si evince dunque che la semplice operazione di settare il valore di una variabile da 0 ad uno diverso da 0 ha un impatto notevole sulla performance del programma in termini di costi. Un caso simile si era verificato nel caso del programma `example.sol`.

3.4.2 Costrutto for

Testando i cicli for si è ottenuto un risultato interessante. La compilazione di questi programmi con solc produce sempre un bound infinito. Al contrario i test con GASTAP hanno prodotto una stima finita dei consumi. A titolo esemplificativo riportiamo di seguito i due esempi.

loop1.sol

```
//loop1.sol
//esegue la moltiplicazione
//di number*a

pragma solidity ^0.4.11;

contract Loop1{

    uint sum = 0;
    uint number;

    function multiply(uint a){

        for(uint i = 0; i<a; i++){
        sum = sum+number;
        }
    }

}
```

Gli output ottenuti con solc e GASTAP sono, rispettivamente:

```
$~solc --gas loop1.sol

====== loop1.sol:Loop1 ======
Gas estimation:
construction:
5099 + 39200 = 44299
external:
multiply(uint256):  infinite
```

e

```
GASTAP: 9+ (222+20476*nat(a))
```

loop2.sol

```
//loop2.sol
//somma i primi 10 elementi di un vettore

pragma solidity ^0.4.11;

contract Loop2 {

    function sum (uint[] nums) returns (uint sol) {
        sol = 0;
        for(uint i = 0; i < 10; i++)
                sol = sol+nums[i];
        }

}
```

Gli output ottenuti con solc e GASTAP sono, rispettivamente:

```
======= loop2.sol:Loop2 =======
Gas estimation:
construction:
111 + 59200 = 59311
external:
sum(uint256[]):     infinite
```

e

```
GASTAP: 3*max([4+nat(nums)+1,4+nat(nums)+2])+pow(max([4+nat(nums)
    +1,4+nat(nums)+2]),2)/512+(1746+3* (1/32))
```

Come si può evincere da quest'ultimo caso gli upper bound forniti da GASTAP possono essere parametrici. Nell'esempio il parametro è determinato dal valore in input di una delle funzioni pubbliche del contratto.

Più in generale possiamo dire che l'output prodotto da GASTAP è parametrico:

- nella dimensione dei parametri delle funzioni
- nello stato del contratto
- nei dati della blockchain dai quali dipendono i cosumi di gas (es. valore dell'ether)

3.4.3 Cicli for Annidati

Per verificare la gestione dei cicli for annidati si è implementato un semplice programma che con il metodo `suma(uint a)` esegue `a` incrementi della variabile locale `sum` attraverso un ciclo for.

```
//nested1.sol

pragma solidity ^0.4.11;

contract Nested1 {

    uint total_loops;

    // restituisce un valore uguale ad a, ottenuto sommando a volte
        1.
    // ad ogni iterazione incrementa la var total_loops.
    function suma (uint a) returns (uint sum) {
        sum = 0;
        for(uint i = 0; i < a; i++)
                sum = sum+1;
                total_loops = total_loops +1;
    }

}
```

Il programma è stato modificato in successione, inserendo un ciclo for annidato alla volta all'interno di `suma(uint a)`. Ad ogni incremento abbiamo nuovamente calcolato il bound alla funzione `suma(uint a)` con entrambi i programmi. Denotiamo con la variabile n il livello di annidamento dei cicli for. I risultati ottenuti sono mostrati nella Tabella 3.1.

Continuando ad incrementare il numero di cicli, si è potuto dare un bound al livello di annidamento. Per $n = 15$ GASTAP non riesce a mappare le funzioni nella outline. Questo implica che non può essere condotta l'analisi sul programma `nested15.sol`. Il limite dunque è dato dalla struttura del codice. L'esempio è mostrato in Figura 3.3

Tabella 3.1: Risultati dell'analisi dei contratti nested*.sol

n	solc	GASTAP
1	infinite	$15 + (20508 + 70 * nat(a))$
2	infinite	$15 + (20548 + 70 * nat(a) + 20276 * nat(a))$
3	infinite	$15 + (20588 + 70 * nat(a) + 20276 * nat(a) + 20276 * nat(a))$
4	infinite	$15 + (20628 + 70 * nat(a) + 20276 * nat(a) + 20276 * nat(a) + 20276 * nat(a))$

[1] La funzione nat è definita come nat(l)=max(0,l)

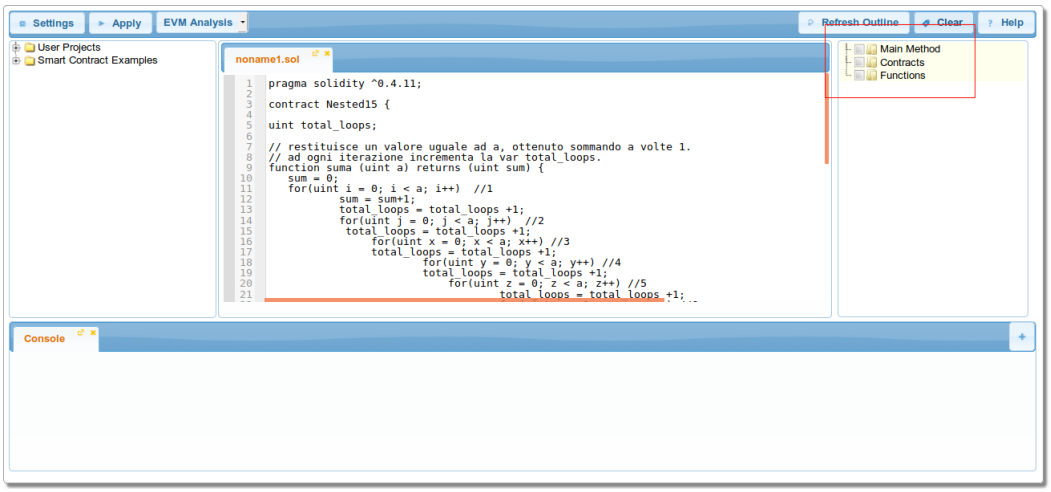

Figura 3.3: nested15.sol in GASTAP

Dai risultati ottenuti conducendo i nostri test è stato possibile ricavare la seguente formula di ricorrenza per il bound determinato da GASTAP per i programmi nested$n$.sol:

$$\forall 0 < n \leq 14 \quad \text{UB} = 15 + (20508 + (n-1) * 40 + 70 * nat(a) + (n-1) * 20276 * nat(a))$$

3.4.4 Costrutto while

Nel testare il comportamento dei tool di analisi di fronte a contratti contenenti dei cicli while abbiamo preso come input i file `while1.sol` e `while2.sol`.

while1.sol

Il primo programma implementa un algoritmo molto simile a quello di `nested1.sol`. Abbiamo sostituito il ciclo for con un while, dunque il numero di iterazioni è facilmente determinabile, poiché equivalente al valore del parametro `a`. Come nel caso dei cicli for solc produce un output infinito. Il risultato di GASTAP rispecchia un comportamento simile al caso precedente: viene determinato un bound parametrico in `a`.

```
//while1.sol

pragma solidity ^0.4.11;

contract A{

    uint number = 0;

    function init(uint a){

        while (a > 0) {
            number = number + 1;
            a = a - 1;
        }
    }

}
```

```
======= while1.sol:A =======
Gas estimation:
construction:
5093 + 37600 = 42693
external:
init(uint256):      infinite
```

e

```
GASTAP: 9+ (209+20271*nat(a))
```

while2.sol

Un caso differente si ha con il programma `while2.sol`. Questo contratto implementa un algoritmo che calcola la radice quadrata del parametro in input `x`. Anche in questo caso il bound dato da solc è infinito. Non solo GASTAP non riesce a determinare il bound - comportandosi quindi in modo simile a solc - ma addirittura va in errore.

```
//while2.sol

pragma solidity ^0.4.11;

contract B{

    function sqrt(uint x) returns (uint y) {

        uint z = (x + 1) / 2;
        y = x;

        while (z < y) {
            y = z;
            z = (x / z + z) / 2;
        }
    }
}

======= while2.sol:B =======
Gas estimation:
construction:
99 + 48800 = 48899
external:
sqrt(uint256):      infinite
```

e

```
GASTAP: evm_solver:non_terminating
```

3.4.5 Ricorsione

Per testare la ricorsione abbiamo utilizzato tre diversi programmi che implementano delle tecniche ricorsive diverse: `ricorsione-diretta.sol`, `ricorsione-indiretta.sol` e `ricorsione-multipla.sol`. Per tutti e tre gli input entrambi i tool di analisi falliscono nel tentativo di stimare un bound. In base a questi risultati possiamo dunque asserire che i programmi ricorsivi non sono gestiti correttamente dai tool di analisi statica.

Riportiamo di seguito il codice sorgente dei programmi utilizzati insieme ai risultati dell'analisi.

Ricorsione Diretta

Questo programma contiene una sola funzione pubblica, `fact(uint x)`. Questo metodo implementa l'algoritmo di calcolo del fattoriale di un numero. Consideriamo questo algoritmo ricorsivo *diretto* in quanto la funzione fact richiama direttamente sé stessa.

```
//ricorsione-diretta.sol

pragma solidity ^0.4.19;

contract Factorial{

    function fact(uint x) returns (uint y) {
        if (x == 0) {
        return 1;
        }
        else {
        return x*fact(x-1);
        }
    }
}
```

Gli output ottenuti con solc e GASTAP sono, rispettivamente:

```
======= ricorsione-diretta.sol:Factorial =======
Gas estimation:
construction:
93 + 42200 = 42293
external:
fact(uint256):  infinite
```

e

```
GASTAP: ../../bin/ethirweb   /tmp/ei_filesONMhje/noname1.sol  -
    entries  Factorial.fact:uint256 -type_file  solidity  > /dev
    /null ; cat /tmp/costabs/output.xml
```

Ricorsione Indiretta

Il seguente programma contiene uno schema di ricorsione *indiretta*: il primo metodo, `uno(uint n)` esegue una chiamata del secondo, `due(uint n)`, che a sua volta richiama direttamente il primo.

```
//ricorsione-indiretta.sol

pragma solidity ^0.4.19;

contract ricorsione_indiretta {

    function uno(uint n) returns (uint m){
        if (n < 1) {
            return 1;
        }
        else {
            return due(n - 1); // chiamata di due
        }
    }

    function due(uint n) returns (uint m){
        if (n < 0) {
            return 0;
        }
        else {
            return uno(n/2); // chiamata di uno
        }
    }
}
```

Gli output ottenuti con solc e GASTAP sono, rispettivamente:

```
======= ricorsione-indiretta.sol:ricorsione_indiretta =======
Gas estimation:
construction:
117 + 68600 = 68717
external:
due(uint256):   infinite
uno(uint256):   infinite
```

e

```
GASTAP: ../../bin/ethirweb   /tmp/ei_filesa6ppD5/noname1.sol  -
    entries  ricorsione_indiretta.uno:uint256
    ricorsione_indiretta.due:uint256 -type_file  solidity  > /
    dev/null ; cat /tmp/costabs/output.xml
```

Ricorsione Multipla

Il seguente programma contiene un metodo, `fib(uint x)` che calcola il numero di Fibonacci del parametro `x`. Tale metodo implementa una ricorsione *multipla* in quanto contiene più chiamate a sé stesso.

```
//ricorsione-multipla.sol

pragma solidity ^0.4.19;

contract Fibonacci {

    function fib(uint x) returns (uint y) {

        if (x == 1 || x == 2) {
        return 1;
        }

        else {
        return fib(x-1)+fib(x-2);
        }
    }
}
```

Gli output ottenuti con solc e GASTAP sono, rispettivamente:

```
======= ricorsione-multipla.sol:Fibonacci =======
Gas estimation:
construction:
99 + 46200 = 46299
external:
fib(uint256):   infinite
```

e

```
GASTAP: ../../bin/ethirweb   /tmp/ei_filesVQtMmj/noname1.sol  -
   entries  Fibonacci.fib:uint256 -type_file  solidity  > /dev/
   null ; cat /tmp/costabs/output.xml
```

3.4.6 Caso di Studio: four-function.sol

Questo contratto contiene quattro funzioni pubbliche che si richiamano a vicenda.

```
//four-function.sol

pragma solidity ^0.4.11;

contract Sum {

    function suma () returns (uint sol) {
        sol = 0;
        for(uint i = 0; i < 5; i++)
                sol = sol+11;
        hola();
        adios(10);
    }

    function hola() {
        uint i = 0;
        i = i+15;
    }

    function adios(uint m) {
        uint c = 14;
        c = c+m;
        comer(c);
    }

    function comer(uint x) {
        x = x*x;
        hola();
    }

}
```

La Tabella 3.2 mostra i risultati di solc e GASTAP a confronto. Nonostante il programma contenga delle chiamate ad altri metodi, si differenzia dal caso precedente dove era presente la ricorsione multipla. In questo caso non si verificano cicli nel grafo delle

chiamate di funzione, dunque `four-function.sol` non è ricorsivo.

metodo	**solc**	**GASTAP**
adios(uint256)	314	9+305
comer(uint256)	302	9+293
hola()	226	9+217
suma()	infinite	15+802

Tabella 3.2: Risultati dell'analisi del contratto four-function.sol

Ciò che emerge è che nel caso di bound costanti i risultati dei due tool sono uguali. È da notare il caso della funzione `suma()`: solc non è in grado di produrre un bound. Questo è dovuto alla presenza del ciclo for all'interno del corpo della funzione, insieme alla chiamata della funzione `adios()`. Rimuovendo le relative linee di codice riusciamo infatti ad ottenere un bound.

```
//four-function.sol:suma()

function suma () returns (uint
    sol) {
     sol = 0;
     //for(uint i = 0; i < 5; i
        ++)
     //         sol = sol+11;
     hola();
     //adios(10);
}
```

```
=== four-functions.sol:Sum ===
Gas estimation:
construction:
123 + 74600 = 74723
external:
adios(uint256):     314
comer(uint256):     302
hola():     226
suma():     275
```

Conclusioni

L'utilizzo dell'analisi statica nella verifica delle proprietà di sicurezza dei contratti di Ethereum ha riscosso successo recentemente, portando allo sviluppo di alcuni software che combinano diverse tecniche di analisi statica. Solo una piccola porzione di questi programmi si focalizza sui consumi di gas; abbiamo visto MadMax [15], che conduce un'analisi orientata a individuare vulnerabilità nel codice legate al gas. Un altro tool simile è Gasper [6], che grazie all'analisi dei consumi è in grado di individuare dei pattern ai quali è associato un elevato consumo di gas; offre dunque un servizio di ottimizzazione del codice. Nessuno di questi però è in grado di produrre dei bound.

La stima dei consumi di gas resta quindi un topic poco trattato. I lavori condotti in questa direzione sono ancora pochi: oltre al tool Gastap [4], sul quale ci siamo focalizzati durante questa trattazione, è importante citare anche il lavoro di Marescotti et al. [22], che sottolineando l'importanza del tema, propone due algoritmi per la stima dei consumi di gas; questo approccio tuttavia non è stato ancora implementato, motivo per cui non è stato possibile includerlo in questo lavoro.

Va inoltre considerato che questi strumenti hanno ancora molte limitazioni, impedendo la verifica di programmi sofisticati. Costrutti come i cili while o la ricorsione non vengono gestiti correttamente, ponendo una forte limitazione allo sviluppatore che desideri verificare i consumi del proprio programma. È questo uno dei motivi per cui l'analisi statica non è ancora molto utilizzata nella stima del gas, la ricerca dovrebbe quindi cercare di studiare i limiti di questo problema, assieme a tecniche di analisi statica più espressive.

Bibliografia

[1] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, German Puebla, and Guillermo Román-Díez. Saco: Static analyzer for concurrent objects. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 562–567, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[2] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *International Static Analysis Symposium*, pages 221–237. Springer, 2008.

[3] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In *International Symposium on Automated Technology for Verification and Analysis*, pages 513–520. Springer, 2018.

[4] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. Gastap: A gas analyzer for smart contracts. *CoRR*, abs/1811.10403, 2018.

[5] Giorgio Ausiello, Fabrizio d'Amore, and Giorgio Gambosi. *Linguaggi modelli complessità*. F. Angeli, 2003.

[6] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446. IEEE, 2017.

[7] costa group. Ethir: A framework for high-level analysis of ethereum bytecode, 2018.

[8] Fangfang Dai, Yue Shi, Nan Meng, Liang Wei, and Zhiguo Ye. From bitcoin to cybersecurity: A comparative study of blockchain application and security issues. In *2017 4th International Conference on Systems and Informatics (ICSAI)*, pages 975–979. IEEE, 2017.

[9] Phil Daian. Dao attack.(2016). *URL: http://hackingdistributed. com/2016/06/18/analysis-of-the-dao-exploit*, 2016.

[10] denisglotov (https://github.com/denisglotov). Gas estimates infinite, 2018.

[11] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic sat-compilation of planning problems. In *IJCAI '97: IJCAI-97, Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176, Nagoya, Aichi, Japan, aug 1997.

[12] Ethereum. Solidity docs, 2018.

[13] ethereum. The solidity contract-oriented programming language, 2019.

[14] Maurizio Gabbrielli and Simone Martini. *Linguaggi di programmazione: Principi e Paradigmi*. McGraw-Hill, 2011.

[15] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):116, 2018.

[16] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Foundations and tools for the static analysis of ethereum smart contracts. In *International Conference on Computer Aided Verification*, pages 51–78. Springer, 2018.

[17] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. Kevm: A complete semantics of the ethereum virtual machine. Technical report, 2017.

[18] Sergey Potekhin (https://ethereum.stackexchange.com/users/4406/sergey potekhin). How to get the cost (in gas) of the non-constant function call?, 2017.

[19] Marco Iansiti and Karim R Lakhani. The truth about blockchain. *Harvard Business Review*, 95(1):118–127, 2017.

[20] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

[21] Minhaj Ahmad Khan and Khaled Salah. Iot security: Review, blockchain solutions, and open challenges. *Future Generation Computer Systems*, 82:395 – 411, 2018.

[22] Matteo Marescotti, Martin Blicha, Antti EJ Hyvärinen, Sepideh Asadi, and Natasha Sharygina. Computing exact worst-case gas consumption for smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*, pages 450–465. Springer, 2018.

[23] medvedev1088 (https://ethereum.stackexchange.com/users/18932/medvedev1088). Infinite gas estimation from solc for simple function, 2018.

[24] melastone. tesi-triennale/materiale/smart-contract-ex/, 2019.

[25] melonproject. Oyente: An analysis tool for smart contracts, 2018.

[26] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[27] Porfirio Tramontana. 10-analisi statica. University Lecture, 2019.

[28] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

Ringraziamenti

Un sentito ringraziamento al professore Ugo Dal Lago, che con la sua disponibilità e pazienza mi ha supportata durante lo sviluppo di questa tesi, fornendo sempre un contributo positivo.

Grazie a Luca e Marcella per il loro supporto, ma soprattutto aver scelto di fidarsi di me.

Ringrazio infine coloro che mi sono accanto. Se non potessi condividere questo momento con voi non avrebbe lo stesso valore. A voi il mio affetto più sincero; sono profondamente grata per avervi nella mia vita.