Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Stimare i Consumi di Gas nell'Esecuzione di Smart Contract in Ethereum

Relatore:
Chiar.mo Prof.
Ugo Dal Lago

Presentata da:
Melania Ghelli

Sessione II
Anno Accademico 2018-2019

Questa è la Dedica*:*
ognuno può scrivere quello che vuole,
anche nulla ...

Indice

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 [11] 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 [12], 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 [3].

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 [10], offrendo innovazione gra-

zie 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 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 peró risiede nel fatto che questo tipo di programmi diventano un bersaglio facile per attacchi malevoli. THE DAO Vedremo come queste vulnerabilitá possono essere sfruttate per ottenere comportamenti dannosi per la rete.

Dato il valore monetario associato agli smart contract il rischio che si corre in caso di attacchi informatici è una perdita di denaro. Per questo motivo è necessario individuare

possibili criticitá 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. Poter conoscere a priori quest'informazione permetterebbe non solo un investimento adeguato da parte degli utenti, ma anche uno strumento di prevenzione da possibili attacchi.

Attualmente non esistono strumenti in grado di calcolare con precisione la quantitá di gas richiesto durante una computazione. Cercheremo di capirne le ragioni ma soprattutto di individuare dei margini di miglioramento.

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 simile ad una lista *crescente*, 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*. (Algoritmo che mappa dati di dimensione arbitrari 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 è 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 dunque 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 SI POTREBBERO INSERIRE ESEMPI CITATI NELL'ARTICOLO [10], sebbene ad oggi il suo uso più conosciuto sia 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 [12]. 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 La piattaforma Ethereum

1.3 La Ethereum Virtual Machine

1.4 Gli smart contract

1.5 Ethereum, gli Smart Contract e la EVM

All'interno di quest'elaborato verrà presa in considerazione solo il network Ethereum, una piattaforma decentralizzata basata su una blockchain, che come Bitcoin possiede una

propria valuta: l'*ether*.

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". Sebbene 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 [6]. 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.

EVM: The virtual machine that forms the key part of the execution model for an Account's associated EVM Code.

EVM Code: The bytecode that the EVM can natively execute. Used to formally specify the meaning and rami- fications of a message to an Account.

1.6 Il ruolo del gas

Dallo yellow paper:

Gas: The fundamental network cost unit. Paid for exclusively by Ether (as of PoC-4), which is converted freely to and from Gas as required. Gas does not exist outside of the internal Ethereum computation engine; its price is set by the Transaction and miners are free to ignore Transactions whose Gas price is too low.

Per *gas* si intende l'unità di misura dello sforzo computazionale richiesto dalla EVM per eseguire ciascuna istruzione. Diremo quindi che eseguire uno smart contract *costa*

una certa quantità di gas. Nello specifico, 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 [14] sono riportati nella Tabella 1.1.

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 1.1: Costi espressi in gas di alcune istruzioni della EVM

Il gas viene pagato in ether dagli utenti che intendono far eseguire una *transazione*. Con transazione si intende l'azione di creare uno smart contract o di chiamarne delle funzioni e di pagare un miner per far eseguire la transazione stessa.
Per fare in modo che la sua transazione venga scelta, un utente stabilisce la quantità di gas che è disposto a pagare per farla portare a termine. Il miner poi, in base a questo parametro, sceglie quali transazioni eseguire TRA QUALI??????????. È importante

fornire un'adeguata somma: i miner, al termine della transazione, possono beneficiare dell'ether destinato all'acquisto di gas che è rimasto inutilizzato. Questo significa che gli smart contract con più probabilità di essere scelti sono quelli degli utenti disposti a pagare di più.
La scelta adottata da Ethereum 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.
Dal momento che il gas viene pagato anticipatamente, può succedere che durante la sua esecuzione un programma ecceda la quantità che ha a disposizione. Questo comportamento è indesiderato, in quanto comporta spiacevoli conseguenze. La 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.
La conseguenza indiretta invece è il possibile blocco permanente dello smart contract. Quando durante l'esecuzione di una transazione un'istruzione richiede una quantità di gas superiore a quella disponibile, la EVM solleva un'eccezione di tipo *out-of-gas* e interrompe la transazione. Qualora lo smart contract non preveda una gestione di questo tipo di eccezione resterà bloccato per sempre.

Capitolo 2

Analisi statica

L'analisi statica è un processo di valutazione di programmi che rientra tra le tecniche di verifica del sotware. 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.

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 [4]. 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 sound. 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.

Dall'altro lato 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 *soundness*. Per essere rigorosa infatti l'analisi dinamica dovrebbe coprire ogni possibile configurazione del programma.

Le tecniche di analisi possono essere suddivise in due tipologie in base al loro obiettivo. 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.

2.2 Tecniche di analisi statica in Informatica

Di seguito daremo una panoramica sulle principali tecniche [13] di analisi statica.

2.2.1 La compilazione

Questa tecnica identifica i controlli effettuati dal compilatore, che variano a seconda del linguaggio di programmazione del codice. Le anomalie che possono essere catturate da questo tipo di analisi possono variare, ma in generale comprendono l'incoerenza dei tipi, la mancata dichiarazione delle variabili e il codice non raggiungibile dal flusso di controllo.

2.2.2 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.

2.2.3 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.

2.2.4 Walktrough

Molto simile alla tecnica di precedente nelle 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.

2.2.5 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.

2.2.6 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.

2.2.7 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 [8] questo framework offre la possibilità di analizzare i programmi al fine di verificarne le proprietà di sicurezza. Tali proprietaà, 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 è garantito essere *sound*.

MadMax [7] 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 [9] 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 [1] 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 [], 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 Stimare i consumi di GAS

L'ultima categoria di software che vedremo è la più interessante dal punto di vista della ricerca condotta. 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 di questo elenco, per poi trattarli in modo più dettagliato nei capitoli successivi.

solc [5] 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 [2] 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

Il problema del gas

Spieghiamo il problema del gas nell'esecuzione degli smart contract in ehtereum.

Perchè ci serve la stima??

Se io so quanto è il gas associato ad ogni istruzione, mi basterà sapere come verrà tradotto il mio programma in bytecode. Da lì potrei stabilire il costo fisso. Eh no! Il gas viene consumato in circostanze differenti.

Il costo totale - espresso in gas - richiesto per eseguire un programma è determinato in base a 3 fattori:

1. il costo intrinseco di ciascuna istruzione di basso livello; questo valore è fissato.
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 previsti, altri possono essere determinati soltanto durante l'esecuzione del contratto. Essendo difficili da prevedere, potrebbero far sì che la quantità di gas richiesta in fase di esecuzione ecceda quella messa a disposizione dal committente prima. Dunque conoscere questi costi prima del lancio del programma in rete permetterebbe agli utenti di investire somme di denaro adeguate. Soltanto delle tecniche di analisi precise ci permettono di stimare questi consumi, poichè

ci permettono di calcolare in anticipo quali “sorprese” riserverà il codice durante la sua esecuzione.

Capitolo 4

Risultati sperimentali

Lo scopo di questo capitolo è quello di fornire una panoramica sugli esperimenti condotti.

Spiego che cosa ho fatto.

Dato un set di smart contract molto semplici, sono state condotte delle analisi utilizzando i software disponibili precedentemente elencati. A partire dai risultati ottenuti si è individuato quali di questi programmi fornissero un bound esplicito ai consumi di gas associati all'esecuzione dei programmi.

Conclusioni

Bibliografia

[1] 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.

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

[3] 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.

[4] 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.

[5] Ethereum. Solidity docs, 2018.

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

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

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