UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

FUZZER TO ANALYZE SMART CONTRACT PERFORMANCE

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

DANIEL BERNARDO SOTO GAMBOA

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
ALEJANDRO HEVIA ANGULO
ANDRÉS MUÑOZ ORDENES
FEDERICO OLMEDO BERÓN

SANTIAGO DE CHILE
2020

Resumen

La blockchain de Ethereum promete una infraestructura ideal para desarrollar sistemas parcial o totalmente descentralizados, pero existe una diferencia fundamental entre su máquina virtual, y la de lenguajes como Java: es necesario pagar para ejecutar código en ella. Como es Turing-completa, no se puede determinar *a priori* si es que un programa será terminante o no, lo que crea un problema único en este ecosistema: *estimar el costo de ejecutar un programa*. Herramientas actualmente disponibles para estimar estos costos sólo generan una cota superior del posible costo, la que puede llegar a ser infinita y no entregarle información útil al desarrollador del sistema.

Esta memoria tiene como objetivo general desarrollar una herramienta que genere distribuciones de costos de gas para las funciones de un contrato de Solidity (similar a una clase de Java), como un acercamiento al problema de *estimación de gas*. Además, como objetivos específicos se busca analizar patrones en el código fuente y cómo se relacionan a ciertas distribuciones de costos y generar recomendaciones a próximos autores de código Solidity para escribir código más seguro.

Comienza con una etapa investigativa de Ethereum, en particular sobre la máquina virtual en la cual se ejecutan *transacciones* (similares a ejecutar un método en Java), seguida por un estudio de un lenguaje que corre en esta máquina virtual: Solidity y cómo se diferencia de otros lenguajes populares como Java. Continúa analizando el sistema de gas implementado para controlar el uso de recursos computacionales de cada minero, y herramientas actuales que intentan resolver el problema de *estimación de gas*.

Luego se define la estructura de la herramienta desarrollada, basándose en conceptos de *fuzz testing*. Para esto se desarrollan técnicas únicas para este ecosistema, como la generación de direcciones en la blockchain y de transacciones válidas. El análisis simbólico se introduce para fortalecer un punto débil de la herramienta: su aleatoreidad. Un experimento para evaluar esta herramienta es diseñado para ser ejecutado en una base de datos de 94,878 contratos.

A partir de los resultados obtenidos en el experimento se genera un gráfico que resume el consumo de gas para un servicio dado y se definen clasificaciones para cada uno basadas en los costos obtenidos y en cómo se relacionan a la estimación estática generada por el compilador de Solidity. Cada clasificación se correlaciona con ciertas estructuras en el código que las generó y luego son usadas para realizar un análisis global del comportamiento de la herramienta.

Se concluye la memoria presentando posibles falencias en la lógica de la herramienta, explicando el efecto de agregar análisis simbólico a esta, resumiendo puntos donde concentrar el trabajo futuro y dando recomendaciones a autores de código Solidity para generar código más seguro en el futuro.

Una versión resumida de este trabajo fue presentada en ICSME-NIER 2020 [11]

Abstract

The Ethereum blockchain provides the ideal infrastructure to develop partially or completely decentralized systems, but its virtual machine has a fundamental difference with that of a language like Java: it requires payment for code execution. Since it is Turing-complete, it is not possible *a priori* to determine if a given program will halt or not, which generates a unique problem in this ecosystem: *estimating the cost of executing a program.* Currently available tools to estimate these costs only generate an upper bound for this cost, which might be infinite and of little use to the developer of a system.

This thesis' main objective is to develop a tool that generates distributions of gas costs for each function in a Solidity contract (akin to a Java class), as an approach to the *gas estimation problem.* Furthermore, specific objectives include analyzing patterns in source code and how they relate to a given cost ditribution, and generating recommendations for future Solidity code authors to write safer code.

We start by performing an investigation of Ethereum, in particular its virtual machine on which *transactions* (akin to a method call in Java) are executed, followed by a study of a language that targets this virtual machine: Solidity and how it differentiates itself from other popular languages like Java. As a next step, we analyze the *gas system* implemented to control the use of a miner's computational resources, and current tools that attempt to solve the *gas estimation problem.*

The definition of the developed tool's structure follows, using concepts from *fuzz testing.* For this purpose we develop techniques unique for this ecosystem, such as the generation of valid blockchain addresses, and valid transactions. As a way to reinforce a weak point of the tool, its randomness, we introduce symbolic analysis. Afterwards, we define an experiment to evaluate the tool on a database of 94,878 contracts.

Using these results we generate a *blueprint* that summarizes the gas consumption for each service and classifications are defined based on the gas costs and how they compare to the static estimation generated by the Solidity compiler. Each classification is then correlated with a given structure in its source code and then later used for a global analysis of the tool's behaviour.

The thesis concludes by presenting weak-points of the tool, explaining the effect of adding symbolic analysis on the fuzzer, summarizing where future work should be focused and finally making recommendations to future solidity code authors in order to generate safer code in the future.

A summarized version of this work was presented at ICSME-NIER 2020 [11]

Contents

Introduction **1**

1. Ethereum & Gas Costs **3**
1.1. Blockchains 3
1.2. Ethereum & Solidity 4
1.3. The Gas System 6
1.4. Gas Estimations 6
1.4.1. Static Opcode Analysis 7
1.4.2. GASTAP 8

2. Estimating Gas Costs **9**
2.1. Fuzz Testing 9
2.2. Argument Generation 9
2.3. Transaction Execution 10
2.4. Symbolic Analysis 11
2.5. Implementation Design 12
2.6. Contracts and Experiment Execution 15

3. Results **16**
3.1. Blueprints 16
3.2. Cost trend 17
3.3. Classification 18
3.4. Quantitative analysis 19

Conclusion **22**

Bibliography **23**

Appendices **25**

Appendix A. Example Ballot Contract **25**

Appendix B. Fuzzer Procedures for Solidity Types **29**
B.1. Fixed Size Types 29
B.1.1. Booleans 29
B.1.2. Integers 29

B.1.3. Addresses 29
B.1.4. Fixed-Point Numbers 29
B.1.5. Characters 30
B.1.6. Fixed-size Byte Arrays 30
B.1.7. Function Pointers 30
B.1.8. Fixed-size Arrays 30
B.2. Dynamic Size Types 30

Appendix C. Contract Composition **31**

Table Index

1.1. Resulting estimations from a sample contract 8

Figure Index

1.1. Basic structure of a blockchain. 3

2.1. Frequency of each kind of operation found within `require` statements 11
2.2. Frequency of each operator used within binary operations 12
2.3. Architecture of the developed tool. 13
2.4. Structure used to store fuzzers in the fuzzing layer. 14
2.5. Class hierarchy defined by all the implemented fuzzers. 15

3.1. Gas cost of a service in the less than, constant, category 17
3.2. Gas cost of a service in the less than, constant, category 18
3.3. Heatmap indicating the obtained classifications 19
3.4. Gas cost of the service `transferOwnership`. 21
3.5. Heatmap displaying the results of the experiment with symbolic analysis. . . 21

C.1. Frequency of solidity versions used in the contract database. 31

Introduction

Blockchains started their history slightly more than a decade ago, when on the 31st of October of 2008 Satoshi Nakamoto published the first paper about a peer-to-peer electronic cash system: Bitcoin [10]. This technology is based on the premise that every user (also known as *nodes*) would contribute to keeping a decentralized ledger of the balances of all accounts in the network. It earns its name from being updated in discrete steps, named *blocks*. Where each block contains many *transactions* which define how each account's balance changed.

Since the blockchain is decentralized, recent blocks can't be trusted, since they may not be replicated in the whole network. However, older blocks can be considered as final, thanks to its *consensus mechanism* making it statistically impossible to change blocks a certain amount behind the latest one[1]. This *eventual consensus* can be maintained as long as no single entity controls more than 50 % of the computing power of the whole chain.

In order to update the chain, a user or *miner* must take the chain's latest *block*, along with a set of transactions to compute the next block. Then, the user must hash the new block and obtain a hash that is lower than a value defined by the chain known as its *difficulty*. If the problem is solved by a user, or *miner*, then the block is then published to the network, and other nodes can validate the work and consider it as a valid block. As long as no ill-intentioned entity controls over 50 % of the network's computing power, then incorrect blocks will be safely ignored, and the rest of the chain will continue to be built upon correct blocks. This process constitutes the consensus mechanism which allows the blockchain to be trusted.

Each transaction published in the blockchain is defined as bytecode, along with signatures to authenticate the outgoing transfers. This bytecode is fed to a simple always-terminating virtual machine, to compute the next block in the blockchain.

Following Bitcoin's creation, new ideas that built upon its foundations started being published. Among these is Ethereum [12], which differentiates itself by allowing its virtual machine to be Turing-complete. This opened the doors for programming languages that target the Ethereum VM (henceforth *EVM*), allowing it to act as more than just a decentralized ledger. Every node could compute changes to complex decentralized systems built on the Ethereum blockchain, making it similar to a distributed computer and database in exchange for its cryptocurrency *ether*.

However, this decentralized computing power comes at a cost. Since the EVM is turing-

[1]For bitcoin, this is considered to be at least 6 blocks behind the most recent one.

complete there is no guarantee that code being executed by a node will finish running. To address this problem, the Ethereum designers came up with a *gas system*, which required payment for transaction executions. This payment is measured in *gas*, a unit measuring a given computational cost. Whenever a transaction is sent, it must be sent with a fixed amount of gas. During computation each processed opcode by the VM adds gas to the total gas spent. If this value is at any point higher than the amount sent with the transaction then the execution stops, all side-effects are rewound, and the executing node keeps the spent gas.

One language that targets the EVM is Solidity, which is used to program *smart-contracts* that act similarly to a class in regular object-oriented languages. These can be uploaded to an address in the Ethereum blockchain to later be paid (in ether) to execute their code or store some data. Real-world examples of Ethereum smart-contract usage include: *Gods Unchained* [2], a collectible card game where each card's stats and ownership is stored on-chain; *Augur* [1], an unlimited betting site and *Golem* [3], a decentralized computing framework.

An emerging problem with this decentralized computer is that, because of the halting problem, exact calculations nor upper bound estimations of how much gas any given transaction will cost cannot be made without first executing said code. If accurate estimations of how much gas will be spent cannot be made, final users run the risk of losing all ether and gas sent with a transaction and getting none of the desired results. This is known as the *gas problem*.

In this thesis we introduce Ethereum, the EVM, Solidity and the gas estimation problem (Section 1), develop a tool that estimates gas costs for each of a contract's functions using fuzz testing (Section 2) and present the experimental results of said tool, tested on a large database of *smart-contracts* (Section 3).

This thesis' work was summarized in a paper [11], which will be presented at ICSME - NIER 2020.

Chapter 1

Ethereum & Gas Costs

In this chapter, all important concepts relating to blockchains, Ethereum, the Ethereum Virtual Machine, Solidity and the gas system are explained. Afterwards the gas estimation problem is presented, and current gas-estimation tools are analyzed.

1.1. Blockchains

A blockchain is essentially a decentralized database. Although originally designed to record financial transactions, a blockchain can ledge transactions about anything. It does so by taking an initial state, the *genesis* block, and applying more blocks on top of it. Each block is composed of a reference to the parent block, and a list of transactions to apply. This *chaining* of blocks, as seen in Figure 1.1, earns it its name.

Each block acts as a state transition, defining the changes to apply on the previous state. In order to compute the current state of a blockchain, one would have to start at the *genesis* block and apply the transactions indicated in each following block until up to date with the latest block. Transactions often need authorizations from one or more accounts in the blockchain. These authorizations are done using public key cryptography. Each account has a public address in the blockchain, which is some form of their *public key*, and can *sign* a transaction by using their *private key*.

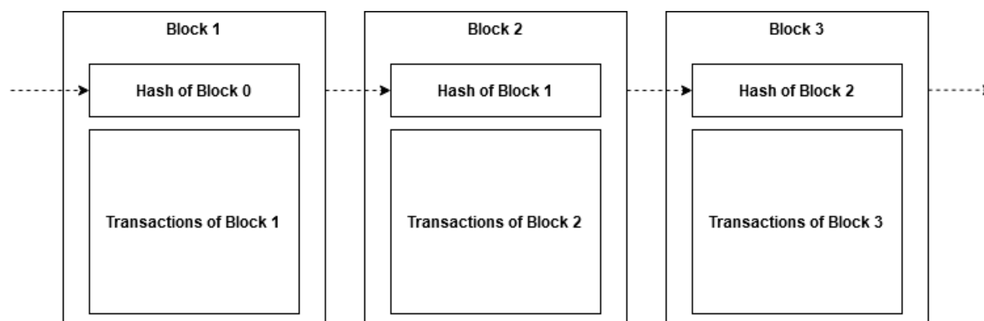

Figure 1.1: Basic structure of a blockchain.

When a miner wants to add a new block on top of the blockchain, they must select the transactions they want to include, ensure they are valid by executing them on a virtual machine, and then assemble the next block using these transactions and the hash of the previous block, which acts as its identifier. This block also includes a *nonce*, a positive integer. This block is then hashed multiple times, changing the *nonce* each time on order to obtain a new hash, until the result is lower than a value set by the blockchain known as its *difficulty*.

This process is the heart of *Proof-of-Work*, the consensus algorithm used in most blockchains, and it is what allows the blockchain to be decentralized, yet trustable. Since computing a hash that is low enough is a computationally expensive problem, it is unfeasible for a dishonest node to publish many invalid blocks to its benefit since they must compute said hash for each one in order for other nodes to consider them valid. However, if said dishonest node possessed more than 50 % of the chain's computational power, then they'd be able to continue publishing invalid blocks at a faster rate than the rest of the blockchain can publish valid nodes, losing the decentralized property characteristic of blockchains.

Since dishonest nodes can still publish invalid blocks occasionally, blocks are considered *confirmed* when a certain number of blocks are published on top of them. This number depends on the frequency of blocks being published to the blockchain. For Bitcoin, a block is published roughly every 10 minutes, and a block is considered confirmed after 6 other blocks are published on top of it. For Ethereum, a block is published every 10 to 19 seconds, and a block is considered confirmed after 30 confirmations.

Blockchains can be used as a database or computing service with high availability, since as long as there are active nodes, transactions will be able to take place. This comes at the cost of the blockchain's confirmation time to ensure the data's integrity.

1.2. Ethereum & Solidity

Ethereum's main differentiation as a blockchain is its Turing-complete virtual machine, which allows it to not only store data, but allow arbitrary code to run on the blockchain. This enables users to develop smart systems that run on the blockchain. For example, the trading card game *Gods Unchained* [2] stores all of the information for its cards in the blockchain, including all their properties and owners. Another example is *Golem* [3], which provides a decentralized computation framework where computationally hard problems, such as rendering a video, can be uploaded to outsource the work to the chain's nodes.

Each account in the Ethereum blockchain contains the following information:

- Its balance.
- A nonce, to count how many transactions have been executed from the account's address.
- A storage section, which can store arbitrary data.
- The hash of the EVM bytecode belonging to the account. Cannot be changed after construction.

Solidity is a programming language that targets the EVM. Its code, and that of other

languages that target the EVM, is known as *smart-contracts*. Solidity is an object-oriented, strongly typed language, similar to Java in many aspects. For example, a Solidity contract roughly corresponds to a Java class and a Solidity function is similar to a Java method. These may be executed either by another function, or through a transaction. As such, a transaction is similar to the classical remote method call.

Solidity's primitive types are much like those of Java, with the addition of primitive types for maps, lists and addresses. The last one stores the identifier for a given account, and can be used to call transactions on it, or transfer some ether.

Differences start to be noticeable upon a closer inspection of Solidity code. Solidity has no standard library, so most computation must be written by the developer. Since Solidity code is immutable, *i.e.,* after being uploaded to the blockchain no changes can be made to it, references to external contracts are rare because of the tight coupling they produce. If an error were found in a contract that acts as a dependency, the depending contract cannot be updated to change the dependency.

Here is an example of a simple contract, which stores how much ether has been deposited into it over its lifetime:

Listing 1.1: Contract which allows for deposits to be made by its owner

```
1  contract SimpleStorage {
2    address owner;
3    uint amount;
4    constructor() public {
5      owner = msg.sender;
6    }
7    function pay() public payable {
8      require(owner == msg.sender);
9      amount += msg.value;
10   }
11 }
```

Lines 4 - 6 are the contract's constructor, which gets executed when the contract is deployed on a blockchain. Line 5 records the sender of the transaction that deployed the contract into a variable called `owner`. As such, the owner variable references an address in the blockchain that uniquely identifies the owner of the contract.

Lines 7 - 10 define a *payable* public function, otherwise known as a *service*. These special functions receive ether sent with a transaction and store it to the contract's address. Line 8 has a `require` statement, which verifies that the transaction's `sender` is the same address as the contract's `owner`. If this validation were to fail, then the service's execution would stop, and the executing node would keep all of the sent gas with the transaction. Finally, line 9 adds the paid amount to a counter within the contract. It is important to note that this assignment does not need to be made to transfer the funds, its sole purpose is to track all of the received ether in the contract's history.

1.3. The Gas System

In a public blockchain, anyone can publish a transaction for any public service. A public service may be designed to produce and store valuable data (*e.g.,* a record of a financial transactions, as in Listing 1.1) in the blockchain. As an incentive for network members (called *miners*) to execute transactions, Ethereum offers a system which allows miners to be paid for each executed transaction. This system operates around a unit called *gas*.

Gas is a unit describing elementary computational steps. A unit of gas corresponds to the cost of executing one atomic instruction: a bytecode. For example, multiplying is a simple operation that requires few computational units (5 gas) while a significantly more complex operation, such as getting the balance of an account costs 400 gas[1].

The system sets a few restrictions upon transactions, which must be satisfied for a miner to attempt their computation:

- It must include a fixed amount of ether in order to pay for the execution (e).
- It must include a rate of conversion between ether and gas, called the *gas fee* (f).
- It must include a threshold called *gas limit* (g_L), that represents the maximum amount of gas the author is willing to pay to the computing miner for the spent resources during execution (both CPU cycles and time).

After executing a transaction, the miner earns ether depending on the execution's result. If g is the amount of consumed gas:

- If the execution failed because of a runtime error, then the miner gets all of the ether intended to pay for the gas in the transaction: e.
- If the execution failed because of an out-of-gas error (*i.e.,* the consumed gas was greater than the gas limit), then the miner gets an amount equal to: $g_L \cdot f$.
- If the execution completed without errors, then the miner gets an amount equal to: $g \cdot f$, and the author is refunded $\mathrm{e} - g \cdot f$ ether. Where g is the amount of gas consumed during the execution.

As such, whenever a transaction is published it should set these values such that $\mathrm{e} = g \cdot f$.

1.4. Gas Estimations

Estimating how much gas will be spent by any given transaction has always been in the interests of developers to ease the optimization of their contract's gas expenditure. For a contract's user knowing how much gas needs to be sent along with a transaction is also very important, since the gas exceeds should be minimized in order to lose less gass in case a runtime error happens, and all ether to be paid for gas-coverage is lost. On the other hand, it is of the utmost importance to set a gas-limit that manages to cover the costs of the transaction's execution, since as soon as the consumed gas surpasses said limit, all effects produced by the transaction are disposed, and all sent gas is lost.

[1] `https://github.com/crytic/evm-opcodes`

Solidity designers predicted this and included within their compiler a simple gas estimation system.

1.4.1. Static Opcode Analysis

Solidity estimates the upper limit of the gas cost of a given contract by performing static analysis on the contract's compiled bytecode and returns an upper bound for each function's gas-consumption. These estimations do not take any contract state into account, which causes frequent mis-estimations. This approach estimates infinite upper bounds in the following cases:

1. If some part of the contract's state with variable size is used in the function.
2. If unavailable bytecode is called, for example a call to an external contract's function.
3. If any ether transfers are made within the function's call tree.

In particular, Item 2 has led Solidity developers to heavily rely on code duplication. Sharing common interfaces and contracts to be later bundled with the rest of the contract's code, and finally reuploaded to some address in the blockchain.

As an example, consider the contract in Listing 1.2. It requires payment in *wei*, which is equal to 10^{-17} ether, and for each *wei* transferred, it computes the next fibonacci number. This service is estimated with an infinite gas cost, which gives no information to its users about how much gas should be sent along with the transaction.

Listing 1.2: Contract with gas-estimation problems

```
contract FibonacciCalculator {
  uint previous;
  uint number;
  constructor() public {
    previous = 0;
    number = 1;
  }
  function increase() public payable returns (uint fibonacci) {
    return doWork(msg.value);
  }
  function doWork(uint amount) private returns (uint fibonacci) {
    for (uint i = 0; i < amount; i++){
      uint t = number;
      number = number + previous;
      previous = t;
    }

    return number;
  }
}
```

1.4.2. GASTAP

GASTAP [8] also operates on the bytecode of a contract but does not generate a number for their upper-bound. Instead it produces two functions, parametrized over the contract's state to generate an always accurate, never infinite, upper-bound. One of these functions estimates the gas costs associated with memory consumption, and the other the gas costs associated with opcode processing.

This is achieved by processing the bytecode through 5 distinct processes, which consist of:

1. Transforming the smart-contract's bytecode into a control flow graph through a modified version of Oyente [4].
2. Using a modified version of EthIR [7] to transform said control flow graph into its rule-based representation.
3. Generating size relations for the smart-contract's different state variables from its rule-based representation, using SACO [5].
4. Defining gas equations from said size relations.
5. Solving the gas equations using PUBS [6]

The contract found in Listing A.1 was processed through GASTAP. The output estimations can be found in table Table 1.1

Function Name	Memory Cost	Opcode Cost
`delegate(address)`	9	$656 + max(108, 1 + failed(no_rf))$
`winningProposal()`	15	$1312 \cdot nat(proposals)$
`chairperson()`	26	471
`voters(address)`	24	1496
`vote(uint256)`	24	61587
`winnerName()`	15	$1150 + 1312 \cdot nat(proposals)$
`giveRightToVote(address)`	27	21367
`proposals(uint256)`	18	1028

Table 1.1: Resulting estimations from a sample contract

Most results are simple to interpret, but in some complex cases such as in `delegate(address)` it depends on functions that are not intuitive, nor defined anywhere in the documentation.

Chapter 2

Estimating Gas Costs

This chapter introduces the fundamental concept behind our gas-estimation tool: Fuzz Testing. We then explain the technologies used for the tool's development and describe the design of the fuzzer's architecture. The chapter ends by presenting the designed experiment to evaluate the tool.

2.1. Fuzz Testing

Fuzz testing [13] is a technique used to automatically test software's robustness against different inputs. It consists of automatically executing a program many times with randomly generated inputs in an attempt to find logic or runtime errors. More advanced variants include improved argument-generating processes, like argument mutation or machine learning, that enable the fuzzer to maximize a certain metric like code coverage.

Since fuzz testing can be used to maximize metrics such as code coverage, we applied these techniques to the gas-estimation problem, by attempting to maximize the consumed gas for each function. An initial approach generated a naïve fuzzer that: (i) randomly and uniformly produced values for primitive types and (ii) created transactions that randomly invoke Solidity services.

2.2. Argument Generation

Since Solidity shares many common constructions with any general-purpose programming language such as Java or Python, the fuzzer we designed and implemented for Solidity follows the traditional guidelines to build fuzzers [13]. This included randomly generating primitive values such as integers or characters and using probability distributions to create complex types like byte arrays or strings. We used a geometric distribution in order to generate these size-variable types.

Most primitive types in Solidity are easily fuzzed through traditional techniques like generating a random integer within a given range or choosing a random character from an available dictionary. However, types like addresses need a special procedure in order to pro-

vide valid values. Since transactions are sent from an address in the blockchain, the owner of said address needs to prove their ownership. This is done through public key cryptography, where the address is a representative version of the public key and the secret key is used to sign transactions to authorize them. As such, generating a random 20-byte integer proves insufficient for transaction publication because of the lack of a matching private key. As an additional restriction, most services store their information using addresses as identifiers, for example, storing the amount of ether deposited into the contract by each account, in order to later allow its retrieval.

The designed procedure consists of storing within the fuzzer state containing all known key pairs and whenever a transaction needs to be created, a weighed coin toss determines if: (i) a new address will be returned by generating the corresponding key pair, storing it, and then returning the address related to the generated public key, or (ii) an address related to one of the previous key pairs is used. By reusing previous addresses, we achieve the effect of the same user calling multiple services from a single contract.

When an account is initially generated, it needs a non-0 ether balance in order to pay for gas-costs and potential *payable* services. As it is customary while developing and testing Ethereum applications, a special *faucet address* exists, which has an infinite amount of ether and gives ether to any address that requests it. Using this *faucet*, newly created accounts receive an initial deposit of ether to carry out their transactions.

Whenever a *payable* service is fuzzed, the procedure also needs to determine how much ether to send along with the transaction. The fuzzer achieves this by recording all of the ether balances of the created accounts, and sends with the transaction a random amount of ether, discounted from the chosen account's balance.

Fuzzer procedures for every type in Solidity are documented in Appendix B.

2.3. Transaction Execution

To execute the transactions generated by the fuzzer, an EVM implementation is needed. Many implementations of the EVM are available, however we chose `py-evm` [9] because it passes all standard VM tests and is well documented. Each transaction is then executed as follows:

- A contract is randomly selected.
- A public service within the contract is randomly chosen.
- The arguments for the services and the calling address are generated by the fuzzer.
- The generated transaction is sent to the blockchain for execution, along with around 10^7 wei to pay for any possible gas costs. Wei is a small fraction of ether, the cryptocurrency coin used to express account balances.

After the transaction's execution, the consumed gas cost measured by `py-evm` is recorded, along with the estimation made by the solidity compiler. This data is then used to generate a distribution of gas costs for each service provided by a contract.

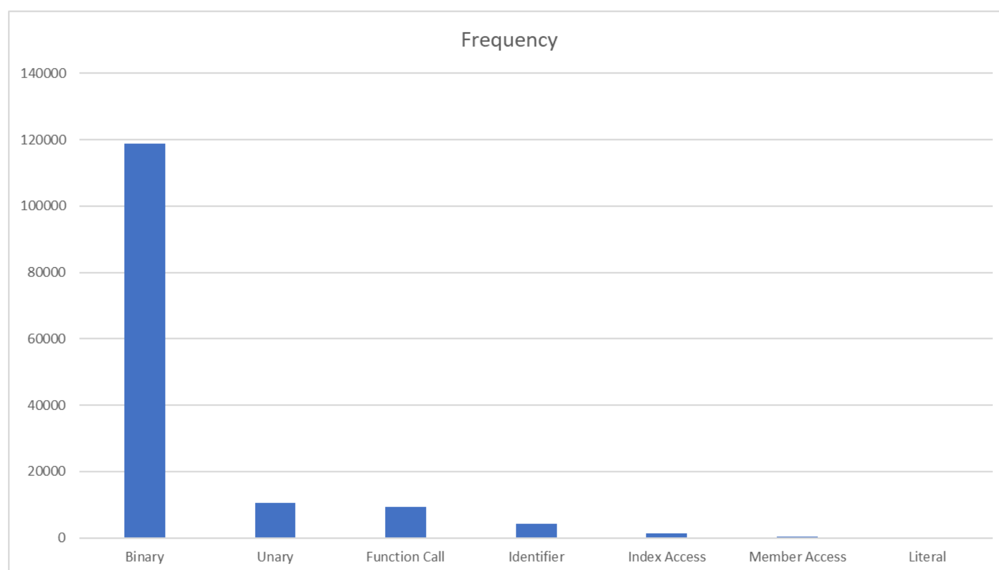

Figure 2.1: Frequency of each kind of operation found within `require` statements

2.4. Symbolic Analysis

Since the fuzzing is done naïvely, it is probable for it to never execute some branches of the source code. In order to avoid this problem, we added symbolic analysis on `require` statements with numerical conditions found within. We conjectured this simple restriction would be enough to help our fuzzer reach further into most functions with these assertions.

Beforehand, we performed some analysis on the abstract syntax trees belonging to a set of 94,878 contracts, in order to corroborate this conjecture. Figure 2.1 reveals that 82 % of operations performed within `require` statements are binary operations. Figure 2.2 shows that 88 % of said operations use operators that apply to numeric arguments. Optimistically, our symbolic analysis could help the fuzzer pass up to 72 % of `require` statements found within the contract. A more detailed breakdown of contract composition can be found in Appendix C.

To implement the symbolic analysis, we first preprocessed the input contract to find all of the contract's private variables. We then inserted new public functions into the source code that returned said variables. This allowed our fuzzer to read the contract's state at run-time, which in turn provided it with most of the information necessary to satisfy the targeted numerical conditions.

During the initial fuzzer setup, each function's body would be parsed to extract every `require` statement and the boolean expression that would satisfy it. These expressions would then be transformed into a set of restraints for each parameter of the function. For example, Listing 2.1 shows a function with a simple restriction, that would cause the generated value for `number` to always be greater than 15. Since only the expression inside the `require` statement is processed, inaccurate constraints for the generated arguments might be generated if any function's parameter is reassigned in its body.

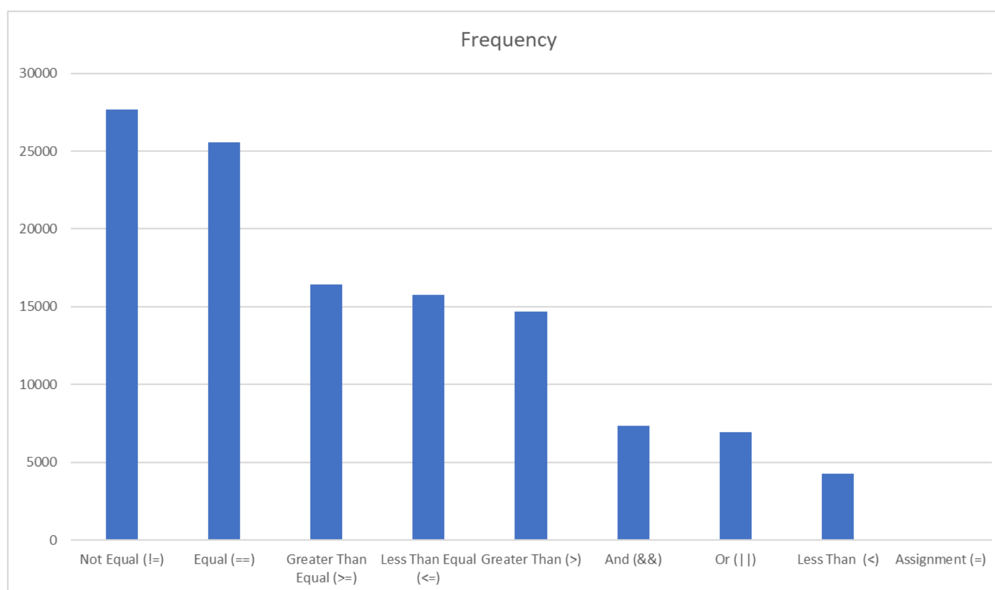

Figure 2.2: Frequency of each operator used within binary operations

Listing 2.1: Simple function that generates a single static constraint for `number`

```
function addToCount(uint8 number) public {
  require(number > 15);
  this.count += number;
}
```

In the case a condition referenced part of the contract's state, we would execute a transaction to obtain the latest value of said state. This function always existed thanks to our preprocessing that created functions for private fields, and public fields generating said functions by default. Listing 2.2 has a function similar to that of Listing 2.1, but the `require` statement forces the value for `number` to always be greater than the value of the contract's state variable `count`.

Listing 2.2: Simple function that generates a single dynamic constraint for `number`

```
  function addToCount(uint8 number) public {
    require(number > this.count);
    this.count += number;
  }
```

It is important to note that we did not add symbolic analysis on `if-then-else` statements because we cannot know on which branch the code would generate a larger gas cost, so choosing a branch to execute would be an arbitrary decision.

2.5. Implementation Design

The implementation divided the fuzzer into three distinct layers, a preprocessing layer, a blockchain layer and a fuzzing layer as illustrated in figure Figure 2.3.

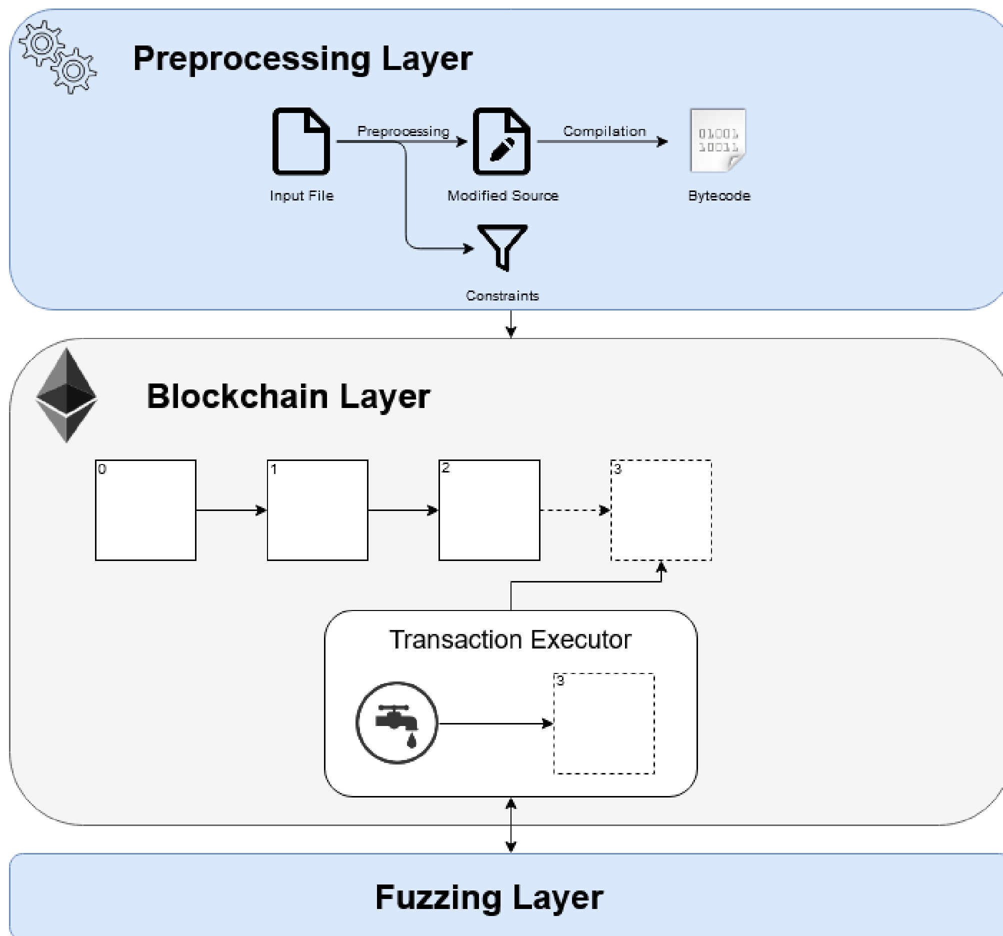

Figure 2.3: Architecture of the developed tool.

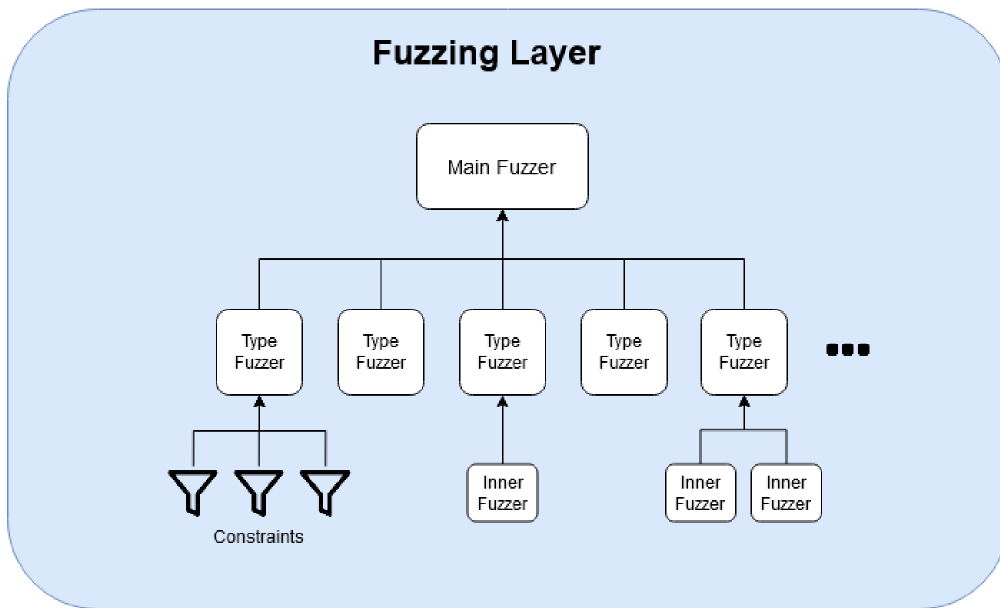

Figure 2.4: Structure used to store fuzzers in the fuzzing layer.

The preprocessing layer is in charge of parsing the input file in search of private fields, for which it would then insert public accessors into the source file in order to make them accesible through the blockchain layer. Afterwards the modified source code is compiled, which resulted in the bytecode that would be uploaded to the blockchain. During the compilation, the modified source code's abstract syntax tree would be parsed to find the `require` statements and transform them into *constraints* for the fuzzer.

The blockchain layer manages the blockchain state. During initialization, it receives the constraints found during parsing in the previous layer, and forwards them to the fuzzer, stating which function they belong to. It creates the faucet address and begins each fuzzing step by choosing a function randomly and requesting its arguments from the fuzzing layer.

The fuzzing layer uses a tree structure (Figure 2.4) to store a set of fuzzers dedicated to every one of a function's arguments. As explained in Appendix B, some fuzzers have child fuzzers which generate the composing sub-types for the parent. For example, the fuzzer for type `uint[]` has a child fuzzer for the type `uint`. Each fuzzer would also store the constraints defined in the preprocessing layer and would use them to attempt to generate values that passed said constraints. Whenever a constraint needs to access the contract's current state, the fuzzer would request the blockchain to execute a transaction that would return the required value. This executed a public accesor function, which is generated during the preprocessing stage for private variables.

Each fuzzer is derived from a base type which managed the execution flow during argument generation, where each derived type specified how to generate its own argument. The final class structure is displayed in Figure 2.5.

The tool was implemented using python and `py-evm` for EVM emulation. The fuzzing and symbolic analysis systems were implemented from scratch. The biggest challenges in these implementations was the symbolic analysis and how constraints would interact with

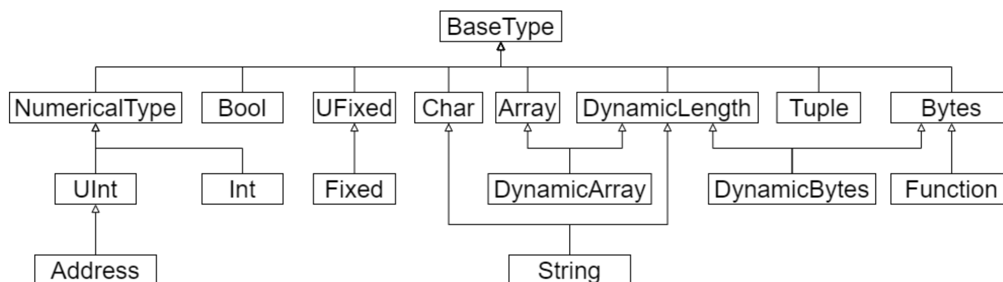

Figure 2.5: Class hierarchy defined by all the implemented fuzzers.

the fuzzing mechanism, as well as keeping the codebase simple.

2.6. Contracts and Experiment Execution

We evaluated our fuzzer on a large number of contracts obtained from `http://EtherScan.io`. We ran our fuzzer on 19,325 Solidity files. A file may contain more than one contract. These files define 94,878 contracts, 233,369 public services, totaling 6,075,210 lines of code. The analyzed contracts were all written in version 0.4 of solidity.

The experiment is designed as follows. We iterate over the Solidity files. For each file, we create a new blockchain. We upload to that new blockchain the bytecode produced by compiling the contracts of the file. Each service is then executed 15 times on average, using the fuzzer without symbolic execution. The whole process is repeated 10 times, to total 150 executions per service, for each file in the dataset.

After the initial run of experiments, the same process is repeated using the fuzzer with symbolic execution to see if the result estimations changed.

Chapter 3

Results

This chapter presents the results obtained from the experiment's execution, and draws conclusions based on the obtained data. Suggestions are made for smart-contract *authors to make solidity code safer for their users.*

3.1. Blueprints

We designed a visual support to summarize the result of our fuzzing algorithm on an individual service. The blueprint is composed of a histogram of the gas cost, represented by the frequency of the service invocation for a given gas cost range. A vertical dashed red line indicates the assessment made by the Solidity compiler using static analysis of the gas cost of the service, and a vertical green line displays the average cost measured throughout the experiment.

Listing 3.1: Source code of the `Essence.buy` service.

```
1  function buy() payable public {
2    uint amount = msg.value / buyPrice;
3    transferFrom(this, msg.sender, amount);
4  }
5  function transferFrom(
6    address _from,
7    address _to,
8    uint256 _value
9  ) returns (bool) {
10     var _allowance = allowed[_from][msg.sender];
11     balances[_to] = balances[_to].add(_value);
12     balances[_from] = balances[_from].sub(_value);
13     allowed[_from][msg.sender] = _allowance.sub(_value);
14     Transfer(_from, _to, _value);
15     return true;
16 }
```

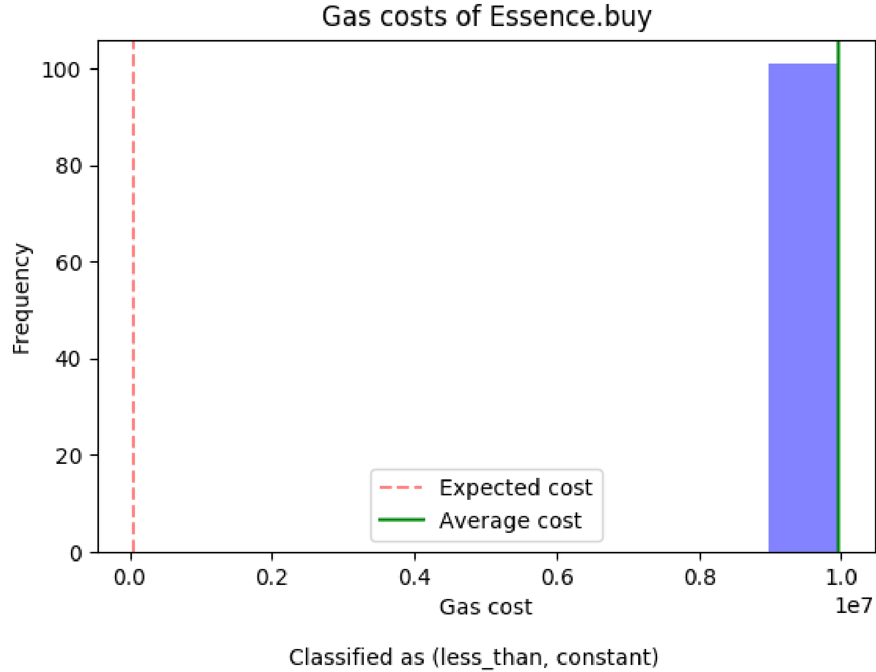

Figure 3.1: Gas cost of a service in the less than, constant, category

Figure 3.1 illustrates the blueprint of a service with a constant consumption during the fuzzing test. However, the expected gas cost, as predicted by Solidity, is significantly lower. This example reflects a runtime error. Having an error at runtime has dramatic consequence, in particular, all gas sent within a transaction is consumed when the transaction fails. Our simulation sends a very large amount of gas with each transaction, thus emphasizing the wrong estimation made by Solidity. The source code of the service illustrated in the figure can be found in Listing 3.1.

The runtime error stems from the fact that the code tries to make a transfer towards an invalid address in the simulation blockchain. The service `transferFrom` is a user-defined wrapper to transfer transaction amount that performs particular checks.

Since the service `buy()` leads to an error if any of the two provided addresses are inadequate, the gas sent along with the transaction is consumed. As a result, this error causes a higher gas cost than what Solidity estimated. Solidity estimates a service cost based on summing the individual gas cost of each bytecode contained in the contract, and does not consider possible errors, such as the one illustrated above.

3.2. Cost trend

The gas consumption histogram may display an overall trend. Consider Figure 3.2. The figure shows three buckets in which the frequency is decreasing for higher costs. Such a trend may be due to the presence (i) of a conditional statement in which our fuzzer is struggling to let the execution go through, or (ii) the service exiting early to avoid runtime errors.

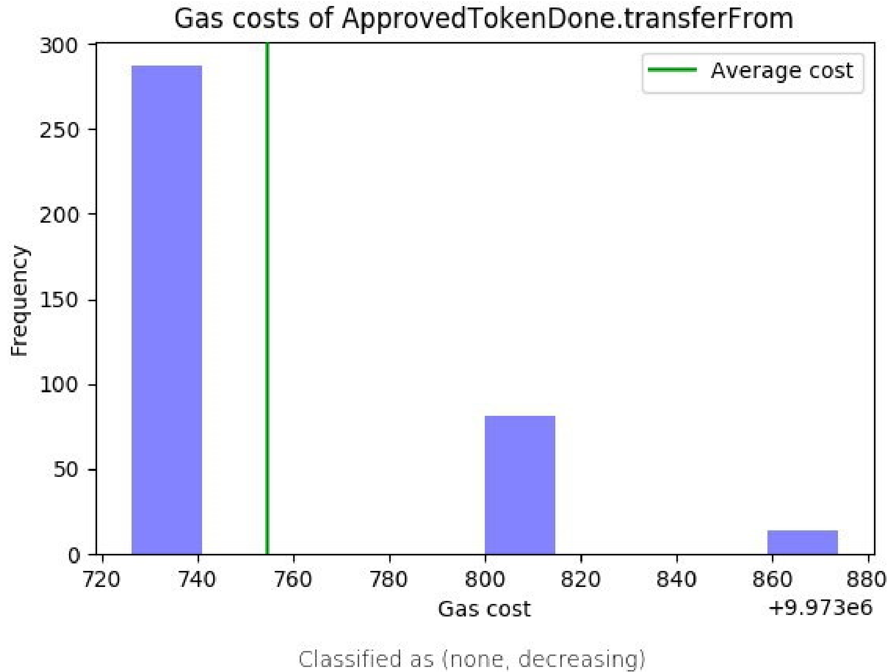

Figure 3.2: Gas cost of a service in the less than, constant, category

The blueprint given in Figure 3.2 does not have the Solidity estimation (*i.e.* the dashed vertical line), this is because Solidity estimated the consumption as *infinite*, indicating a situation for which Solidity cannot determine an upper bound. Such a situation may happen if the contract has to invoke another contract on the blockchain, for which static analysis is not possible.

A cost trend may decrease (*e.g.* Figure 3.2), be stable (*e.g.* Figure 3.1), or increase, and it is a simple, albeit expressive visual support to indicate the amount of branches the code contains and how restrictive their associated conditions are.

3.3. Classification

The cost estimation blueprint displays three distinct aspects of the fuzzing (i) the average gas cost, (ii) the gas cost estimation made by Solidity, and (iii) the general trend in the frequency of gas costs. As such, each blueprint can be classified along two dimensions, the average cost and the cost trend:

- *Average cost*: Comparing the average gas cost obtained during the fuzzing against the Solidity estimation, four situations may happen: (i) the estimation is *less* than the average cost, (ii) it is *equal*, (iii) it is *greater*, and (iv) Solidity cannot statically estimate the cost and merely states that the service may have an *infinite* gas consumption.
- *Cost trend*: The cost trend may belong to one of the following four categories: *decreasing* (greater gas costs have a lower frequency than lower gas costs), *constant*, *increasing* (greater gas costs have a higher frequency than lower gas costs), and *other* (gas costs

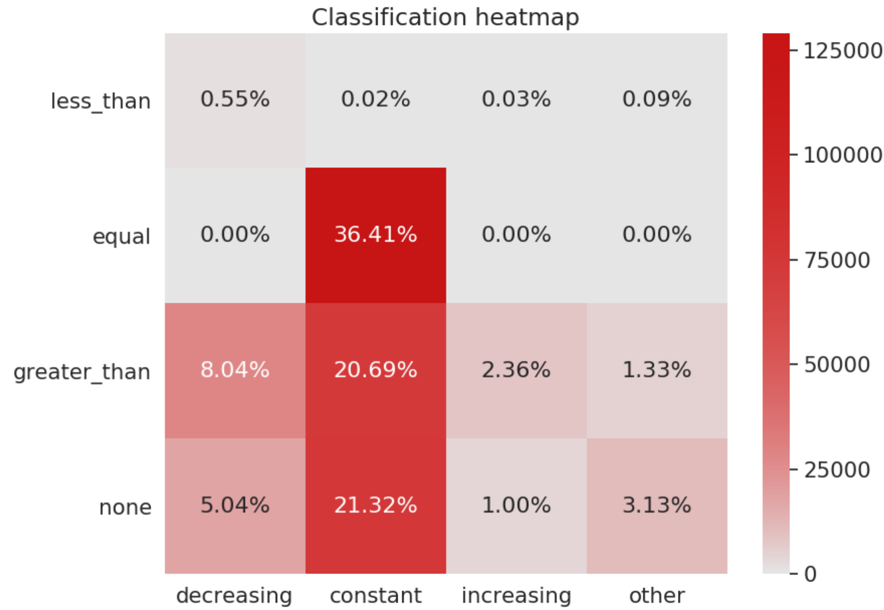

Figure 3.3: Heatmap indicating the obtained classifications

do not establish a monotonic trend).

If for a given function we have only both of these classifications, we can already get a good idea of its general behavior, and a glimpse into its control flow structures.

Manually inspecting the *less* category for the average costs reveals that this category is likely due to runtime errors (for which the gas fee associated with a transaction is lost when an error occurs).

3.4. Quantitative analysis

Our fuzzer has produced 233,369 blueprints (one per public service), and each is classified in the categories described above. Figure 3.3 shows how the blueprints are distributed across the categories.

Our results indicate that only **36.12 % of the public services have a gas cost equal to the estimate made by Solidity from a static analysis**. A manual inspection on a representative sample of the source codes indicates that services with a constant cost are mostly variable accessors or calls to external services and operations which might result in an error. Note that these methods are often automatically generated by the Solidity compiler. Service accessors for fields having a dynamically-sized type, such as maps, strings, and dynamic arrays, have an infinite gas cost.

The *none* category, defined by services whose estimation was infinity, are mostly composed of services where one of the following happens:

1. A function call to an external contract is made;
2. A transfer is made;
3. A dynamically-sized field is returned;
4. The code contains a loop whose length depends on the state of the contract.

The *decreasing* case of the *none* category mostly corresponds to cases (3) and (4). This is caused by a bias in how the fuzzer generates dynamically sized fields, where the value generator uses a geometric distribution to determine the length of the sequences, hence the tendency for such functions to be classified as *decreasing*.

The *greater than* category is made up mostly of services that have some kind of error checking with *if* statements, and simply returning if some condition is not true. These services naturally have an average cost which is lower than the estimation, since they are called with essentially random arguments. As such there is a very low probability that the execution would pass through the error checking statements. Because Solidity gives an upper bound for the expected gas cost, then the actual gas cost is lower than the estimation because the execution took a shorter branch of the code, mostly exiting early to avoid errors. Listing 3.2 displays an example of code that generates this behavior. Figure 3.4 contains the blueprint generated for this service.

Listing 3.2: Function sample that is classified in the *greater than* category

```
function transferOwnership(address _newOwner) public ownerOnly {
  if(owner != _newOwner) {
    owner = _newOwner;
  }
}
```

Decreasing, increasing and other trends are tightly tied to the *greater than* and *none* classifications. Decreasing and increasing trends generally check for errors in their code, but their trend is defined by how restrictive this check is. In increasing trends, most of the executions manage to pass through the `require` statements. In decreasing trends, the opposite happens. The *other* classification is also closely related to them since both definitions take strictly decreasing or increasing frequencies. It takes only a single outlier to break the pattern and make the function be classified as *other*.

We repeated the same experiment after adding symbolic execution for numeric `require` statements to our fuzzer. Figure 3.5 illustrates the results of said experiments. No statistically significant remark can be made by comparing them to the original experiment, other than the equal constant category having a larger frequency after applying symbolic execution. This might be due to the fact that less runtime errors were encountered, and so the solidity compiler did not underestimate in the said cases.

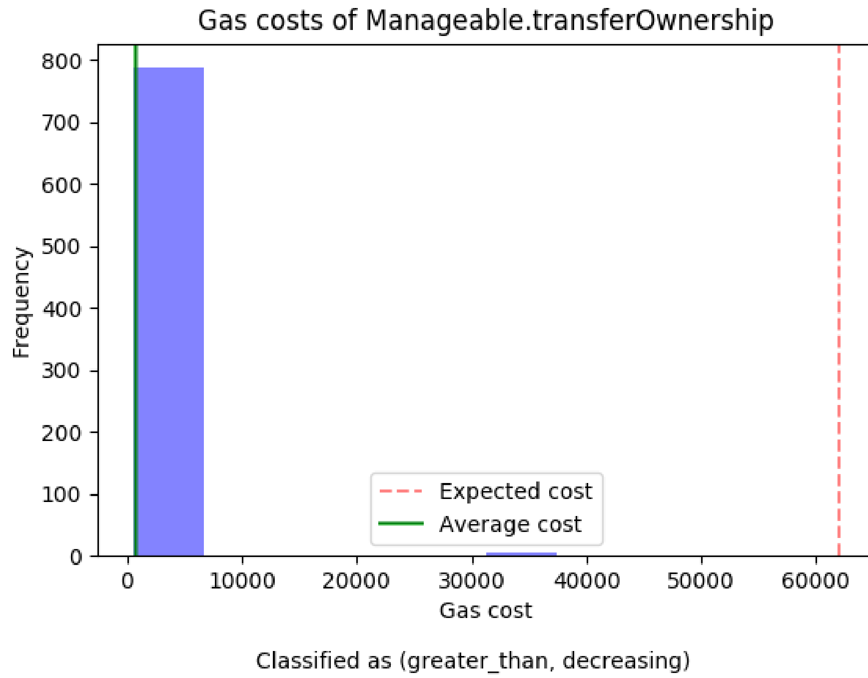

Figure 3.4: Gas cost of the service `transferOwnership`.

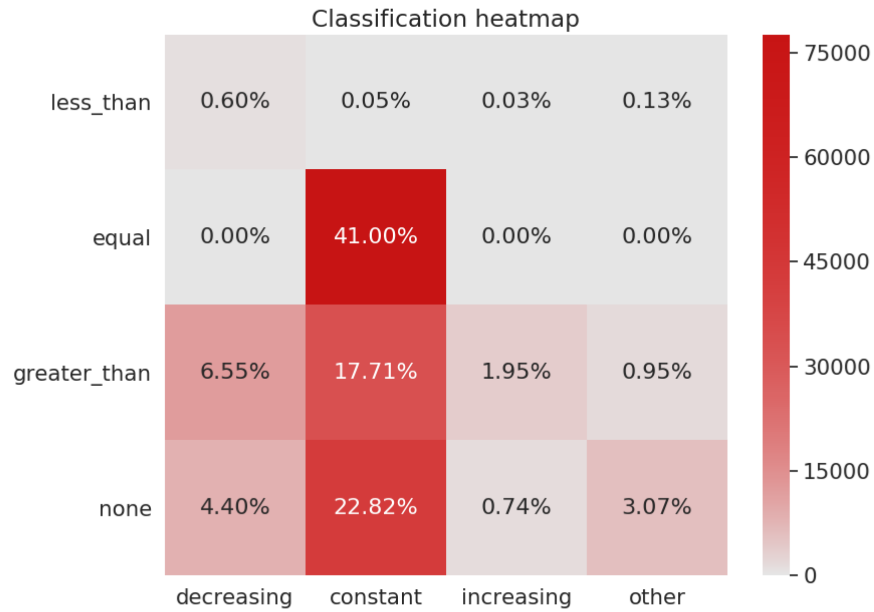

Figure 3.5: Heatmap displaying the results of the experiment with symbolic analysis.

Conclusion

We have developed a fuzz testing tool that allows one to classify each service of a contract using two criteria: performance against the estimation made by Solidity, and the trend of the gas cost exhibited during the testing. Each classification offers us some insight about what the associated code might do, including the restrictiveness of the error checking, possible optimizations and encountered runtime errors. Note that we designed our testing tool for ease of use, as such, no configuration is necessary to employ it.

Our fuzzer gives a pragmatic approach to gas estimations, by executing the code multiple times and checking how much each transaction cost. The estimates however are not as useful as we would like, because the tool does not take into account what kind of arguments are expected by the services. Since the fuzzing is done completely at random, it might violate some implicit contract expected by the service. This might explain the large amounts of services that exit during error checks.

Adding basic symbolic execution on require statements with numeric conditions did not generate any statistically significant difference in our results, so a more sophisticated alternative might be necessary for detecting better gas limits and trends.

Future work will explore wether improving the fuzzer with more advanced techniques allows for faster and better estimations, and generating better metrics to ease comparison between the fuzzer and other tools for gas estimation.

We also make a simple recommendation for safer solidity code. Early returns with *if* statements should be chosen over runtime errors caused by *require* calls, in order to minimize lost gas.

All the work developed throughout this thesis was compiled into a paper [11], which was presented at ICSME - NIER 2020.

Bibliography

[1] Augur homepage. `https://www.augur.net/`. Accessed: 2020-07-11.

[2] Gods unchained homepage. `https://godsunchained.com/`. Accessed: 2020-07-11.

[3] Golem homepage. `https://golem.network/`. Accessed: 2020-07-11.

[4] Oyente: An Analysis Tool for Smart Contracts. `https://github.com/melonproject/oyente`.

[5] 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.

[6] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis*, pages 221–237, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. *CoRR*, abs/1805.07208, 2018.

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

[9] Ethereum Foundation. A python evm implementarion. `https://github.com/ethereum/py-evm`.

[10] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[11] Daniel Soto, Alexandre Bergel, and Alejandro Hevia. Fuzzing to estimate gas costs of ethereum contracts. In *2020 the 36th International Conference on Software Maintenance and Evolution – New Ideas and Emerging Results (ICSME-NIER)*, pages 1–5, September 2020. To appear.

[12] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger revision

(4ea7b96 – 2020-06-08), 2020. Accessed: 2020-07-12.

[13] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The Fuzzing Book. In *The Fuzzing Book*. Saarland University, 2019. Retrieved 2019-09-09 16:42:54+02:00.

Appendix A

Example Ballot Contract

Listing A.1: An example Ballot contract provided by the Solidity Documentation

```
pragma solidity ^0.4.22;

/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;

    // This declares a state variable that
    // stores a "Voter" struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically-sized array of "Proposal" structs.
    Proposal[] public proposals;

    /// Create a new ballot to choose one of "proposalNames".
    constructor(bytes32[] proposalNames) public {
```

```
32            chairperson = msg.sender;
33            voters[chairperson].weight = 1;
34
35            // For each of the provided proposal names,
36            // create a new proposal object and add it
37            // to the end of the array.
38            for (uint i = 0; i < proposalNames.length; i++) {
39                // "Proposal({...})" creates a temporary
40                // Proposal object and "proposals.push(...)"
41                // appends it to the end of "proposals".
42                proposals.push(Proposal({
43                    name: proposalNames[i],
44                    voteCount: 0
45                }));
46            }
47        }
48
49        // Give "voter" the right to vote on this ballot.
50        // May only be called by "chairperson".
51        function giveRightToVote(address voter) public {
52            // If the first argument of "require" evaluates
53            // to "false", execution terminates and all
54            // changes to the state and to Ether balances
55            // are reverted.
56            // This used to consume all gas in old EVM versions, but
57            // not anymore.
58            // It is often a good idea to use "require" to check if
59            // functions are called correctly.
60            // As a second argument, you can also provide an
61            // explanation about what went wrong.
62            require(
63                msg.sender == chairperson,
64                "Only chairperson can give right to vote."
65            );
66            require(
67                !voters[voter].voted,
68                "The voter already voted."
69            );
70            require(voters[voter].weight == 0);
71            voters[voter].weight = 1;
72        }
73
74        /// Delegate your vote to the voter "to".
75        function delegate(address to) public {
76            // assigns reference
77            Voter storage sender = voters[msg.sender];
78            require(!sender.voted, "You already voted.");
```

```
79
80         require(to != msg.sender, "Self-delegation is disallowed.");
81
82         // Forward the delegation as long as
83         // "to" also delegated.
84         // In general, such loops are very dangerous,
85         // because if they run too long, they might
86         // need more gas than is available in a block.
87         // In this case, the delegation will not be executed,
88         // but in other situations, such loops might
89         // cause a contract to get "stuck" completely.
90         while (voters[to].delegate != address(0)) {
91             to = voters[to].delegate;
92
93             // We found a loop in the delegation, not allowed.
94             require(to != msg.sender, "Found loop in delegation.");
95         }
96
97         // Since "sender" is a reference, this
98         // modifies "voters[msg.sender].voted"
99         sender.voted = true;
100        sender.delegate = to;
101        Voter storage delegate_ = voters[to];
102        if (delegate_.voted) {
103            // If the delegate already voted,
104            // directly add to the number of votes
105            proposals[delegate_.vote].voteCount += sender.weight;
106        } else {
107            // If the delegate did not vote yet,
108            // add to her weight.
109            delegate_.weight += sender.weight;
110        }
111    }
112
113    /// Give your vote (including votes delegated to you)
114    /// to proposal "proposals[proposal].name".
115    function vote(uint proposal) public {
116        Voter storage sender = voters[msg.sender];
117        require(!sender.voted, "Already voted.");
118        sender.voted = true;
119        sender.vote = proposal;
120
121        // If "proposal" is out of the range of the array,
122        // this will throw automatically and revert all
123        // changes.
124        proposals[proposal].voteCount += sender.weight;
125    }
```

```
126
127         /// @dev Computes the winning proposal taking all
128         /// previous votes into account.
129         function winningProposal() public view
130                 returns (uint winningProposal_)
131         {
132             uint winningVoteCount = 0;
133             for (uint p = 0; p < proposals.length; p++) {
134                 if (proposals[p].voteCount > winningVoteCount) {
135                     winningVoteCount = proposals[p].voteCount;
136                     winningProposal_ = p;
137                 }
138             }
139         }
140
141         // Calls winningProposal() function to get the index
142         // of the winner contained in the proposals array and then
143         // returns the name of the winner
144         function winnerName() public view
145                 returns (bytes32 winnerName_)
146         {
147             winnerName_ = proposals[winningProposal()].name;
148         }
149     }
150
```

Appendix B

Fuzzer Procedures for Solidity Types

B.1. Fixed Size Types

B.1.1. Booleans

Boolean values are the most straightforward to generate. The implementation chooses between true and false with a 50 % chance for each.

B.1.2. Integers

In Solidity, integers come in two main variants: `int` and `uint`, both 256 bits long. Each of these types can be further typed to describe how many bits it should use. Valid bit-sizes are multiples of 8, and go from 8 to 256 bits. For example, the type `uint64` is a 64-bit unsigned integer, with values ranging from 0 to $2^{64} - 1$. The implementation determines the minimum and maximum values for the type, and then returns a value inbetween the limits.

In summary:

- Given the type `uintN` a value is generated within the range $\left[0, 2^N\right[$.
- Given the type `intN` a value is generated within the range $\left[-2^{N-1}, 2^{N-1}\right[$.

B.1.3. Addresses

Addresses have two different generation procedures, depending on what it will be used for. If the address is to be received by a service as an argument a `uint160` fuzzer is used to generate a random value. If the address is needed for transaction creation (*i.e.,* the address of the transaction's author), then the procedure described in section 2.2 is followed.

B.1.4. Fixed-Point Numbers

In Solidity, fixed-point numbers are declared as `fixedNxM` or `ufixedNxM`, where N is the number of bits used by the number, and M is the number of decimal points available. M

must be divisible by 8, and at most 256 bits. N must be an integer in the range $[0, 80]$. If not specified, the values for N and M are 128 and 18 repectively.

The implementation is as follows:

- Given the type `ufixedNxM` a value v is generated within the range $\left[0, 2^N\right[$.
- Given the type `ufixedNxM` a value v is generated within the range $\left[-2^{N-1}, 2^{N-1}\right[$.

The fuzzer then returns $v/10^M$.

B.1.5. Characters

The implementation chooses a random character from a whitelisted set, composed of all digits, all ASCII letters, all punctuation marks and whitespace.

B.1.6. Fixed-size Byte Arrays

Solidity includes types for byte arrays with a fixed size. These are declared as `bytesN` , where N is the number of bytes to include in the array, and ranges from 0 to 32. The implementation is straightforward, and generates N times 8 random bits, which are later inserted into a byte array.

B.1.7. Function Pointers

Function pointers are equivalent to a 20-byte array, and as such uses the `bytes20` fuzzer. Even though they do not point to a real function's pointer this did not prove to be a problem, since it is a rarely used feature in Solidity.

B.1.8. Fixed-size Arrays

Arrays can be declared as `T[N]`, where `T` is the array's underlying type, and N is the amount of elements in the array. The implementation simply uses the underlying fuzzer's type to generate N values, and returns an array containing said values.

B.2. Dynamic Size Types

These types include arrays without a defined size (declared as `T[]`), arrays of bytes without a defined size (`bytes`), strings and tuples. All array-based types use the underlying type's fuzzer a variable amount of times n. This value is obtained from a geometric distribution, so it tends to have a bias for shorter arrays.

Finally tuples are declared as a comma-separated list of internal types inside parentheses. (*e.g.,* `(uint64, bytes16, bool)`). This fuzzer uses each of the internal types' fuzzers to generate the final value.

Appendix C

Contract Composition

During the development of the symbolic analysis for our fuzzer, we perfomed analysis on our database of 19,089 solidity files, defining in total 94,878 unique contracts.

Figure C.1 illustrates the versions of solidity with which the contracts were written.

Within these contracts, 480,276 unique functions were defined. Overall, 37 % of contracts contained `require` statements, and 15 % of functions contained at least one `require` statement.

On average, each file contained 7,58 contracts, each contract defined 1,53 functions, and each function had 0,30 `require` statements. Ignoring contracts and functions without `require` statements, each contract defined 4,13 functions, and each function contained 2,06 `require` statements.

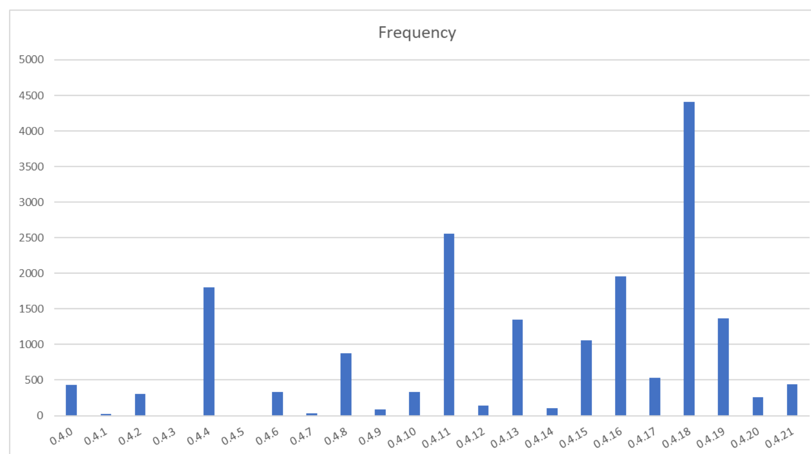

Figure C.1: Frequency of solidity versions used in the contract database.