Analysis techniques for smart contracts: generation of complete control flow graphs

Técnicas de análisis para contratos inteligentes: generación de grafos de control de flujo completos

Universidad Complutense de Madrid
Facultad de Informática

Trabajo de fin de grado del Doble Grado en Ingeniería Informática y Matemáticas

Alejandro Hernández Cerezo

Dirigido por:
Elvira Albert Albiol

Curso 2019-2020

Abstract

Ethereum is the most popular blockchain. It has become really well-known in the last few years because it lets users deploy their smart contracts on top of it. Gas is used to measure the computational effort when executing a transaction and to reward miners. Users set the gas limit when proposing a transaction, and if the miner runs out of gas before performing it, an out-of-gas exception is raised, reverting to the previous state before execution. Thus, inferring gas consumption is really important for not losing resources.

Besides, some exploits have been found that have led to major economic losses, due to subtle bugs in the code. An example of it is the famous *DAO attack*.

In order to tackle the efficiency and soundness problems mentioned above, we have to rely on formal methods that guarantee the soundness and accuracy of possible analysis. Research has been done previously in this topic, and it has led to the creation of tools that analyze different features on Ethereum Virtual Machine(EVM) code. Among them, Gastap is one of the few tools based on static analysis that manages to infer gas upper bounds for transactions. Gastap is one of the most accurate tools in the field, having a great success rate. It generates a Control-Flow-Graph (CFG) as an intermediate representation of the analysis. However, the current algorithm used by Gastap is not precise. Therefore, a considerable number of smart contracts cannot be analyzed.

This dissertation proposes a new algorithm for generating a Complete CFG from an EVM smart contract. We will prove that this algorithm is sound, and prove completeness is lost only in certain cases.

It greatly improves the performance from the previous version. Experiments corroborate this fact: we have analyzed a total of 10,736 files, generating a CFG from roughly 90%, in contrast with the 80% of the contracts that could be analyzed before. From the 10% remaining, only less than 1% of the contracts still fail due to our analysis. Besides, we achieve a great efficiency: CFG generation time takes less than 0,01% of the total time for the analysis.

Another key feature of the proposed algorithm is that it can be easily implemented and adapted to other stack-based programs.

Keywords:

Blockchain, Control-Flow Graph, Data-flow analysis, Ethereum, Ethereum Virtual Machine, Formal Methods, Gas, Operational Semantics, Smart Contracts, Symbolic Execution .

Resumen

Ethereum es la cadena de bloques más popular. Se ha convertido en una plataforma muy conocida en los últimos años, permitiendo a los usuarios desplegar sus dApps en ella. Ethereum utiliza el llamado *Gas* para medir el coste computacional de ejecutar una transacción y así recompensar a los mineros. Para ello, los usuarios tienen que fijar un límite de gas a la hora de proponer una transacción, de tal forma que si se agota durante la ejecución, se genera una excepción, y se revierte al estado previo a la ejecución. Es por ello que determinar el consumo de gas es muy importante para no malgastar recursos.

Además, se han encontrado algunas vulnerabilidades que pueden conducir a grandes pérdidas económicas. Un ejemplo de ello es el conocido *DAO attack*.

Para abordar las situaciones de eficiencia y corrección descritas anteriormente, tenemos que recurrir a métodos formales que nos garanticen la precisión de posibles análisis. Se han llevado a cabo diversas investigaciones en este campo, que han llevado a la creación de distintas herramientas para el análisis de propiedades de los contratos inteligentes. Entre ellas, destaca Gastap, una herramienta que permite inferir cotas superiores del consumo de gas de transacciones. Es una de las herramientas con mayor porcentaje de contratos analizados con éxito. Para ello, genera un Grafo de control de flujo (CFG) en una de las etapas intermedias del análisis. Sin embargo, el algoritmo empleado actualmente no es preciso, por lo que hay un número considerable de contratos que no son analizados correctamente.

Este trabajo propone un nuevo algoritmo para generar un Grafo de control de flujo completo de un contrato inteligente. Probaremos que este análisis es correcto, y que perdemos información únicamente en ciertos casos.

Los resultados muestran como el nuevo algoritmo mejora bastante con respecto al anterior: hemos analizado un total de 10,736 archivos, de los que podemos generar su CFG para casi un 90%. Con el análisis anterior, éramos capaces de generar un 80% de los casos. Del 10% de contratos restantes, menos del 1% fallan por culpa de nuestro algoritmo. Además, este algoritmo es muy eficiente, pues supone menos de un 0,01% del tiempo de ejecución del análisis completo.

Otro aspecto relevante de este algoritmo es que es fácilmente implementable, y que puede ser adaptado a otros lenguajes basados en la pila.

Palabras Clave:

Análisis de Flujo de Datos, Cadena de bloques, Contratos inteligentes, Ejecución Simbólica, Ethereum, Ethereum Virtual Machine, Gas, Grafo de Control de Flujo, Métodos Formales, Semántica Operacional .

Agradecimientos

Es muy difícil incluir en tan sólo unas líneas toda la gente que ha contribuido de forma directa o indirecta a que haya podido concluir con este trabajo, así que voy a repartir mis agradecimientos entre los dos trabajos de fin de grado que me tocan. Algo bueno había que sacar de hacer dos TFGs en vez de uno...

En primer lugar, dar las gracias a mi directora Elvira, por apostar por mí a la hora de realizar este proyecto, y por guiarme a lo largo de este proceso, sugiriendo ideas sobre como abordar el tema, resolviendo todas mis dudas y dándome pautas que han hecho de este trabajo sea mucho más asumible de lo que podría haber imaginado en un primer momento. Gracias también por el apoyo que me has ido brindando todos estos meses.

Otro agradecimiento enorme va para Pablo Gordillo. Gracias por ayudarme tanto a entender el trabajo previo, por ofrecerte desinteresadamente a reunirnos en persona para comentar y resolver detalles de la implementación y resolver todas mis dudas puntuales en cualquier momento. Sin duda, aunque oficialmente no figure así, has sido mi otro director de TFG.

Este trabajo supone la culminación de cinco años de mi vida, llenos de muchos momentos duros y horas intensas de trabajo. No me cansaré de repetirlo, si he llegado a este punto, es por mis maravillosos compañeros de carrera. Uno esperaría que en una carrera tan exigente para entrar como es el doble grado existiese una competitividad muy alta. Nada más lejos de la realidad, en todo momento ha existido una colaboración entre todos nosotros, conscientes de que todos nos hallábamos en la misma situación, y que por ende, la mejor forma de afrontarla era unir fuerzas. En muchos casos, me llevo un vínculo mucho más fuerte que compañeros. Ojalá nuestros caminos se crucen en algún futuro no muy lejano.

Y bueno, sin duda lo mejor de la carrera ha sido conocer a un grupo de gente tan genial a los que tengo la suerte de llamar amigos. Gracias a Enrique, Cristina, Rubén, Rafa, Ming, Rivas y Guille por todos esos momentos juntos de diversión, esos viernes por la tarde en la biblioteca hasta las 9, esos grupos de apoyo de trabajo intensivo... Sin vosotros, no habría podido sacar todo esto adelante.

Gracias papá y mamá por confiar en mi y darme la oportunidad de poder estudiar fuera, haciéndome la vida más fácil posible y apoyándome en todo momento. Y en general, por hacerme la vida más fácil siempre, por apoyarme en mis mejores y peores momentos, por inculcarme el valor del trabajo duro, por estar siempre ahí cuando lo necesitaba.

Gracias a ti Mari Loli, porque sin duda eres tú has sido mi compañera de viaje a lo largo de este período. Hemos compartido todos los buenos y malos momentos juntos, haciendo que nuestra relación crezca y crezca día a día.

Y por último, gracias a ti lector. Si has llegado a este punto después de toda esta parrafada, es que verdaderamente te importo y mereces quedar reflejado en los agradecimientos.

Contents

Abstract iii

Resumen v

Agradecimientos vii

Contents x

List of Figures xi

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Contributions . . . 6

2 Foundations 9
- 2.1 Data-flow analysis . . . 9
- 2.2 Solidity . . . 11

3 Ethereum Virtual Machine 13
- 3.1 Overview . . . 13
- 3.2 Gas . . . 14
- 3.3 EVM Semantics . . . 15
  - 3.3.1 A simplified EVM semantic for analysis purposes . . . 19

4 Analysis 23
- 4.1 The transfer function . . . 23
- 4.2 The constraint equation system . . . 25
- 4.3 Control Flow Graph . . . 28

5 Soundness 31

6 Completeness 35

7 Implementation 39
- 7.1 Gastap architecture . . . 39
- 7.2 Oyente* implementation . . . 41
- 7.3 Gas Bounds for Sums Case Study . . . 43

8 Evaluation **47**
8.1 Experiments generation 47
8.2 Oyente* statistics 48
8.3 Gastap Statistics 49

9 Conclusions and Future Work **51**

Bibliography **54**

List of Figures

1.1 Example of CFG showing how ambiguous paths can be found 3
1.2 Comparative between a program CFG, and its cloned version 5

3.1 Solidity code for `Sum` example contract and its associated CFG 17
3.2 Simplified EVM semantics for handling jumps 20

4.1 Updating function 24
4.2 Jumping address system equations 25

7.1 Gastap Architecture 39
7.2 Gas bounds for `Sums`. Function `nat` defined as `nat(l) = max(0,l)`. 43
7.3 Solidity code for `sums` example contract 45

8.1 (Top) Comparison between old and new analysis results when using EthIR. (Bottom) Reason breakdown for Oyente* errors. 48
8.2 (Top) Statistics of gas usage on the analyzed 34,460 smart contracts from Ethereum blockchain. (Bottom) Timing breakdown for Gastap on the analyzed 34,460 smart contracts. 50

Chapter 1

Introduction

1.1 Context

The Ethereum Virtual Machine (EVM) is a stack machine that allows the execution of smart contracts in the Ethereum blockchain. It executes EVM bytecode, the low-level language to which high-level instructions compiles to.

The aim of this project is to improve the accuracy of an already existing tool: EthIR [1]. EthIR is a framework that generates a high-level rule-based representation (RBR) from low-level EVM bytecode. This representation is crucial for other tools to infer properties related to the EVM code, such as Gastap [2] or SAFEVM [3].

Gastap is a tool that infers parametric gas bounds from an EVM contract. This upper bound can depend on the sizes of the input parameters of the functions, the contract state, and/or the blockchain data that the gas consumption depends upon.

EthIR relies on Oyente*, an intermediate tool contained in EthIR that generates a control flow graph (CFG) of the program. This tool is built on top of Oyente [4], another tool that generates a CFG and analyzes smart contracts so as to find possible vulnerabilities. However, this analysis is not sound nor complete, as it has a fixed number of iterations to generate the information from the contract. It also skips some of the information needed for generating the CFG in some cases.

Oyente* has removed these constraints as an attempt to generate a complete CFG, and has made other changes to ensure needed data is obtained. However, current analysis introduces unfeasible paths in our graph, leading to inaccuracies in Gastap analysis.

This document will propose a new sound analysis, taking into account the particularities of the EVM architecture. Completeness cannot be achieved, as EVM bytecodes may contain persistent information that cannot be resolved in a static analysis. Nevertheless, we will also prove that we lose the completeness only in certain cases.

Definitions, analysis and results from this dissertation are planned to be published in an international forum. Currently, we have published an arxiv paper [5]. Most of the definitions and some results are the same in both this project and the report. Nevertheless, the analysis proposed in this dissertation is original, and has been improved through the contributions of all authors in the paper. My main contributions are the implementation of the algorithm, the development of the proofs for soundness and completeness, improvements in some of the definitions to perform the analysis and the proposed jump semantics. I have also contributed to the design of experiments for the new version of Gastap, creating the bash files to carry out experiments and controlling everything went smoothly. New examples have been added in this document, to contribute to a broader

understanding on how the algorithm works. Besides, a new case study for GASTAP has been added, in order to highlight the strengths and limitations this tool has.

We are aware that the reader may not be familiarized with some of the concepts mentioned above. An explanation of all of them will be provided in Chapter 3.

1.2 Motivation

Data-flow analysis is one of the main techniques used by compilers to optimize a program or to run static analysis. It consists in obtaining the possible values memory may have at different points of the program.

One way to construct this data-flow is by obtaining the control-flow graph (CFG) from a program. A control-flow graph is a directed graph that represents the flow of a program during its execution: paths found in a CFG correspond to possible flows.

Nodes represent the basic blocks of the program, *i.e.* subsets of instructions that are *always* executed uninterruptedly and with no jump instructions within it. Edges represent the possible jumps between different basic blocks. For the sake of brevity, from this point we will be denoting basic blocks as simply blocks.

In many different low-level languages, computing a CFG is really easy. Many of these languages specify directly the target of branch instructions, and therefore, nodes and edges can be easily identified:

- The first instruction of a block is either the initial instruction of the program, the target of a conditional or non-conditional jump instruction, or the instruction that follows a branch instruction.
- The final instruction of a block can be the *Jump* instruction itself, an exception instruction or the last instruction of the program.

Edges are constructed by joining each node containing a jump instruction with another block that starts with the target address from that jump.

We know that this description may not cover all different cases for blocks in many languages, but we want to emphasize that knowing statically the target of branch instructions makes the generation of CFG trivial.

However, there are other mechanisms to handle jumps. In particular, in some languages, the address of a branch instruction is determined in the moment of the execution. For instance, some stack-based languages can store beforehand the values of jumps in the stack, and once a branch instruction is executed, they retrieve the address from the top of it.

In these cases, generating a CFG becomes really tricky. Now we cannot keep track of the possible jump values. Thus, there is no *direct* way to determine how nodes are connected in the graph.

A possible solution involves executing the program to obtain these values, annotate all the possible jump addresses a block can jump to, and add an edge for each of these possibilities. However, this approach introduces several unexpected problems we need to cope with.

The first problem has to do with those languages in which dynamic values that cannot be inferred through static analysis are involved. For instance, we can find this problem when accessing a memory that contains persistent information that was stored before considering the static analysis.

We didn't have this problem with static jumps, as these instructions didn't have any impact in the generation of the CFG. However, now branch instructions can depend on them. So if we simply tried to execute the program, we could reach a point in which a value is needed in order to resolve a jump instruction, but we have no way to obtain it. At this point, the execution would fail.

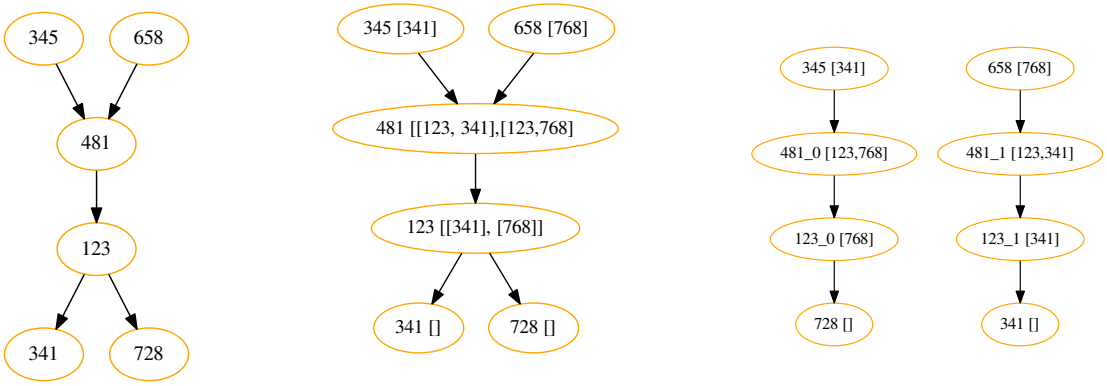
This problem can be overcome using symbolic execution. This way, instead of executing the code *per se*, we contemplate the possibility that certain values may be undefined, and we explore every possibility once we need to manage them. The analysis would still fail if target address is unknown though. Nevertheless, there are other situations in which analysis would normally halt, but now we can deal with: conditional jumps that rely on dynamic values in order to decide whether to take them or not. In this case, we would explore both possibilities.

This approach has also a big downsize. Exploring every possibility also leads to inadvertently introduce program flows that cannot appear through regular execution. Then we would introduce more edges than expected in our CFG, and some of the paths in our graph would represent unfeasible flows. At this point, we lose accuracy in the analysis.

Nevertheless, the same problem can be found in other languages that include conditional branch instructions. In both cases, it is really difficult to overcome this problem, as determining these dynamic values becomes totally impossible in practice. A possible solution would involve using a SMT-Solver to try to infer whether a condition in a jump instruction is always true or false and avoid exploring unfeasible conditions. However, this wouldn't necessarily cover all the cases.

The second problem has to do with the paths we can find in a CFG. Static branch instructions guarantee that any path found from an entry point to a terminating node is feasible: all edges correspond to fixed jumps, and therefore it doesn't matter the execution of instructions in the node, because jump addresses will always be the same. We know that this is not true in conditional jumps, so we will omit these cases for the rest of this section.

The reasoning above doesn't hold anymore with a stack. We can see in the following example how non-existing paths can be found when obtaining paths in the CFG:



(a) Initial CFG

(b) Initial CFG with annotated stacks

(c) Cloned CFG

Figure 1.1: Example of CFG showing how ambiguous paths can be found

Example 1.1. *In Figure 1.1a , we can see a certain flow in a CFG. We will assume there is an entry point connected to blocks* `345` *and* `658`*, but will be omitted for the sake of making the example clearer. We can find four different paths within the graph:*

- *345* $\mapsto$ *481* $\mapsto$ *123* $\mapsto$ *341*
- *658* $\mapsto$ *481* $\mapsto$ *123* $\mapsto$ *341*
- *345* $\mapsto$ *481* $\mapsto$ *123* $\mapsto$ *728*
- *658* $\mapsto$ *481* $\mapsto$ *123* $\mapsto$ *728*

If we include all the possible stack values at the beginning of those nodes, we have the following situation, depicted in Figure 1.1b

It is clear that from the paths discovered above, two of them are feasible and two of them don't represent real paths.

The way we are addressing is by cloning those nodes that are common to different paths, but can lead to ambiguous flows appearing. Cloning means that we duplicate common nodes, one per possible feasible path. This way, we ensure different paths cannot have common nodes, and we prevent ambiguous paths to appear. These nodes share the same instructions, but from the perspective of the graph, they are totally independent.

In the example above, there are two paths that share nodes `481` *and* `123`*. By cloning them, we obtain the graph shown in Figure 1.1c.*

It is easily followed that only two paths can be found, which corresponds to those that are feasible.

Now we may wonder how to perform the cloning. A naive approach could consist in generating the CFG without addressing this problem, and make a copy of each block per shared path. Then, we could link an edge for each copy to a copy of the next block in the original graph. In simple cases like the example above, this algorithm seems to work fine.

However, if we have cycles in the initial CFG, it isn't really clear how to perform the analysis. Next example illustrates a situation in which we make more copies than expected.

Example 1.2. *Figure 1.2 shows a CFG from a real contract if the cloning problem isn't considered and its version with our proposed algorithm. Note that we cannot just clone the node* `361` *and join it to each of its matching nodes. As we can see in the cloned version, there are 5 copies of this node, while we would expect just 3 different copies to appear.*

This example aims to show why cloning cannot just be performed by duplicating each node for each edge that is pointed. Instead, we need to keep track of the stack during the symbolic execution, as it reveals hidden paths that cannot be discovered directly from the final CFG representation.

We will use this example in Chapter 4 for performing our algorithm. Its related code can be found in Figure 3.1.

From this example, it is followed that we need to keep track of the stacks for each block. One easy approach would consist in modifying the nodes of the CFG, so they represent the combination of a basic block and a stack associated before executing the instructions. From now on, we will use the term node in a CFG to denote the combination

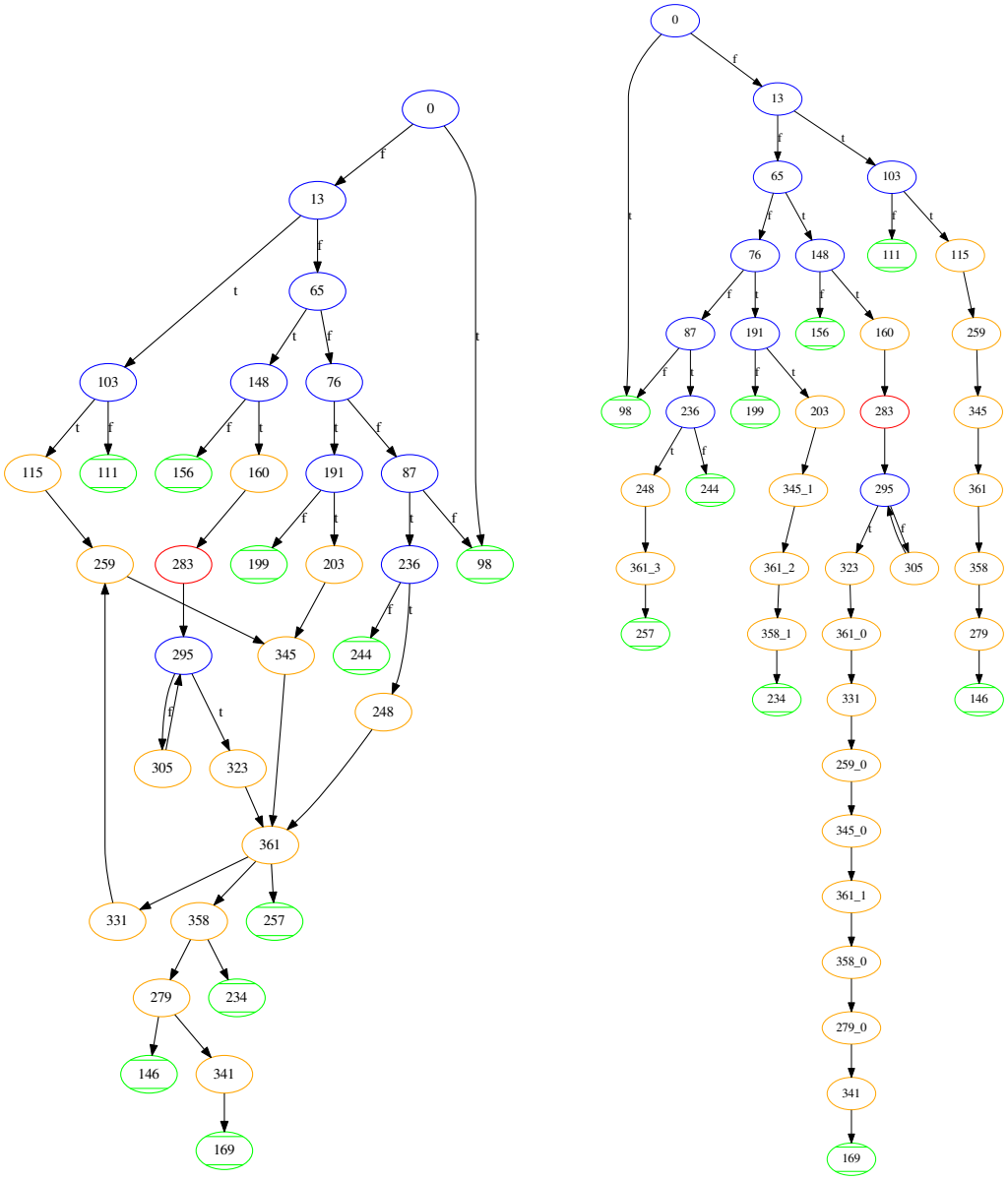


Figure 1.2: Comparative between a program CFG, and its cloned version

of a basic block of instructions and an input stack. Then, a direct edge would join a node with another one if the first node's jump target matches the second's block initial address, and the stack obtained after executing the instructions in the first node is the same as the initial one in the second node.

In this dissertation, we will propose an algorithm based in the ideas mentioned above. In particular, we will try to address the following points:

- Develop an algorithm that takes an EVM program as an input, and generates CFG as an output. This algorithm has to resolve the cloning issue, ensuring ambiguous paths cannot be found in the CFG.
- Introduce some definitions and ideas that can be useful for carrying out similar analysis with other languages.

- Guarantee the termination of the analysis, and show that it is also efficient.

- Prove our analysis is sound. This way, we will ensure that for each possible program flow, it exists its corresponding path in our CFG.

- Determine and prove those cases in which accuracy is lost with our analysis. In order to achieve this goal, we are going to define a semantics that overestimates the real semantics, and prove completeness. Therefore, we know the loss of accuracy is produced as a result of the overestimation, which can be tracked down. In particular, we will prove that the only source of overestimation comes from conditional jump instructions that are always taken or skipped. This problem is also present in languages based in fixed jumps, so this way we manage to resolve all problems intrinsically associated to deal with jumps in a stack.

- Propose an efficient implementation from the analysis, and discuss the details.

We have previously discussed briefly the main features a language must have in order to be considered. Now we are going to formally specify the requirements needed for our proposed algorithm:

- Branch instructions operands are obtained from a global stack, that is previously introduced through a `PUSH` instruction and no arithmetical instructions are involved in obtaining this value.

- Branch instructions determine the target address at the moment they are executed.

Note that we can also have other architectural structures, such as memory or registers. They don't interfere in the analysis, as long as they don't have any impact on how jumps are resolved.

In our case, instead of studying a general language, we will focus on EVM, the virtual machine Ethereum uses for deploying contracts and executing transactions.

1.3 Contributions

This dissertation proposes a method for generating a control-flow graph (CFG) without ambiguity, by cloning blocks that are contained in different paths. Thus, the following chapters have been included:

- Chapter 2 is introductory, it gives a brief description of data-flow analysis and Solidity. Data-flow analysis provides the theoretical basis of our proposed analysis and the Solidity language shows some of the capabilities of smart contracts.

- In Chapter 3, we present a general overview of EVM code behaviour and the features we need to consider for running the analysis. CFG generation relies on determining the possible jump directions, which exclusively depends on the stack. For proving the correctness of the analysis, we also introduce an EVM semantics based on stack states. These semantics focuses on the semantics of basic stack operations (`PUSHx`, `DUP x`, `SWAP x`), while all other operations just change the size of the stack.

- In Chapter 4, we propose a method to obtain a non-ambiguous CFG. It takes an EVM program as an input, and iteratively obtains the contents of the stack for each program point. Then, each block is cloned for each stack associated to its first instruction, and edges of the CFG are added by checking whether the output stack of a cloned block and the input stack of one of the copies of next block match.

- Chapters 5 and 6 give detailed proofs of soundness and completeness of our analysis respectively, by using the semantics defined in Section 3.3.1. Soundness can also be extended to EVM semantics, whereas completeness only works for our simplification. Nevertheless, by proving completeness, we find out which cases are the ones that have been over-approximated with our analysis.

- All the proposed changes have been included in Oyente*, a tool that obtains a CFG representation from EVM code integrated in Gastap. Oyente* is included in EthIR project, as CFG is an intermediate product obtained for constructing a rule-based representation of blocks that is later used in Gastap, the tool whose performance we are trying to improve. A CFG generated by Oyente* is obtained by symbolic execution, and doesn't consider duplicating nodes. Thus, non-existing paths may be obtained from this representation, which can lead to inaccuracy issues in other tools that rely on it. Commits are reflected in Github repository, where detailed instructions about installation and usage can be found. Explanations of Gastap architecture, Oyente* implementation and a case of use of Gastap will be included in Chapter 7.

- In Chapter 8, we present the results of the experiments, comparing the performance of the algorithm using Oyente* before and after including our algorithm, and the overall performance of Gastap. We have analyzed a total of 10,796 files.

- Finally, we present the conclusions of this project, as well as the modifications that will be considered for further development in Chapter 9.

Chapter 2

Foundations

In this chapter, we introduce briefly some of the concepts we will be using in later chapters and that are background knowledge for this dissertation: data-flow analysis and Solidity. EVM will be discussed in Chapter 3, as we need to analyze in more depth its properties and introduce some related definitions in order to formalize the proposed analysis.

Solidity section assumes the reader is familiarized with some of the concepts related to smart contracts and Ethereum. If this is not the case, its better to skip Section 2.2 and return once Chapter 3 has been read.

2.1 Data-flow analysis

Data-flow analysis is a static analysis technique that aims at obtaining information about the properties of a program at different program points. This information can be later used for several purposes, such as program optimization, program debugging, or checking whether a certain program point can be reached or not.

A data-flow value for a program point is a set of program values that can be obtained at that point. The set of these values represents the *domain* we are considering for our analysis. For instance, if we want to study all possible variable assignments at certain point, a data-flow value represents a set of possible variable assignments when reaching that program point.

Data-flow analysis is based on the study of the flow of data along program execution paths. For that purpose, instructions are grouped into basic blocks, *i.e.* sequences of instructions that are always executed together. This way, a path corresponds to a sequence of basic blocks $B_1 \mapsto B_2 \mapsto ... \mapsto B_n$, such that the data-flow values at the beginning of block B_i correspond to the final data-flow values of block B_{i-1}.

Usually, the execution paths are represented through a Control-Flow graph. It consists of a graph representation of the possible flows of the program, where its nodes represent the basic blocks, and its edges represent the flow between different nodes. This way, the propagation of values between different blocks can be performed. As we have stated previously, it is usually very easy to generate this graph directly from the code. This graph can contain cycles, representing the possibility of infinite paths.

Then, using the information from the Control-Flow graph, we can define the *prec* function, which is a function that given a block, obtains its possible previous blocks considering all execution paths:

$$prec(B_i) = \{B_j \mid \exists \textit{ a program execution path s.t. } B_0 \mapsto ... \mapsto B_j \mapsto B_i ...\}$$

There are two different types of analysis, depending on whether we are interested in propagating information backwards (Backward problems) or forward (Forward problems). From this point, we will consider only Forward problems, as we are interested in propagating values forward.

Following this approach, we can define a set of data-flow equations, two for each block B_i:

$$in(B_i) = \bigsqcup_{B_j \in prec(B_i)} out(B_j)$$

$$out(B_i) = \tau_i(in(B_i))$$

where $\tau_i(S)$ is the corresponding transfer function of that analysis, a function that given a set of data-flow values, obtains the corresponding data-flow values after executing all the instructions in the block according to its semantics.

We also need to define $\sqcup$ operator, to combine data-flow values from different predecessors.

When solving these equations, we generate all the in/out data-flow values for each block. The information at a certain point inside a block is easily obtained by applying the transfer function to the data-flow values at the beginning of that block.

Note that several solutions may satisfy the equations above. Solutions correspond to fixed point of the propagation function our equations define. In some cases, we might be interested in the greatest fixed point or the least fixed point, depending on the particular analysis. In our case, we are interested in computing the least fixed point, so we will introduce the requirements needed for computing this value.

A possible algorithm consists in initializing the sets *in* and *out* and evaluate all the equations until a fixed point is reached, *i.e.* until the whole system stabilizes.

In order to perform this algorithm, we first need to identify the possible entry points *i.e.* the set of all possible instructions that can be executed first. Their associated blocks correspond to the first block in an execution path B_0, so we have to initialize $in(B_0)$ to a proper value if we want to guarantee that correct results are obtained. The remaining blocks can be initialized to the empty set.

However, this is not sufficient to guarantee that a fixed point will be obtained through this procedure. We need to introduce a partial order $\sqsubseteq$ in our *domain* D, such that $(D, \sqsubseteq)$ is a complete lattice, *i.e.* $\sqcup Y$ exists for any subset Y. It is also required that the transfer function τ is monotone according to the complete lattice. Besides, if we want to guarantee that the algorithm converges, infinite chains cannot exist.

Therefore, an instance of a data-flow analysis must include the following elements:

1. A CFG, so that we have a representation for the execution paths in our program.
2. A complete lattice $(D, \sqsubseteq)$, which must also guarantee no infinite chains can be found.
3. Initial data-flow values for those blocks that represent the beginning of a path.
4. An operator $\sqcup$, so that we can combine information from predecessors.

5. A transfer function τ_n for each block n in the CFG, to propagate the values following the instruction semantics. This function must be monotone according to the partial order.

Our analysis differs slightly from the idea above. We are trying to generate a CFG following a data-flow analysis, and therefore, we cannot generate beforehand all the equations in the system, as we cannot delimit the boundaries of the basic blocks nor to define the *prec* function. Instead, equations will be generated while performing the propagation, and there will be a *in* and *out* function for each program point.

Once the solution is obtained, we will be able to build a CFG. The initial point of a block corresponds to either the entry points, or to those program points that have several equations tied to them, which means they have several predecessors. The final point of a block correspond to end instructions, jump instructions or those points which next point corresponds to the beginning of another block. An edge joins each node with its successors and predecessors.

2.2 Solidity

Most of the information discussed in this chapter has been directly adapted from Solidity documentation [6]. Solidity is constantly evolving and adding new features, so it is difficult to find other up-to-date references.

Solidity is a high-level programming language that allows users to create smart contracts that can be later compiled to bytecode and run on the Ethereum Virtual Machine (EVM). It is an object-oriented language, which resembles C++, Python and JavaScript.

It is also a Turing-Complete language, allowing users to develop smart contracts as expressive as they need. This is the main advantage over Bitcoin, which only allows executing basic transactions.

It supports inheritance, and includes some complex data structures, such as mappings, arrays or user-defined structs. It also allows creating *Events*, which is a functionality Solidity includes to generate log information for a transaction. They are stored in a special data structure in the blockchain, and they can be accessed externally. Applications can also subscribe to a certain event in a contract, so whenever it is triggered, they are notified of the contents of the log.

Solidity is compiled into EVM bytecode by the **solc** compiler. This compiler also can estimate the gas consumption of a transaction in case this consumption is constant.

Solidity files contain one or several contracts. These contracts can be either normal contracts, or other special ones called libraries and interfaces.

First line in source files usually includes a *pragma* declaration, that allows enabling or disabling certain features in the compiler by stating the versions of the compiler that support current declarations. This feature is included to ensure compilation declarations cannot produce errors due to incompatibilities with other versions.

Contracts can be created via a transaction, or from a call from another Solidity contract. Contracts can contain a *constructor* function, which is a function that will be only invoked when deploying the contract. Once a contract is created, all the information about public and external functions becomes visible, and the code is stored on the blockchain.

Functions included in a contract can have four different types of visibilities, depending on the cases they can be called. These four values are *external*, *public*, *internal* and

private. Nevertheless, all functions are visible to external observers, as they are stored in the blockchain.

Multiple data types are supported by Solidity: boolean, integer, unsigned integer or string. Integers and unsigned integers can have different sizes, from 8 to 256 bits in steps of 8 bits. Arithmetic and bitwise operations can be performed, as well as other operations such as *exp* or *modulo*.

It is also included a new type called *address*, that represents Ethereum addresses. It has two different forms: *address*, that represents a plain address; or *address payable*, which allows the user to send Ether to that address.

Data can be allocated in three different structures: memory, storage and calldata. Memory's lifetime is limited to an external function call, whereas storage values are persistent through different transactions. Calldata is a special structure, that is used to store and retrieve the parameters of a function call.

In previous versions, data location could be omitted. However, in new versions, an explicit location must be given for complex types. In our case, we are working with contracts in a version prior to 0.5.0, so our examples don't include this explicit behaviour.

Contracts also contain some special variables and functions, that are used for meta information about the blockchain or other capabilities. For instance, we can obtain the address of the sender of the current transaction through `msg.sender` or obtain the gas left through `gasleft`.

We haven't discussed into much detail some of the definitions above, as a full overview is included in EVM section, in Chapter 3. EVM bytecodes correspond to the compilation of Solidity code, so most of the structures and data types we have mentioned have its correspondent low-level version, that are managed through EVM instructions.

Two examples of Solidity code be found in Figures 3.1 and 7.3.

Chapter 3

Ethereum Virtual Machine

3.1 Overview

Smart contracts are computer programs deployed in a blockchain that can store values and also contain functions whose execution cannot be falsified. This technology opens a wide variety of opportunities, probably being economic transactions the most notorious ones. More and more applications are starting to rely on blockchain, as it provides a safe and trusted network for making transactions.

Nowadays, Ethereum has become the most popular platform to deploy smart contracts. Ethereum allows any user to deploy a contract easily, and have it added to a block. Blocks take around 13s to be included in the Ethereum blockchain. This is incredibly fast, taking into account that it needs to be replicated worldwide. Thus, roughly a million transactions are carried out every day. This data can be looked upon Etherscan website [7].

Ethereum allows developers to deploy their dApps on top of its platform. Miners compete to validate and include transactions in a block, which will be later added into the blockchain. The first person to solve a difficult hashing problem is the one that includes the transaction, and who will be rewarded for mining. This approach is called Proof of Work, which tries to solve the Byzantine fault, by setting a difficult problem to solve but easy to check. If a malicious node tries to falsify a transaction in the network, it will take longer to figure out a solution for the problem in comparison with trustful nodes. So when it wants to share its information with other nodes, its chain of blocks will be shorter than the ones trustful nodes have, and therefore, such false information will be rejected.

This approach is really power consuming, and Ethereum is planning to shift into Proof of Stake: instead of forcing validators to compete, it lets them mine blocks if they own a certain participation in the system. The more participation you hold, the higher chance you have to validate a transaction.

Ethereum uses the Ethereum Virtual Machine (EVM) to run smart contracts. Each node in the network has its own instance of EVM, which allows anybody to deploy their own smart contracts. Coders usually program in high-level program languages, such as Solidity or Vyper. Then high level code is translated into EVM bytecode instructions, which are run by EVM. This way, portability is ensured.

EVM is a stack-based, big endian virtual machine that works with 256-bit words. It allows users to create contracts, or send transactions to existing contracts, which can carry payload for specifying the interaction with the contract or additional information.

For instance, function calls are specified by using the first 4 bytes of data sent with a transaction, using the four first bytes of the SHA3 representation of the function signature.

When contracts are created, a constructor function will be invoked. This function will be only invoked at this point, and no further transactions can invoke it. Nevertheless, the entry point of the smart contract will be always the first bytecode, so there is no distinction between creating a contract or running a transaction.

Handling data is really tricky, as there are four different ways to deal with it. We have already mentioned the first one: when carrying out a transaction, it can have a dataload associated that can be accessed through `CALLDATALOAD`, `CALLDATASIZE` and `CALLDATACOPY` instructions.

The three other ways involve different data structures that can be used to store data. The first one is the *stack*, which only stores uint256 values. It is mainly used to perform the execution: opcodes retrieve their operands from the stack, and it is also used to store function arguments and return addresses. Therefore, in order to perform our analysis, we will need to understand deeply how it works, and propose an abstraction. Nearly all opcodes modify the stack, but we want to remark those that are used to manage directly its content: `PUSHx v`, `DUPx`, `SWAPx` and `POP`, with the usual meaning.

Finally, there are two other structures that can be used by programmers to store values. The first one is called *memory*, which is a non persistent array of uint8 values. These values are lost between different transactions, and can be managed through `MLOAD`, `MSTORE` and `MSTORE8` opcodes. The second one is called *storage*, which is a persistent associative map which uses uint256 as both keys and values. All global variables that we declare in our contract are stored in this structure. It can be accessed through `SLOAD` and `STORE`.

We might wonder why making a explicit separation between memory and storage. We will study later that executing an opcode has an implicit cost associated. Working with the storage has a higher cost associated because these data is stored in the blockchain, thus encouraging developers to use their resources wisely.

One of the main properties of EVM is being a Turing-complete language. This feature allows smart contracts to be as expressive as we want, being able to use loops or other basic structures to program. However, it also arises several problems, for instance being able to create contracts that cannot terminate. This is a usual problem for any programming language, but it becomes even more serious in Ethereum, as miners are the ones that run transactions and resources are limited.

There is also the issue of incentivizing people to become miners. More miners in the network, more secure and fast the system becomes. As Ethereum currently works under Proof of Work, the reward must be worthy to encourage people to compete.

We will see in next subsection how both issues are dealt in Ethereum.

3.2 Gas

Ethereum proposes a way to avoid non-terminating executions and reward miners by introducing *gas* into the system. Gas is a unit that measures the amount of computational effort it is needed to execute each EVM instruction.

Each EVM bytecode has a gas cost associated, that can be either constant or that depends on some parameter. For instance, creating a contract costs 21000 units of gas. Then, when miners execute a transaction and include it in a block, they are awarded an

amount of gas dependent on the operations executed.

The gas doesn't have a monetary value per se: the sender of the transaction fixes the conversion between gas and ether, which is the Ethereum official token, when sending the transaction. This value is usually set to 20 Gwei, a unit of ether. If the sender wants to make sure the contract is mined really fast, then by setting a high gas price, it makes it more appealing for possible miners.

Senders also must set a gas limit beforehand. This limit provides a way to avoid non-terminating executions: the execution of the contract will be aborted once this limit is exceeded and the system state will be reverted to the initial one. However, this transaction will still be recorded in the ledger, and the execution fee will be still paid to the miner.

In case the execution is successful and there is some remaining gas available, this gas will be returned to the sender by using the conversion to ether.

It seems clear then that estimating the possible amount of gas spent on a transaction is really useful beforehand. Most opcodes have a constant gas fee associated, so in many cases it is really easy to determine this value. However, there are other opcodes that depend on parameters, so inferring an upper bound for gas limit becomes really difficult. In fact, Solidity compiler **solc** can estimate the amount of gas spent in those cases in which this value is constant. Otherwise, it just infers gas limit is $+\infty$.

A naive solution consists of just setting a really high gas limit to ensure we won't be facing a shortage situation. This way, all non-spent gas would just be returned to the owner. However, there are two possible flaws in this approach:

- Ether is spent before submitting the transaction, once we have settled both the gas price and the gas limit. If gas limit is high then we would require a big initial investment beforehand, so we need to have more Ether and sender won't be able to recover it until a miner has validated the transaction.

- Transactions with high gas limit are less appealing for miners. Transactions are stored in blocks that are later added to the blockchain. These blocks have a gas limit, and miners can only add more transactions to the block while this limit is not exceeded. They can dismiss a transaction directly if the remaining capacity of their current block is exceeded or if they hint that most of the gas will be returned.

It is clear that the best case scenario is the one that sets a limit that guarantees the final execution, but returns as little gas as possible.

Some tools address this problem, trying to infer an upper bound that may depend on some parametric value. Among them, we can find Gastap, that relies on EthIR for obtaining the RBR representation. In this context, it is clear that ensuring the generation of CFG is vital for correctly finding the gas limit: inaccuracies may lead to the inability to obtain an upper bound and to gas limit estimations that cannot ensure the execution. As explained, this can lead to major economic losses.

3.3 EVM Semantics

Defining a complete EVM semantics is a really difficult task. We have to model stack, memory and storage, being the last one really difficult to analyze, as it is persistent between transactions and therefore, we cannot keep track of all values that have been stored within it. As we are focusing in how jumps are managed, then we can define an overestimated semantics that just takes into account how stack evolves through execution.

There is a possible major inconvenience when only focusing on the stack: there could be some jump values that are somehow obtained from the storage or the memory, and as we are just omitting it, we cannot deduce the jump address. As we will see in Section 8.2, there is only one specific type of contracts in which branch target is obtained from memory: those that involve high order functions. It only represents a small fraction of all contracts considered (less than 0,01%).

In fact, in all other cases, jump target address is obtained directly from a `PUSHx` instruction. Other instructions that can modify this value in the stack are `SWAPx` and `DUPx`, whereas we can just ignore the bitwise or arithmetic operations.

We should also note that not all values pushed to the stack can be eventually a branch address. EVM has a special opcode to denote the possible jump destination from a branch opcode: `JUMPDEST`. This will help us reduce our analysis, as we just need to consider all those stack values whose associated opcode (if any) is a `JUMPDEST` instruction.

Before starting with the definitions, we will introduce the example we will be using to illustrate our analysis:

Example 3.1. *Example 1.2 shows contract `Sum` associated CFGs. In next chapters, we are going to analyze in depth this contract and apply our proposed algorithm.*

It contains four different functions that make several calls to function `hola`, thus generating paths that share nodes, but end in different routes, as we have previously discussed.

Figure 3.1 shows the Solidity code, and a subgraph of the CFG, containing the corresponding opcodes for each of the blocks.

Nodes correspond to the different blocks of code, and edges represent the possible execution paths. In our example, there are two type of nodes. The white nodes represent those nodes that won't be cloned, and the gray ones represent the nodes that need to be cloned.

There are also two type of edges: solid edges represent the path that starts from block `73` and dashed edges the one that starts from `A0`. These two entry points correspond to publicly invoking the function `__callback` and invoking from function `suma`, respectively. The end directions `92` and `A9` are pushed in instructions marked with a black and blue star respectively. In this Chapter, we will be using the hexadecimal notation, whereas our generated CFGs use decimal notation instead. This explains why previous directions don't appear in Example 1.2: blocks `73` and `A0` correspond to blocks `115` and `323`, respectively.

Note that this separation isn't made explicit before performing the proposed analysis. This example reflects how different non-feasible paths could be found if we didn't make an explicit separation. ∎

Firstly, we will study how EVM programs are structured.

Definition 3.1 (EVM program)**.** *An EVM program $P = \{b_0...b_n\}$ is a set of bytecodes $b_0...b_n$.*

These bytecodes have arbitrary size, depending on the instruction they encode. Therefore, not every bytecode represents an instruction. We are using *size(b)* to refer to the number of bytes an instruction increments the value of the program counter after being executed. Most bytecodes increment this number only by one unit, but there are other opcodes, such as `PUSHx v` that adds the word v of x bytes to the stack. In this case, size(`PUSHx`) = x+1.

```
1
2  pragma solidity ^0.4.11;
3
4  contract Sum {
5
6  function suma () returns (uint sol) {
7      sol = 0;
8      for(uint i = 0; i < 5; i++)
9              sol = sol+11;
10     hola();
11     adios(10);
12  }
13
14 function hola() {
15     uint i = 0;
16     i = i+15;
17     }
18
19 function adios(uint m) {
20     uint c = 14;
21     c = c+m;
22     comer(c);
23 }
24
25 function comer(uint x) {
26     x = x*x;
27     hola();
28 }
29
30 }
```

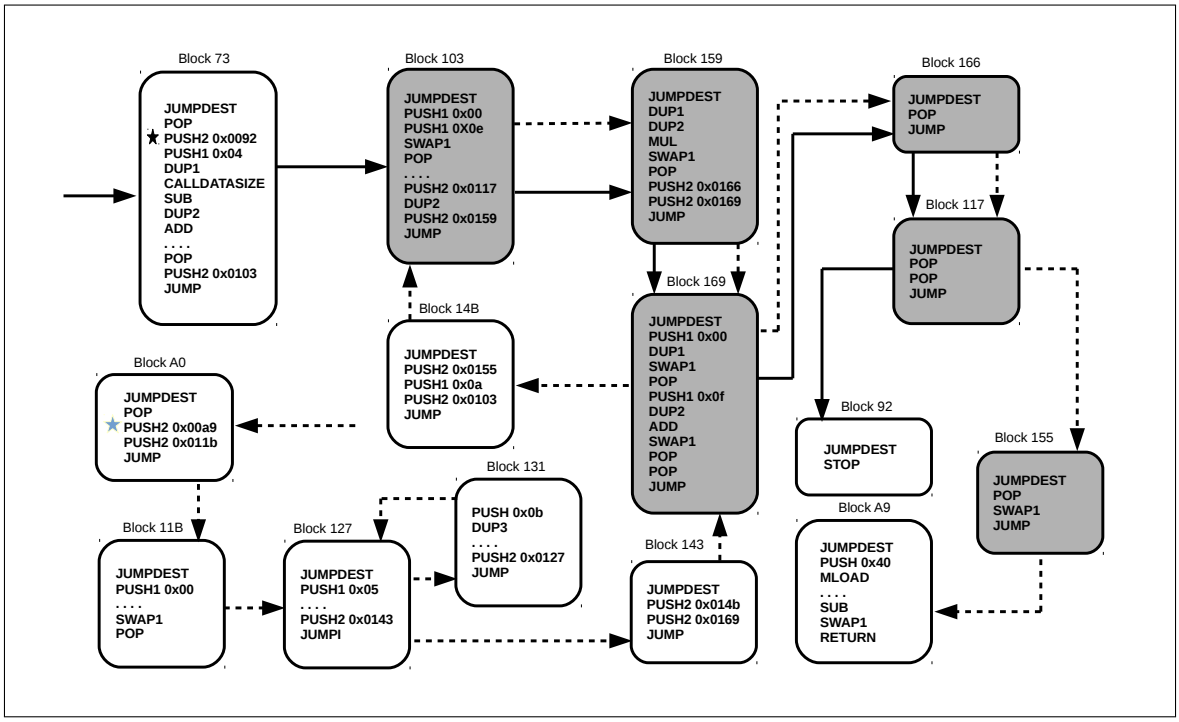


Figure 3.1: Solidity code for `Sum` example contract and its associated CFG

Definition 3.2. *Given a program $P = \{b_0...b_n\}$, we denote*

$$\mathcal{J}(P) = \{i \mid b_i \equiv \texttt{JUMPDEST}\}$$

as the set of all possible jump addresses. As P is a finite program, $\mathcal{J}(P)$ is also finite and well-defined.

This feature is really useful for analysis, as we can statically determine which values in the stack are the ones that can determine an edge in a CFG. We can just ignore all those values that we know for sure they cannot be a possible branch target. Thus, instead of using a global definition of an abstract stack, it will be associated to a concrete program.

From this point, we will simply use $\mathcal{J}$ instead of using $\mathcal{J}(P)$, unless it is not clear which program the set is associated to.

Example 3.2. *The set of possible jump addresses corresponding to contract in Figure 3.1 is*

$$\mathcal{J} = \{\texttt{73}, \texttt{A0}, \texttt{11B}, \texttt{127}, \texttt{143}, \texttt{169}, \texttt{14B}, \texttt{103}, \texttt{159}, \texttt{169}, \texttt{166}, \texttt{117}, \texttt{92}, \texttt{155}, \texttt{A9}\}$$

Note that all these directions correspond to the beginning of a block, but not all blocks start with a `JUMPDEST` *instruction. For example, block* `131` *starts with a push instruction.* ∎

EVM stacks have a unique feature that will make our analysis easier: it has a maximum size of 1024 elements. This means we can assume the stack is flattened in those Solidity programs without recursion. Each stack variable will be denoted as s_i, and we will use $\mathcal{V}$ to represent the set of all possible stack variables. In our representation, if a stack has size n, the variable stack s_{n-1} represents the top of the stack, s_{n-2} the following element and so on.

Definition 3.3 (Abstract stack)**.** *An abstract stack is a pair $\langle n, \sigma \rangle$, where n is the number of elements in the stack and $\sigma : \mathcal{V} \mapsto \mathcal{J}$ is a partial mapping that maps a variable stack with its associated value in case it is a valid jump target.*

Using a partial mapping instead of a total mapping has two advantages for our analysis:

- We have assumed previously that jump targets can only be obtained through a `PUSHx v` instruction. Thus, by using this partial mapping, we can just ignore those values we would have to define in case we were using a total mapping.
- It provides a simple representation for deciding whether two abstract stacks are the same or not: two abstract stacks are equivalent iff they have the same size and also share the same partial mapping. This way, we define an equivalence relation that subsumes all stacks that are essentially the same for our analysis, not considering values that aren't taken into account in branch instructions.

We saw in Example 1.1 that we are dealing with set of stacks, instead of just stacks. This will be our motivation for defining an abstract state, the structure we will be working with in our analysis.

For that purpose, we define the set of all stack states $\mathcal{S} = \{\langle n, \sigma \rangle | 0 \leq n \leq |\mathcal{V}| \wedge \sigma(s) \in \Sigma_n\}$. Σ_n is the set of all mappings using n stack variables, defined recursively as follows: $\Sigma_i = \Sigma_{i-1} \cup \{\sigma[s_i \mapsto j] \mid \sigma \in \Sigma_{i-1} \wedge j \in \mathcal{J}\}$; $\Sigma_0 = \{\sigma_\emptyset\}$, where $\sigma_\emptyset$ is the empty mapping.

Definition 3.4 (Abstract state)**.** *Given a program P, we will denote abstract state as a partial mapping π of pairs of the form $\mathcal{S} \mapsto \mathcal{P}(\mathcal{S})$.*

This definition may seem odd at first. Considering examples we have seen before, using a set instead of a mapping would be probably the first option. In fact, a set is sufficient for proposing an algorithm, but working with a partial map makes it easier to formalize the generation of edges in the CFG. We will study it later in Section 4.3.

Once we have identified the elements we will be working with in our analysis, we are going to introduce an ordering $\sqsubseteq$ between them that will be useful for developing the theoretical basis.

The abstract domain is the lattice $\langle AS, \pi_\top, \pi_\bot, \sqcup, \sqsubseteq \rangle$, where AS is the set of abstract states and $\pi_\top$ is the top of the lattice defined as the mapping $\pi_\top$, such that $\forall s \in \mathcal{S}, \pi_\top(s) = \mathcal{S}$. $\pi_\bot(s)$ is defined as the empty abstract state.

The function $img(\pi, s)$ is defined as follows: $\pi(s)$ if $s \in dom(\pi)$, $\emptyset$ otherwise.

Given two abstract states π_1, π_2, we use $\pi = \pi_1 \sqcup \pi_2$ to denote the least upper bound, which is an abstract state that satisfies $\forall s \in dom(\pi_1) \cup dom(\pi_2),\ \pi(s) = img(\pi_1, s) \cup img(\pi_2, s)$. Besides, $\pi_1 \sqsubseteq \pi_2$ iff $dom(\pi_1) \subseteq dom(\pi_2)$ and $\forall s \in dom(\pi_1), \pi_1(s) \subseteq \pi_2(s)$.

In fact, it was sufficient to define abstract states as partial mappings of the form $\mathcal{S} \mapsto \mathcal{S}$, as each abstract stack will only point to another abstract stack in our analysis. However, by using our definition, we characterize the abstract domain as a complete lattice instead of a chain-complete ordered set.

With these notions of abstract stack, abstract state and abstract domain, we can finally introduce the stack-based semantics we will be working with.

3.3.1 A simplified EVM semantic for analysis purposes

Given a program $P = \{b_0, ..., b_n\}$, we will denote the *program state* S as a pair of the form $\langle pc, s \rangle$, where $pc \in \{0, ..., n\}$ and s is an abstract stack. pc refers to the point of the program we are currently in, and the abstract stack reflects a possible stack state before executing instruction b_{pc}.

The following notation is used to fully express the rules in our semantics. We refer to positions in the stack with s_i, being the top of the stack s_{n-1}. Given a partial mapping σ, the expression $\sigma[s_i \mapsto y]$ indicates the map that shares the same values $\forall s_j \neq s_i$ as σ, and $\sigma(s_i) = y$. We are also using $\sigma \backslash [s_i]$ for denoting the map that shares all the values with σ for all $s_j \neq s_i$ and is undefined for s_i.

Figure 3.2 shows our proposed semantics for handling jumps. We will be using this semantics throughout the analysis section, as it contains the needed information we need to represent for developing our algorithm.

This semantics over-approximates the real EVM semantics. This can be seen with rules (2) and (3), that can be both applied for the same opcode `JUMPI`: our semantics is non-deterministic, as opposed to the real one. However, we will prove that this is the only source of overestimation in the analysis.

From the rules of our semantics, it is clear that all of them focus on branch opcodes, `PUSHx v`, `SWAP x` and `DUP x`. This is due to previous assumption that addresses are introduced in the stack through `PUSHx v` instruction, and no arithmetic instructions can manipulate them.

The rest of instructions just have an impact in the size of the abstract stack: this is simulated with rule (12), where δ and α refers to the amount of elements of the stack consumed and added to the stack respectively by executing that instruction. These values

$$(1)\quad \frac{b_{pc} = \texttt{JUMP}}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle \sigma(s_{n-1}), \langle n-1, \sigma\backslash[s_{n-1}]\rangle\rangle}$$

$$(2)\quad \frac{b_{pc} = \texttt{JUMPI}}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle \sigma(s_{n-1}), \langle n-2, \sigma\backslash[s_{n-1}, s_{n-2}]\rangle\rangle}$$

$$(3)\quad \frac{b_{pc} = \texttt{JUMPI}}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle pc+size(b_{pc}), \langle n-2, \sigma\backslash[s_{n-1}, s_{n-2}]\rangle\rangle}$$

$$(4)\quad \frac{b_{pc} = \texttt{PUSH}x\ v, v \in \mathcal{J}}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle pc+size(b_{pc}), \langle n+1, \sigma[s_n \mapsto v]\rangle\rangle}$$

$$(5)\quad \frac{b_{pc} = \texttt{PUSH}x\ v, v \notin \mathcal{J}}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle pc+size(b_{pc}), \langle n+1, \sigma\rangle\rangle}$$

$$(6)\quad \frac{b_{pc} = \texttt{DUP}x, s_{n-x} \in dom(\sigma)}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle pc+size(b_{pc}), \langle n+1, \sigma[s_n \mapsto \sigma(s_{n-x})]\rangle\rangle}$$

$$(7)\quad \frac{b_{pc} = \texttt{DUP}x, s_{n-x} \notin dom(\sigma)}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle pc+size(b_{pc}), \langle n+1, \sigma\rangle\rangle}$$

$$(8)\quad \frac{b_{pc} = \texttt{SWAP}x, s_{n-1} \in dom(\sigma), s_{n-x-1} \in dom(\sigma)}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle pc+size(b_{pc}), \langle n, \sigma[s_{n-x-1} \mapsto \sigma(s_{n-1}), s_{n-1} \mapsto \sigma(s_{n-x-1})]\rangle\rangle}$$

$$(9)\quad \frac{b_{pc} = \texttt{SWAP}x, s_{n-1} \in dom(\sigma), s_{n-x-1} \notin dom(\sigma)}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle pc+size(b_{pc}), \langle n, \sigma[s_{n-x-1} \mapsto \sigma(s_{n-1})]\backslash[s_{n-1}]\rangle\rangle}$$

$$(10)\quad \frac{b_{pc} = \texttt{SWAP}x, s_{n-1} \notin dom(\sigma), s_{n-x-1} \in dom(\sigma)}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle pc+size(b_{pc}), \langle n, \sigma[s_{n-1} \mapsto \sigma(s_{n-x-1})]\backslash[s_{n-x-1}]\rangle\rangle}$$

$$(11)\quad \frac{b_{pc} = \texttt{SWAP}x, s_{n-1} \notin dom(\sigma), s_{n-x-1} \notin dom(\sigma)}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle pc+size(b_{pc}), \langle n, \sigma\rangle\rangle}$$

$$(12)\quad \frac{b_{pc}^{\delta,\alpha} \in otherwise, b_{pc}^{\delta,\alpha} \notin End}{\langle pc, \langle n, \sigma\rangle\rangle \Rightarrow \langle pc+size(b_{pc}), \langle n-\delta+\alpha, \sigma\backslash[s_{n-1}, ..., s_{n-\delta}]\rangle\rangle}$$

Figure 3.2: Simplified EVM semantics for handling jumps

are specific of every opcode, and can be found in Ethereum Yellow Paper [8]. However, this document contains too many details to understand easily which opcodes are available and their related semantics. A summary of EVM bytecodes with a brief explanation of their usage can be found in reference web [9].

We also use the function *size* defined previously for updating the value of the program counter to the next bytecode which contains an instruction.

It is important to note that `PUSHx v`, `SWAPx` and `DUPx` rules are not single rules, but rather a family of parametric rules that have a different effect depending on the value of x.

Rules 1 to 3 refer to branch instructions. As we have discussed before, we can safely assume that $\sigma(s_{n-1})$ contains a valid target address. `JUMP` instruction just takes the top stack value and changes the value of the program counter to that value, whereas `JUMPI` takes two top values and should decide whether to jump or not depending on the second one. As this second value is a boolean one, we cannot guarantee this is a known value, and we simulate both possibilities by defining two different rules that can be applied.

Rules 4 and 5 include both cases for `PUSHx` instruction: the value we have pushed can either be a jump target or not. In the first case we need to update the partial mapping, whereas in the second it remains the same. Therefore, we need to define two different rules for simulating both cases.

Same approach happens with rules 6-7 and 8-11. There are 4 rules that involve `SWAP x` opcode, as we have two values changed at the same time and there are four possibilities according to being a branch address or not.

Rule 12 summarizes the rest of possible opcodes, that have no real impact in the values we store in the stack. Using the α and δ notation allows us to cover all the rules that have no significant impact in the stack in just one rule.

A complete execution is a trace of the form $S_0 \Rightarrow S_1 \Rightarrow \ldots \Rightarrow S_n$ where $S_0 \equiv \langle 0, \langle 0, \sigma_\emptyset \rangle \rangle$ is the initial state, $\sigma_\emptyset$ is the empty mapping, and S_n corresponds to the final state.

If we use the real semantics instead for defining a complete execution, there will no infinite traces, as any transaction that executes EVM code has a finite gas limit and every instruction executed consumes some amount of gas. When the gas limit is exceeded, an out-of-gas exception occurs and the program halts immediately.

However, using our proposed semantics, there could be cases in which an infinite trace would be obtained, as gas consumption isn't considered for our semantics, assuming we have ∞ gas available. In that situation, two different cases may occur:

- The same program state appears twice in the execution. This would lead to a cycle appearing in our CFG, meaning that this trace is correctly represented in the graph. However, even if we can generate infinite traces in our CFG using cycles, we know that only finite paths correspond to possible flows, so no information is lost.

- The same program state doesn't appear more than once in a trace. We already know that this situation will be impossible with the real semantics even if we had ∞ gas, as there is a finite number of possible program states due to stack restrictions. Therefore, our analysis of that contract would fail, and the same that would happen if we tried to invoke the corresponding transaction in Ethereum. An example of this situation will be provided in Section 8.2.

We conclude that our simplified semantics doesn't lose nor include information for being able to generate infinite traces.

Example 3.3. *We are going to simulate a complete execution in our previous example. Let's assume we have a previous trace* $S_0 \Rightarrow S_m$*, where* $S_m = \langle \texttt{12D}, \langle 4, \sigma[s_1 \mapsto \texttt{A9}]\rangle\rangle$*. This program counter corresponds to instruction* `PUSH2 0x0143` *in block* `127`*. We have already determined in Example 3.2 that* $\texttt{143} \in \mathcal{J}$*, and we also know that size(*`PUSH2 0x0143`*)=3.*

Therefore, by applying rule 4, we obtain that

$$S_m \Rightarrow \langle \texttt{130}, \langle 5, \sigma[s_4 \mapsto \texttt{143}, s_1 \mapsto \texttt{A9}]\rangle\rangle := S_{m+1}$$

Now we have that $b_{\texttt{130}} =$ `JUMPI`*, we can either apply rule 2 or rule 3. In the first case, we would obtain that*

$$S_{m+1} \Rightarrow \langle \texttt{143}, \langle 3, \sigma[s_1 \mapsto \texttt{A9}]\rangle\rangle$$

whereas in the second one

$$S_{m+1} \Rightarrow \langle \texttt{131}, \langle 3, \sigma[s_1 \mapsto \texttt{A9}]\rangle\rangle$$

This way, we would generate all possible flows in our program. ∎

Chapter 4

Analysis

In order to define a concrete algorithm for obtaining a CFG, we need to introduce some definitions that formalize the ideas mentioned in Section 1.2.

As we have mentioned before, in general, a *block* is a maximal sequence of straight-line consecutive code in the program with the property that its execution always starts from the first instruction, and it cannot halt or branch, except possibly at its last instruction.

This definition is adapted for an EVM program as follows:

Definition 4.1 (blocks)**.** *Given a EVM program* $P = \{b_0, \ldots, b_n\}$*, we define*

$$blocks(P) = \left\{ B_i \equiv b_i, \ldots, b_j \;\middle|\; \begin{array}{l} (\forall k.i < k < j, b_k \notin Jump \cup End \cup \{\texttt{JUMPDEST}\}) \wedge \\ (\ i{=}1 \vee b_i{\equiv}\texttt{JUMPDEST} \vee b_{i-1}{=}\texttt{JUMPI}\ ) \wedge \\ (\ j{=}n \vee b_j \in Jump \vee b_j \in End \vee b_{j+1}{\equiv}\texttt{JUMPDEST}\ ) \end{array} \right\}$$

where

$$\begin{array}{rcl} Jump & = & \{\texttt{JUMP},\texttt{JUMPI}\} \\ End & = & \{\texttt{REVERT},\texttt{STOP},\texttt{INVALID}\} \end{array}$$

For EVM, it is also easy to determine blocks, mainly because `JUMPDEST` opcode always identifies the target of a jump. However, other stack-based languages may not have this feature, and therefore we need to identify blocks while performing the analysis.

Blocks in Figure 3.1 have been delimited using this definition.

4.1 The transfer function

In previous Section 3.3, we have already modeled how we are representing EVM structures. Now we are introducing a function that describes the effect of each EVM opcode on that representation, so that we can simulate how our stack evolves.

We define the updating function $\lambda(b^{\delta,\alpha}, s)$ as a function that given an EVM opcode and an abstract stack s, returns the abstract stack corresponding to the execution of that opcode on s. This function is defined using Figure 4.1.

The values of α and δ are again the same of the semantics defined in Section 3.3.1. Clearly, transfer function is based on that semantics, and aims to formalize those ideas into a function that can be translated into code. In fact, the following lemma holds:

Lemma 4.1. *Let* $P = \{b_0, ..., b_n\}$ *be an EVM program and* s, s' *abstract stacks. For any index* i *such as* $0 \leq i \leq n$ *and* $b_i \notin End$*, it exists a derivation for some other index* j *such that* $\langle i, s\rangle \Rightarrow \langle j, s'\rangle$ *iff* $\lambda(b_i, s) = s'$*. In case* $b_i \in Jump$*, we need the additional requirement that* $\sigma(s_{n-1})$ *is defined.*

	$b^{\delta,\alpha}$	$\lambda(b, \langle n, \sigma\rangle)$	
(1)	`PUSH x`	$\langle n+1, \sigma[s_n \mapsto x]\rangle$	*when* $x \in \mathcal{J}$
		$\langle n+1, \sigma\rangle$	*when* $x \notin \mathcal{J}$
(2)	`DUP x`	$\langle n+1, \sigma\rangle$	*when* $s_{n-x} \notin dom(\sigma)$
		$\langle n+1, \sigma[s_n \mapsto \sigma(s_{n-x})]\rangle$	*when* $s_{n-x} \in dom(\sigma)$
(3)	`SWAP x`	$\langle n, \sigma\rangle$	*when* $s_{n-1} \notin dom(\sigma) \wedge s_{n-x-1} \notin dom(\sigma)$
		$\langle n, \sigma[s_{n-x-1} \mapsto \sigma(s_{n-1}), s_{n-1} \mapsto \sigma(s_{n-x-1})]\rangle$	*when* $s_{n-1} \in dom(\sigma) \wedge s_{n-x-1} \in dom(\sigma)$
		$\langle n, \sigma[s_{n-1} \mapsto \sigma(s_{n-x-1})]\backslash[s_{n-x-1}]\rangle$	*when* $s_{n-1} \notin dom(\sigma) \wedge s_{n-x-1} \in dom(\sigma)$
		$\langle n, \sigma[s_{n-x-1} \mapsto \sigma(s_{n-1})]\backslash[s_{n-1}]\rangle$	*when* $s_{n-1} \in dom(\sigma) \wedge s_{n-x-1} \notin dom(\sigma)$
(4)	*otherwise*	$\langle n-\delta+\alpha, \sigma\backslash[s_{n-1}, \ldots, s_{n-\delta}]\rangle$	

Figure 4.1: Updating function

Proof. Straightforward by considering all possible opcodes b_i might be and comparing the result of applying its associated rule in the semantics with the transfer function. □

Nevertheless, we are also interested in using the definition above for abstract states, as we may have different stacks in the same program point. This is our motivation for defining the transfer function:

Definition 4.2 (Transfer function)**.** *Given the set of abstract states AS and the set of EVM instructions Ins, the transfer function τ is a function of the form:*

$$\tau \ : \ Ins \times AS \mapsto AS$$

defined as follows:

$$\tau(b, \pi) = \pi' \ \ where \ \forall s \in dom(\pi), \ \pi'(s) = \lambda(b, \pi(s))$$

We will illustrate the idea with an example:

Example 4.1. *Given the following initial abstract state $\{\langle 2, \{\}\rangle \mapsto \{\langle 2, \{\}\rangle\}\}$, which corresponds to the initial stack state for executing block `73`, the application of the transfer function τ to the block that starts at EVM instruction `73`, produces the following results (between parenthesis we show the program point). To the right we show the application of the transfer function to block `A0` with its initial abstract state $\{\langle 2, \{\}\rangle \mapsto \{\langle 2, \{\}\rangle\}\}$.*

(73)	JUMPDEST	$\{\langle 2, \{\}\rangle \mapsto \{\langle 2, \{\}\rangle\}\}$
(74)	POP	$\{\langle 2, \{\}\rangle \mapsto \{\langle 1, \{\}\rangle\}\}$
(75)	PUSH2 092	$\{\langle 2, \{\}\rangle \mapsto \{\langle 2, \{s_1 \mapsto 92\}\rangle\}\}$
(78)	PUSH1 04	$\{\langle 2, \{\}\rangle \mapsto \{\langle 3, \{s_1 \mapsto 92\}\rangle\}\}$
(7A)	DUP1	$\{\langle 2, \{\}\rangle \mapsto \{\langle 4, \{s_1 \mapsto 92\}\rangle\}\}$
(7B)	CALLDATASIZE	$\{\langle 2, \{\}\rangle \mapsto \{\langle 5, \{s_1 \mapsto 92\}\rangle\}\}$
(7C)	SUB	$\{\langle 2, \{\}\rangle \mapsto \{\langle 4, \{s_1 \mapsto 92\}\rangle\}\}$
(7D)	DUP2	$\{\langle 2, \{\}\rangle \mapsto \{\langle 5, \{s_1 \mapsto 92\}\rangle\}\}$
(7E)	ADD	$\{\langle 2, \{\}\rangle \mapsto \{\langle 4, \{s_1 \mapsto 92\}\rangle\}\}$
⋮		
(8D)	POP	$\{\langle 2, \{\}\rangle \mapsto \{\langle 3, \{s_1 \mapsto 92\}\rangle\}\}$
(8E)	PUSH2 0103	$\{\langle 2, \{\}\rangle \mapsto \{\langle 4, \{s_1 \mapsto 92, s_3 \mapsto 103\}\rangle\}\}$
(91)	JUMP	$\{\langle 2, \{\}\rangle \mapsto \{\langle 3, \{s_1 \mapsto 92\}\rangle\}\}$

(A0)	JUMPDEST	$\{\langle 2, \{\}\rangle \mapsto \{\langle 2, \{\}\rangle\}\}$
(A1)	POP	$\{\langle 2, \{\}\rangle \mapsto \{\langle 1, \{\}\rangle\}\}$
(A2)	PUSH2 00a9	$\{\langle 2, \{\}\rangle \mapsto \{\langle 2, \{s_1 \mapsto A9\}\rangle\}\}$
(A5)	PUSH1 011b	$\{\langle 2, \{\}\rangle \mapsto \{\langle 3, \{s_2 \mapsto 11B, s_1 \mapsto A9\}\rangle\}\}$
(A8)	JUMP	$\{\langle 2, \{\}\rangle \mapsto \{\langle 2, \{s_1 \mapsto A9\}\rangle\}\}$

■

4.2 The constraint equation system

Now we have all the necessary definitions to define a proper algorithm to generate the CFG.

Our first step is to obtain all the abstract states for each program point. As we will see later, this is the critical part of the algorithm: once we have them, obtaining the edges of the CFG and deciding how to clone the nodes becomes really easy.

This step is going to be achieved by defining a constraint equation system that once solved obtains the abstract states:

Definition 4.3 (Addresses equation system)**.** *Given an EVM program $P = \{b_0, ..., b_n\}$, its addresses equation system $\mathcal{E}(P)$ contains the following equations according to EVM bytecode instructions $b_{pc} \in P$:*

	b_{pc}	C_{pc}
(1)	`JUMP`	$\mathcal{X}_{\sigma(s_{n-1})} \sqsupseteq idmap(\lambda(b_{pc}, \langle n, \sigma\rangle))$ $\forall s \in dom(\mathcal{X}_{pc}), \langle n, \sigma\rangle \in \mathcal{X}_{pc}(s)$
(2)	`JUMPI`	$\mathcal{X}_{\sigma(s_{n-1})} \sqsupseteq idmap(\lambda(b_{pc}, \langle n, \sigma\rangle))$ $\mathcal{X}_{pc+size(b_{pc})} \sqsupseteq idmap(\lambda(b_{pc}, \langle n, \sigma\rangle))$ $\forall s \in dom(\mathcal{X}_{pc}), \langle n, \sigma\rangle \in \mathcal{X}_{pc}(s)$
(3)	$b_{pc} \notin End \wedge b_{pc+size(b_{pc})} =$ `JUMPDEST`	$\mathcal{X}_{pc+size(b_{pc})} \sqsupseteq idmap(\lambda(b_{pc}, \langle n, \sigma\rangle))$ $\forall s \in dom(\mathcal{X}_{pc}), \langle n, \sigma\rangle \in \mathcal{X}_{pc}(s)$
(4)	Otherwise, with $b_{pc} \notin End$	$\mathcal{X}_{pc+size(b_{pc})} \sqsupseteq \tau(b_i, \mathcal{X}_i)$
(5)	Initial opcode b_0	$\mathcal{X}_0 \sqsupseteq \{\langle 0, \sigma_\emptyset\rangle \mapsto \{\langle 0, \sigma_\emptyset\rangle\}\}$

Figure 4.2: Jumping address system equations

where *idmap* returns a map π such that $dom(\pi) = \{s\}$ and $\pi(s) = \{s\}$.

We add an equation for initializing the system: EVM has a unique entry point at instruction b_0, and at this point, the stack is empty. This way, we also prevent the empty solution to be valid.

The constraint equations system has an equation for all program points of the program; only terminal instructions don't have. The main reason is that our constraint equation system represents the flow of the program linking each program point to the next one and *End* instructions have no following instruction.

Following this approach, it is easy to understand why we need to separate rules 1 to 3 from the others. Rule 1 represents an unconditional jump, so the equation links the current set with the branch target. The same happens with rule 2, but in this case we need to separate in two different equations, representing both the possibility of taking or not the jump.

Rule 3 is different from the other two. In this case, the only difference with rule 4 is the usage of function *idmap*.

So, why are we using *idmap* instead of the transfer function for rules 1-4? As we have discussed before, the purpose of using partial maps instead of sets in abstract state definition is to keep a relation between the abstract stack at the beginning of a block to its evolution to the current program point. Equations 1-4 represent those instructions in

which the next one is the beginning of another block. Thus, we need to *reset* the mapping. In order to achieve so, we use the function *idmap*. For each abstract stack contained in $\mathcal{X}_{pc}(s)$ for some abstract stack s, we generate a new abstract state that contains both as key and value $\mathcal{X}_{pc}(s)$ and $\{\mathcal{X}_{pc}(s)\}$ respectively. This way, we ensure abstract states associated to a block contain as keys the corresponding abstract stack at the beginning of that block.

It is also important to remember that we are working under the assumption that jump values are only obtained through `PUSHx`, `DUPx` and `SWAPx` instructions. For those cases the address is obtained from a previously stored value, our reasoning cannot be applied, as the abstract state before a jump instruction doesn't contain a known value for s_{n-1} and thus, no equation is generated. Therefore, we are losing accuracy, and the analysis directly fails when trying to generate this new equation. Nevertheless, by using system, our experiments report a success rate of 99%, meaning we can safely consider this assumption.

We will show how equations are generated with an example:

Example 4.2. *As it can be seen in Figure 3.1, we can jump to block* `103` *from two different blocks:* `73` *and* `14B`*. The computation of the jump equations system will produce the following equations for the entry program points of these two blocks:*

$$\begin{aligned}
\mathcal{X}_{73} &\sqsupseteq \{\langle 2, \{\}\rangle \mapsto \{\langle 2, \{\}\rangle\}\} \\
&\vdots \\
\mathcal{X}_{91} &\sqsupseteq \{\langle 2, \{\}\rangle \mapsto \{\langle 4, \{s_1 \mapsto \texttt{92}, s_3 \mapsto \texttt{103}\}\rangle\}\} \\
\mathcal{X}_{103}{}^{①} &\sqsupseteq \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle \mapsto \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle\}\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{X}_{14B} &\sqsupseteq \{\langle 4, \{s_1 \mapsto \texttt{A9}\}\rangle \mapsto \{\langle 4, \{s_1 \mapsto \texttt{A9}\}\rangle\}\} \\
&\vdots \\
\mathcal{X}_{154} &\sqsupseteq \{\langle 4, \{s_1 \mapsto \texttt{A9}\}\rangle \mapsto \{\langle 7, \{s_6 \mapsto \texttt{103}, s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle\}\} \\
\mathcal{X}_{103}{}^{②} &\sqsupseteq \{\langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle \mapsto \{\langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle\}\}
\end{aligned}$$

Observe that we have two different stack contents reaching the same program point, i.e. *two equations for* $\mathcal{X}_{64B}$ *are produced by two different blocks, the* `JUMP` *at the end of block* `73`*, identified by* $\mathcal{X}_{103}{}^{①}$*, and the* `JUMP` *at the end of block* `14B`*, identified by* $\mathcal{X}_{103}{}^{②}$*. Thus, the equation that must hold for* $\mathcal{X}_{103}$ *is produced by the application of the operation* $\mathcal{X}_{103}{}^{①} \sqcup \mathcal{X}_{103}{}^{②}$*, as follows:*

$$\begin{aligned}
\mathcal{X}_{103} \sqsupseteq\ & \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle \mapsto \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle\}, \\
& \langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle \mapsto \{\langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle\}\}
\end{aligned}$$

Note that the application of the transfer function τ *for all instructions of block* `103` *applies function* λ *to all elements in the abstract state and updates it accordingly*

$$
\begin{array}{rrcl}
(\texttt{JUMPDEST}) & \mathcal{X}_{103} & \sqsupseteq & \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle \mapsto \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle\}, \\
 & & & \langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle \mapsto \{\langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle\}\} \\
(\texttt{PUSH1 00}) & \mathcal{X}_{104} & \sqsupseteq & \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle \mapsto \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle\}, \\
 & & & \langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle \mapsto \{\langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle\}\} \\
(\texttt{PUSH1 0e}) & \mathcal{X}_{106} & \sqsupseteq & \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle \mapsto \{\langle 4, \{s_1 \mapsto \texttt{92}\}\rangle\}, \\
 & & & \langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle \mapsto \{\langle 7, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle\}\} \\
(\texttt{SWAP1}) & \mathcal{X}_{108} & \sqsupseteq & \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle \mapsto \{\langle 5, \{s_1 \mapsto \texttt{92}\}\rangle\}, \\
 & & & \langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle \mapsto \{\langle 8, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle\}\} \\
(\texttt{POP}) & \mathcal{X}_{109} & \sqsupseteq & \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle \mapsto \{\langle 5, \{s_1 \mapsto \texttt{92}\}\rangle\}, \\
 & & & \langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle \mapsto \{\langle 8, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle\}\}
\end{array}
$$

■

We will prove in Chapter 6 that the constraint variables in a solution of the system overestimate the jumping information of the program.

Previous equation system can also be seen as an operator, which given a set of abstract states, propagates the abstract states in these sets following the equations above. It can be proven that $\mathcal{E}(P)$ is a monotone function according to the partial order defined in Section 3.3 . The least solution corresponds to the least fixed point of this operator.

This approach gives us a naive algorithm, which consists in initializing all the constraints variables to the empty mapping $\pi_\perp$ except for $\mathcal{X}_0$, which is initialized to $\{\langle 0, \sigma_\emptyset\rangle \mapsto \{\langle 0, \sigma_\emptyset\rangle\}$ in order to satisfy directly last equation from the jump equation system. Then the values of these variables are iteratively updated following the equations until the least fixed point is reached:

1. Substitute the current values of the constraint variables in the right-hand side of each constraint, and then evaluate the right-hand side if needed;

2. If each constraint $\mathcal{X} \sqsupseteq E$ holds, where E is the value of the evaluation of the right-hand side of the previous step then the process finishes

3. Otherwise, for each $\mathcal{X} \sqsupseteq E$ which does not hold, let E' be the current value of $\mathcal{X}$. Then update the current value of $\mathcal{X}$ to $E \sqcup E'$. Once all these updates are applied, repeat step 1.

Termination is guaranteed since the abstract domain is finite, as there is a finite number of jump target addresses and the stack size is finite.

The reason why we are interested in solving this system is explicitly stated in next theorem:

Theorem 4.1 (Soundness)**.** *Let $P \equiv b_0, \ldots, b_p$ be a program, $\mathcal{X}_1, \ldots, \mathcal{X}_n$ the solution of the jumps equations system of P, and pc the program counter of a jump instruction. Then for any execution of P, there exists $s \in dom(\mathcal{X}_{pc})$ such that $\langle n, \sigma\rangle \in \mathcal{X}_{pc}(s)$ and $\sigma(s_{n-1})$ contains all jump addresses that instruction b_{pc} might jump to during the execution of P.*

As we will prove later, a stronger result can be derived from this equation system. Nevertheless, we are only interested in how jumps are performed to generate the CFG.

Example 4.3. *Let's analyze how to apply this algorithm with our example. At first, we initialize all our equation sets to the empty set, except for $\mathcal{X}_0$, that we have already initialized using the equation*

$$\mathcal{X}_0{}^0 \sqsupseteq \{\langle 0, \sigma_\emptyset\rangle \mapsto \{\langle 0, \sigma_\emptyset\rangle\}\}$$

Hundreds of equations have been generated from our example, so it is not feasible discussing the whole scheme. Instead, we will analyze how $\mathcal{X}_{103}$ evolves through different iterations.

For that purpose, we will denote $\mathcal{X}_n{}^i$ as the contents of set $\mathcal{X}_n$ in iteration i. This notation will be further used to prove completeness in Chapter 6.

First time an equation involving $\mathcal{X}_{103}$ in the left size appears is in iteration 55, as $b_{91} =$ `JUMP`, $\mathcal{X}_{91}{}^{53} = \pi_\perp$ and $\mathcal{X}_{91}{}^{54} = \{\langle 2, \{\}\rangle \mapsto \{\langle 4, \{s_1 \mapsto$ `92`$, s_3 \mapsto$ `103`$\}\rangle\}\}$.

Thus, a new equation is generated

$$\mathcal{X}_{103} \sqsupseteq idmap(\lambda(\texttt{JUMP}, \langle 4, \{s_1 \mapsto \texttt{92}, s_3 \mapsto \texttt{103}\}\rangle))$$

or equivalently,

$$\mathcal{X}_{103} \sqsupseteq idmap(\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle)$$

Then, it is followed that $\mathcal{X}_{103}{}^{55} = \{\langle 3, \{s_1 \mapsto$ `92`$\}\rangle \mapsto \{\langle 3, \{s_1 \mapsto$ `92`$\}\rangle\}\}$. This value is propagated in the following iteration of the algorithm, obtaining $\mathcal{X}_{104}{}^{56} = \{\langle 3, \{s_1 \mapsto$ `92`$\}\rangle \mapsto \{\langle 3, \{s_1 \mapsto$ `92`$\}\rangle\}\}$ and so on.

This value remains unchanged until iteration 99, as we have obtained in the previous iteration that $\mathcal{X}_{154}{}^{98} = \{\langle 4, \{s_1 \mapsto$ `A9`$\}\rangle \mapsto \{\langle 7, \{s_6 \mapsto$ `103`$, s_4 \mapsto$ `155`$, s_1 \mapsto$ `A9`$\}\rangle\}\}$.

$b_{154} =$ `JUMP`, so we can apply equation (1) from our jump addresses system

$$\mathcal{X}_{103} \sqsupseteq idmap(\lambda(\texttt{JUMP}, \langle 7, \{s_6 \mapsto \texttt{103}, s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle))$$

By evaluating this equation and previous one, it is followed that

$$\begin{aligned}\mathcal{X}_{103} \sqsupseteq\ & \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle \mapsto \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle\},\\ & \langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle \mapsto \{\langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle\}\}\end{aligned}$$

$\mathcal{X}_{103}$ is never evaluated again. This means we have reached a fixed point with $\mathcal{X}_{103}$ and $\mathcal{X}_{103} = \mathcal{X}_{103}{}^{99}$.

We can extend this reasoning with each of the equations in our system.

This example may seem trivial at first sight, but the idea of how the algorithm works will prove useful for later proofs. Every abstract state we obtain comes from evaluating a previous abstract state from another solution set. Thus, each abstract state in a solution can be traced back to the iteration it appeared.

■

4.3 Control Flow Graph

Once we have obtained the solution to the equation system, we can now generate the control flow graph of a program.

We define the function $getId(i, \langle n, \sigma\rangle)$ that given the block identifier i and an abstract stack $\langle n, \sigma\rangle$ returns a unique identifier for the abstract stack $\langle n, \sigma\rangle \in dom(\mathcal{X}_i)$.

Example 4.4. *Given the equation:*

$$\mathcal{X}_{\texttt{103}} \sqsupseteq \{\underbrace{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle}_{1} \mapsto \{\langle 3, \{s_1 \mapsto \texttt{92}\}\rangle\}, \underbrace{\langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle}_{2} \mapsto \{\langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle\}\}$$

we have that $getId(\texttt{103}, \langle 3, \{s_1 \mapsto \texttt{92}\}\rangle) = 1$ *and* $getId(\texttt{103}, \langle 6, \{s_4 \mapsto \texttt{155}, s_1 \mapsto \texttt{A9}\}\rangle) = 2$. ∎

Finally, we can define what our CFG is and how it is obtained:

Definition 4.4. *Given an EVM program P, its blocks $B_i \equiv b_i...b_j \in blocks(P)$ and the least solution of the jumps equations system $\mathcal{X}_{pc}$ for all $pc \in P$, we define the control flow graph $CFG = \langle V, E\rangle$, where*

$$V = \{B_{i:id} \mid B_i \in blocks(P) \wedge \langle n, \sigma\rangle \in dom(\mathcal{X}_i) \wedge id = getId(i, \langle n, \sigma\rangle)\}$$

and a set of edges $E = E_{jump} \cup E_{next}$ such that:

$$\begin{aligned} E_{jump} \;=\; \{B_{i:id} \to B_{d:id_2} \mid\; & b_j \in Jump \wedge \\ & \langle n, \sigma\rangle \in dom(\mathcal{X}_j) \wedge id = getId(i, \langle n, \sigma\rangle) \wedge \\ & \langle n', \sigma'\rangle \in \mathcal{X}_j(\langle n, \sigma\rangle) \;\wedge\; d = \sigma'(s_{n'-1}) \wedge \\ & \langle n'', \sigma''\rangle = \lambda(b_j, \langle n', \sigma'\rangle) \wedge id_2 = getId(d, \langle n'', \sigma''\rangle)\ \} \end{aligned}$$

$$\begin{aligned} E_{next} \;=\; \{B_{i:id} \to B_{d:id_2} \mid\; & b_j \neq \texttt{JUMP} \wedge b_j \notin End \wedge \\ & \langle n, \sigma\rangle \in dom(\mathcal{X}_j) \wedge id = getId(i, \langle n, \sigma\rangle) \wedge \\ & \langle n', \sigma'\rangle \in \mathcal{X}_j(\langle n, \sigma\rangle) \wedge d = j + size(b_j) \wedge \\ & \langle n'', \sigma''\rangle = \lambda(b_j, \langle n', \sigma'\rangle) \wedge id_2 = getId(d, \langle n'', \sigma''\rangle) \end{aligned}$$

Each vertex of the CFG corresponds to the combination of a block initial address and an associated stack at the beginning of that block. This way, we guarantee that execution through that node produces a unique final stack, which means there is no way unfeasible paths are introduced in our CFG. It also means that each vertex points to another node, unless last instruction is `JUMPI`. In that case, it will point to two different nodes: one edge will belong to E_{jump} and the other to E_{next}. However, nodes can be pointed by multiple vertex if they share their final abstract stack.

We make an explicit difference between edges: those produced by *Jump* instructions, whose destination block is obtained from the values store before the jump with $d = \sigma'(s_{n'-1})$; and those whose destination block is computed with $d = j + size(b_j)$. In both cases, we join two nodes if the abstract stack associated to the first instruction of the second node corresponds to applying λ to the abstract stack associated to the last instruction in the first node.

Example 4.5. *Considering the blocks shown in Figure 3.1 and the equations shown at Figure 4.2, the CFG of the program includes non-replicated nodes for those blocks that only receive one possible stack state (white nodes in Figure 3.1). We will omit the subindex in these cases. However, the nodes that could be reached by two different stack states (gray nodes in Figure 3.1) will be replicated in the CFG:*

$$\begin{aligned} V \;= \{& B_{\texttt{73}}, B_{\texttt{A0}}, B_{\texttt{11B}}, B_{\texttt{127}}, B_{\texttt{131}}, B_{\texttt{143}}, B_{\texttt{14B}}, B_{\texttt{92}}, B_{\texttt{A9}}, \\ & B_{\texttt{103:1}}, B_{\texttt{159:1}}, B_{\texttt{169:1}}, B_{\texttt{166:1}}, B_{\texttt{117:1}}, B_{\texttt{115:1}}, \\ & B_{\texttt{103:2}}, B_{\texttt{159:2}}, B_{\texttt{169:2}}, B_{\texttt{166:2}}, B_{\texttt{117:2}}, B_{\texttt{115:2}}\} \end{aligned}$$

Analogously, our CFG replicates the edges according to the nodes replicated (solid and dashed edges in Figure 3.1):

$$\begin{aligned} E \;=\; \{&B_{73} \to B_{103:1}, B_{103:1} \to B_{159:1}, B_{159:1} \to B_{169:1}, B_{169:1} \to B_{166:1}, B_{166:1} \to B_{117:1},\\ &B_{117:1} \to B_{92}, B_{A0} \dashrightarrow B_{11B}, B_{11B} \dashrightarrow B_{127}, B_{127} \dashrightarrow B_{131}, B_{131} \dashrightarrow B_{127},\\ &B_{127} \dashrightarrow B_{143}, B_{143} \dashrightarrow B_{169:2}, B_{169:2} \dashrightarrow B_{14B}, B_{14B} \dashrightarrow B_{103:2}, B_{103:2} \dashrightarrow B_{159:2},\\ &B_{159:2} \dashrightarrow B_{169:2}, B_{169:2} \dashrightarrow B_{166:2}, B_{166:2} \dashrightarrow B_{117:2}, B_{117:2} \dashrightarrow B_{155:2}, B_{155:2} \dashrightarrow B_{A9}\} \end{aligned}$$

Note that obtained CFG is a subgraph of the one in Figure 1.2. We are using hexadecimal notation instead, and the numeration of the blocks doesn't match with the one from our analysis: we start using subindexes after a block has been considered at least once before, and numeration starts from 0 instead of 1.

■

Chapter 5

Soundness

The proof sketch follows the next steps:

1. We first define an EVM operational semantics that describes how EVM programs handle jump addresses on the stack. This step was already made in Section 3.3.1.
2. Then we define an EVM collecting semantics for the operational semantics. Such collecting semantics gathers all transitions that can be produced by the execution of a program P;
3. We continue by defining the jumps-to property as a property of this collecting semantics; and
4. Then we prove a lemma that states that the least solution of the set of constraints generated as described in Section 4.2 is a safe approximation of the EVM collecting semantics w.r.t. the jumps-to property.
5. Finally, we rewrite Theorem 4.1 in terms of the operational semantics and prove it.

Definition 5.1 (EVM collecting semantics)**.** *Given an EVM program P and a set of pairs of program states X, the EVM collecting semantics operator $\mathcal{C}_P$ is defined as follows:*

$$\mathcal{C}_P(X) = \{\langle S, S' \rangle \mid \langle \_, S \rangle \in X \wedge S \Rightarrow S'\}$$

The EVM semantics is defined as $\xi_P = \bigcup_{n>0} \mathcal{C}_P^n(X_0)$, where $X_0 \equiv \{\langle \varnothing, \langle 0, \langle 0, \sigma_\emptyset \rangle \rangle \rangle\}$ is the initial configuration.

Definition 5.2 (jumps-to property)**.** *Let P be an IR program, $\xi_P = \bigcup_{n>0} \mathcal{C}_P^n(X_0)$, and b an instruction at program point pc, then we say that $\xi_P \vDash_{pc} T$ if*

$$T = \{\langle n, \sigma \rangle \mid \langle S, S' \rangle \in \xi_P \wedge \langle pc, \langle n, \sigma \rangle \rangle := S'\}$$

The following lemma states that the least solution of the constraint equation system defined in Definition 4.3 is a safe approximation of ξ_P:

Lemma 5.1 (soundness)**.** *Let $P \equiv b_0, \ldots, b_p$ be a program, pc a program point and $\mathcal{X}_0, \ldots, \mathcal{X}_p$ the least solution of the constraint equation system as defined in Section 4.2. The following holds:*

If $\xi_P \vDash_{pc} T$, then for all $\langle n, \sigma \rangle \in T$, exists $s \in dom(\mathcal{X}_{pc})$ such that $\langle n, \sigma \rangle \in \mathcal{X}_{pc}(s)$.

Proof. We use $\mathcal{X}_{pc}^m$ to refer to the value obtained for $\mathcal{X}_{pc}$ after m iterations of the algorithm for solving the equation system depicted in Section 4.2. We say that $\mathcal{X}_{pc}$ *covers* $\langle n, \sigma\rangle$ in $\mathcal{C}_P^m(X_0)$ at program point pc when this lemma holds for the result of computing $\mathcal{C}_P^m(X_0)$. In order to prove this lemma, we can reason by induction on the value of m, the length of the traces $S_0 \Rightarrow^m S_m$ considered in $\mathcal{C}_P^m(X_0)$.

Case base: if $m = 0$, $S_0 = \langle 0, \langle 0, \sigma_\emptyset\rangle\rangle$ and the Lemma trivially holds as $\langle 0, \sigma_\emptyset\rangle \in \mathcal{X}_0^0(\langle 0, \sigma_\emptyset\rangle)$.

Induction Hypothesis: we assume Lemma 5.1 holds for all traces of length $m \geq 0$.

Inductive Case: Let us consider traces of length $m + 1$, which are of the form $S_0 \Rightarrow^m S_m \Rightarrow S_{m+1}$. S_m is a program state of the form $S_m = \langle pc, \langle n, \sigma\rangle\rangle$. We can apply the induction hypothesis to S_m: there exists some $s \in dom(\mathcal{X}_{pc}^m)$ such that $\langle n, \sigma\rangle \in \mathcal{X}_{pc}^m(s)$. For extending the Lemma, we reason for all possible rules in the simplified EVM semantics (Fig. 3.2) we may apply from S_m to S_{m+1}:

- Rule (1): After executing a `JUMP` instruction S_{m+1} is of the form $\langle \sigma(s_{n-1}), \langle n - 1, \sigma\backslash[s_{n-1}]\rangle\rangle$. In iteration $m+1$, the following set of equations corresponding to b_{pc} is evaluated:

$$\mathcal{X}_{\sigma(s_{n-1})} \sqsupseteq idmap(\lambda(b_{pc}, \langle n', \sigma'\rangle)) \qquad \text{for all } s' \in dom(\mathcal{X}_{pc}), \langle n', \sigma'\rangle \in \mathcal{X}_{pc}(s')$$

  where $idmap(\lambda(b_{pc}, \langle n', \sigma'\rangle)) = \pi_\perp[\langle n' - 1, \sigma'\backslash[s_{n-1}]\rangle \mapsto \{\langle n' - 1, \sigma'\backslash[s_{n-1}]\rangle\}]$ (Case (4) in Fig. 4.1). The induction hypothesis guarantees that there exists some $s'' \in \mathcal{X}_{pc}^m$ such that $\langle n, \sigma\rangle \in \mathcal{X}_{pc}^m(s'')$, where $S_m = \langle pc, \langle n, \sigma\rangle\rangle$. Therefore, at Iteration $m + 1$, the following must hold:

$$\mathcal{X}_{\sigma(s_{n-1})}^{m+1} \sqsupseteq \pi_\perp[\langle n - 1, \sigma\backslash[s_{n-1}]\rangle \mapsto \{\langle n - 1, \sigma\backslash[s_{n-1}]\rangle\}]$$

  so $\langle n - 1, \sigma\backslash[s_{n-1}]\rangle \in \mathcal{X}_{\sigma(s_{n-1})}^{m+1}(\langle n - 1, \sigma\backslash[s_{n-1}]\rangle)$ and thus Lemma 5.1 holds.

- Rules (2) and (3): After executing a `JUMPI` instruction, S_{m+1} is either $\langle \sigma(s_{n-1}), \langle n - 2, \sigma\backslash[s_{n-1}, s_{n-2}]\rangle\rangle$ or $\langle pc + size(b_{pc}), \langle n - 2, \sigma\backslash[s_{n-1}, s_{n-2}]\rangle\rangle$, respectively. In any of those cases the following sets of equations are evaluated:

$$\begin{aligned} \mathcal{X}_{\sigma(s_{n-2})} &\sqsupseteq idmap(\lambda(\texttt{JUMPI}, \langle n', \sigma'\rangle)) && \text{for all } s' \in dom(\mathcal{X}_{pc}), \langle n', \sigma'\rangle \in \mathcal{X}_{pc}(s') \\ \mathcal{X}_{pc+1} &\sqsupseteq idmap(\lambda(\texttt{JUMPI}, \langle n', \sigma'\rangle)) && \text{for all } s' \in dom(\mathcal{X}_{pc}), \langle n', \sigma'\rangle \in \mathcal{X}_{pc}(s') \end{aligned}$$

  where $idmap(\lambda(b_{pc}, \langle n', \sigma'\rangle)) = \pi_\perp[\langle n'-2, \sigma'\backslash[s_{n-1}, s_{n-2}]\rangle \mapsto \{\langle n' - 2, \sigma'\backslash[s_{n-1}, s_{n-2}]\rangle\}]$ (Case (4) of the definition of the update function λ in Fig. 4.1). As in the previous case, the induction hypothesis guarantees that at Iteration m there exists $s'' \in \mathcal{X}_{pc}^m$ such that $\langle n, \sigma\rangle \in \mathcal{X}_{pc}^m(s'')$. Therefore, in Iteration $m + 1$, the following must hold:

$$\begin{aligned} \mathcal{X}_{\sigma(s_{n-1})}^{m+1} &\sqsupseteq \pi_\perp[\langle n - 2, \sigma\backslash[s_{n-1}, s_{n-2}]\rangle \mapsto \{\langle n - 2, \sigma\backslash[s_{n-1}, s_{n-2}]\rangle\}] \\ \mathcal{X}_{pc+1}^{m+1} &\sqsupseteq \pi_\perp[\langle n - 2, \sigma\backslash[s_{n-1}, s_{n-2}]\rangle \mapsto \{\langle n - 2, \sigma\backslash[s_{n-1}, s_{n-2}]\rangle\}] \end{aligned}$$

  and thus Lemma 5.1 holds for these cases as well.

- Rules (4) - (12): We will first consider the case in which any of these rules corresponds to an EVM instruction followed by an instruction different from `JUMPDEST`. All rules are similar, as they all use the set of equations generated by Case (4) in Definition 4.3. We will see Rule (4) in detail.

After executing a PUSHx v instruction, S_{m+1} is $\langle pc + size(b_{pc}), \langle n+1, \sigma[s_n \mapsto v]\rangle\rangle$. We have to prove that exists some $s \in dom(\mathcal{X}_{pc+size(b_{pc})})$ such that $\langle n+1, \sigma[s_n \mapsto v]\rangle \in \mathcal{X}_{pc+size(b_{pc})}(s)$. The following set of equations is evaluated:

$$\mathcal{X}_{pc+size(b_{pc})} \quad \sqsupseteq \quad \tau(\texttt{PUSH}x, \mathcal{X}_{pc}) \tag{5.1}$$

By Definition 4.2, $\tau(\texttt{PUSH}x, \mathcal{X}_{pc}) = \pi'$, where $\forall s' \in dom(\pi), \pi'(s') = \lambda(\texttt{PUSH}x, \mathcal{X}_{pc}(s'))$. By the case (1) of the definition of the update function λ, we have that:

$$\forall \langle n'', \sigma''\rangle \in dom(\mathcal{X}_{pc}), \pi'(\langle n'', \sigma''\rangle) = \langle n''+1, \sigma''[s_n \mapsto v]\rangle \tag{5.2}$$

By the induction hypothesis, at Iteration m there exists some $s \in dom(\mathcal{X}^m_{pc})$ such that $\langle n, \sigma\rangle \in \mathcal{X}^m_{pc}(s)$. Therefore, by 5.1 and 5.2, at Iteration $m+1$ we have that the following holds:

$$s \in dom(\mathcal{X}^{m+1}_{pc+size(b_{pc})}) \text{ and } \langle n+1, \sigma[s_n \mapsto v]\rangle \in \mathcal{X}_{pc+size(b_{pc})}(s)$$

and thus Lemma 5.1 holds for Rule (4).

- Rules (4) - (12), followed by a JUMPDEST instruction. After executing any of these instructions, S_{m+1} is $\langle pc + size(b_{pc}), \langle n''', \sigma'''\rangle\rangle$, where $\langle n''', \sigma'''\rangle$ is obtained according to the rule from Figure 3.2. We have to prove that there exists some $s \in dom(\mathcal{X}_{pc+size(b_{pc})})$ such that $\langle n''', \sigma'''\rangle \in \mathcal{X}_{pc+size(b_{pc})}(s)$. The following set of equations is evaluated:

$$\mathcal{X}_{pc+size(b_{pc})} \quad \sqsupseteq \quad idmap(\lambda(b_{pc}, \langle n', \sigma'\rangle)) \qquad \text{for all } s' \in dom(\mathcal{X}_{pc}), \langle n', \sigma'\rangle \in \mathcal{X}_{pc}(s') \tag{5.3}$$

where $idmap(\lambda(b_{pc}, \langle n', \sigma'\rangle)) = \pi_\perp[\langle n'', \sigma''] \rangle \mapsto \{\langle n'', \sigma''\rangle\}]$, where n'' and σ'' are obtained according to the cases of the updating function detailed in Figure 4.1. Using Lemma 4.1, we can see that $s = \langle n'', \sigma''\rangle$ matches the modification made to the state S_{m+1} by the corresponding rule of the semantics. Therefore, at Iteration it holds that $s \in \mathcal{X}^{m+1}_{pc+size(b_{pc})}(s)$, and Lemma 5.1 also holds.

When the algorithm stops Lemma 5.1 holds, as for any pc, $\mathcal{X}^{m+1}_{pc} \sqsupseteq \mathcal{X}^m_{pc}$ for each iteration of the algorithm for solving the equations system of Section 4.2. □

Now we rewrite Theorem 4.1 in terms of the operational semantics of Figure 3.2. This rewriting actually is stronger than Theorem 4.1, as it guarantees the correctness of the stack states obtained from the jumps equations system at any step of the execution.

Theorem 5.1 (Soundness)**.** *Let $P \equiv b_0, \ldots, b_p$ be a program, $S_0 = \langle 0, \langle 0, \sigma_\emptyset\rangle\rangle$ the initial program state, and $\mathcal{X}_1, \ldots, \mathcal{X}_n$ the solution of the jumps equations system of P. Then for any trace $S_0 \Rightarrow^* S_m$, where $S_m = \langle pc, \langle n, \sigma\rangle\rangle$, there exists $s \in dom(\mathcal{X}_{pc})$ such that $\langle n, \sigma\rangle \in \mathcal{X}_{pc}(s)$.*

Proof. Straightforward from Lemma 5.1, as the EVM collecting semantics takes into account all possible traces of the operational semantics. □

Chapter 6

Completeness

In this Section, we are going to prove our algorithm is also complete according to the semantics shown in Fig 3.2. The proof follows these steps:

1. First, we need to prove that every abstract state in each set of the least solution of the jump equation system has been obtained in a finite iteration of the algorithm m from evaluating an abstract state that appeared in the iteration $m-1$.
2. Then we prove that for every value obtained in an iteration m of the algorithm, it exists an m-trace whose final state corresponds to that value and its program counter matches the set index.
3. Finally, we reason that necessarily each value has a finite trace associated, and therefore, all abstract stacks in program states of our collecting semantics w.r.t jumps-to property are contained in a solution.

The first step allows us to have a direct connection between the length of a trace using our semantics and the iteration in which a value appeared using our algorithm. Using this fact and induction reasoning, we will finally obtain the desired results.

Lemma 6.1. *Let $P = \{b_0, ..., b_n\}$ be a program and $\mathcal{X}_{CFG} = \{\mathcal{X}_0, ..., \mathcal{X}_n\}$ the least solution of the constraint equation system defined in Section 4.2 . Let s be an abstract stack which satisfies that $s \in \mathcal{X}_i(s')$ for some abstract stack s' and some index i. Then, the following holds:*

(1) It exists an index m such that

$$s \in \begin{cases} \mathcal{X}_i^m(s') \backslash img(\mathcal{X}_i^{m-1}, s') & if \quad m > 0 \\ \mathcal{X}_i^0(s') & if \quad m = 0 \end{cases}$$

.

(2) If $m > 0$, it exits another abstract stack $s'' \in \mathcal{X}_j(s''')$ for some abstract stack s''' and index j, that verifies

$$s'' \in \begin{cases} \mathcal{X}_j^{m-1}(s''') \backslash img(\mathcal{X}_j^{m-2}, s''') & if \quad m > 1 \\ \mathcal{X}_j^0(s''') & if \quad m = 1 \end{cases}$$

and $\lambda(b_j, s'') = s$.

Proof. (1) In each iteration of our algorithm, we evaluate all the equations in our jump equation system, obtaining all the abstract states that had appeared in previous iterations. Then $\mathcal{X}_i^m \sqsupseteq \mathcal{X}_i^{m-1} \quad \forall m > 0$. A fixed point is reached in a finite number of iterations, thus it exists an index q which verifies that $\forall r > q$, $\mathcal{X}_i = \mathcal{X}_i^r$.

This means that $s' \in dom(\mathcal{X}_i^q)$ and $s \in \mathcal{X}_i^q(s')$. As we have a finite chain $\mathcal{X}_i^0 \sqsubseteq \mathcal{X}_i^1 \sqsubseteq ... \sqsubseteq \mathcal{X}_i^q$, necessarily one of the these two statements hold:

(a) $s \in \mathcal{X}_i^0(s')$.

(b) $s \in \mathcal{X}_i^j(s')$ for some $0 < j \leq q$, but $s \notin img(\mathcal{X}_i^{j-1}, s')$.

If neither (a) nor (b) holds, then $s \in \mathcal{X}_i^q(s')$ implies that $s \in img(\mathcal{X}_i^{q-1}, s')$, as (b) doesn't hold. $img(\mathcal{X}_i^{q-1}, s')$ cannot be empty, so it is followed that $s' \in dom(\mathcal{X}_i^{q-1})$. We are in the same situation as before, but we have decreased the index q. So we can repeat the same reasoning decreasing the index each step, until we eventually reach 0. We have proven that $s \in img(\mathcal{X}_i^0(s'), s') = \mathcal{X}_i^0(s')$, which means that (a) holds. Obviously, we have reached a contradiction.

The lemma is easily followed by setting $m = 0$ if (a) holds or $m = j$ if (b) holds.

(2) As we have proven in (1), let $s \in \mathcal{X}_i^m(s') \backslash img(\mathcal{X}_i^{m-1}, s')$. This mapping has been obtained from an equation of the form $\mathcal{X}_i \sqsupseteq \tau(\mathcal{X}_j, b_j)$ or $\mathcal{X}_i \sqsupseteq idmap(\lambda(b_j, \langle n, \sigma \rangle))$ with $\langle n, \sigma \rangle \in \mathcal{X}_j(\langle n', \sigma' \rangle)$, for some j. In both cases, it exists at least one abstract stack s'' such that $s'' \in \mathcal{X}_j(s''')$, for some $s''' \in dom(\mathcal{X}_j)$, and $\lambda(b_j, s'') = s$. Note that several abstract states may satisfy the condition, if they belong to different equations in the constraint equation system. We are going to prove that at least one of them satisfies (2).

As we have proven in (1), it exists an index k such that

$$s'' \in \begin{cases} \mathcal{X}_i^k(s''') \backslash img(\mathcal{X}_i^{k-1}, s''') & if \quad k > 0 \\ \mathcal{X}_i^0(s'''), & if \quad k = 0 \end{cases}$$

If we assume that $k < m - 1$, by evaluating the corresponding equation, we obtain that $s \in \mathcal{X}_i^{k+1}(s')$. However, as $k + 1 \leq m - 1$, we already know that $\mathcal{X}_i^{m-1} \sqsupseteq \mathcal{X}_i^{k+1}$ and therefore, $s \in \mathcal{X}_i^{m-1}(s')$ or equivalently, $s \in img(\mathcal{X}_i^{m-1}, s')$. We have reached a contradiction.

Now, let us assume that $\forall s''$ which satisfies the condition, the associated index k is greater than $m - 1$. Choosing t as the minimum between all of them, first time the equation for obtaining s is evaluated is the iteration $t+1$. Therefore, $s \notin img(\mathcal{X}_i^t, s')$, and this implies that $s \notin img(\mathcal{X}_i^m, s')$, as $t \geq m$ and $img(\mathcal{X}_i^t, s') \sqsupseteq img(\mathcal{X}_i^m, s')$. Again, this means $s \notin \mathcal{X}_i^m(s')$ and we have reached a contradiction.

From the two previous paragraphs, it is followed that at least one state s'' has an associated index $m - 1 \geq k$ and $m - 1 \leq k$, or equivalently, $k = m - 1$.

□

In the Lemma above, $img(\mathcal{X}_i^{m-1}, s)$ have been used instead of just $\mathcal{X}_i^{m-1}(s)$, because we cannot guarantee that $s \in dom(\mathcal{X}_i^{m-1})$, and therefore, $\mathcal{X}_i^{m-1}(s)$ could be not well-defined.

Lemma 6.2. *Let $P = \{b_0, ..., b_n\}$ be a program, $S_0 = \langle 0, \langle 0, \sigma_\emptyset \rangle \rangle$ the initial program state, an index i which verifies $0 \leq i \leq n$, and $\langle n, \sigma \rangle$ an abstract stack that verifies it exists some abstract stack s*

$$\langle n, \sigma \rangle \in \begin{cases} \mathcal{X}_i^m(s) \backslash img(\mathcal{X}_i^{m-1}, s) & if \quad m > 0 \\ \mathcal{X}_i^0(s), & if \quad m = 0 \end{cases}$$

.

Then, there exists a trace $S_0 \Rightarrow^m S_m$, where $S_m = \langle i, \langle n, \sigma \rangle \rangle$.

Proof. In order to prove this theorem, we reason by induction on the value of m, the first iteration in which $\langle n, \sigma \rangle$ appears in $\mathcal{X}_i(s)$.

Case base: if $m = 0$, then the only abstract stack that belongs to an equation is $\langle 0, \sigma_\emptyset \rangle$, which belongs to $\mathcal{X}_0(\langle 0, \sigma_\emptyset \rangle)$. Therefore, as $\langle 0, \langle 0, \sigma_\emptyset \rangle \rangle = S_0$ and $S_0 \Rightarrow^0 S_0$, the Lemma 6.2 trivially holds.

Induction Hypothesis: we assume that for all abstract stacks $\langle n, \sigma \rangle \in \mathcal{X}_i^m \backslash img(\mathcal{X}_i^{m-1}, s)$, $\forall i, 0 \leq i \leq n$, it exists a trace $S_0 \Rightarrow^m S_m$, where $S_m = \langle i, \langle n, \sigma \rangle \rangle$.

Inductive Case: let us consider an abstract stack of the form $\langle n', \sigma' \rangle \in \mathcal{X}_j^{m+1}(s') \backslash img(\mathcal{X}_j^m, s')$ for another abstract stack s'. In Lemma 6.1, it was proven that it exists an index i and the abstract stacks s and $\langle n, \sigma \rangle$, such that $\langle n, \sigma \rangle \in \mathcal{X}_i^m(s) \backslash img(\mathcal{X}_i^{m-1}, s)$ and $\lambda(b_i, \langle n, \sigma \rangle) = \langle n', \sigma' \rangle$. Then, we can apply the induction hypothesis to $\langle n, \sigma \rangle$, so it exists a trace $S_0 \Rightarrow^m S_m$, where $S_m = \langle i, \langle n, \sigma \rangle \rangle$. Now we reason for all possible opcodes b_i may be:

- b_i = `JUMP`: in this case, from evaluating $\lambda(b_i, \langle n, \sigma \rangle)$, we deduce that $\langle n', \sigma' \rangle = \langle n - 1, \sigma \backslash [s_{n-1}] \rangle$. Besides, we know that the equation associated to b_i is

  $$\mathcal{X}_{\sigma(s_{n-1})} \sqsupseteq idmap(\lambda(b_i, \langle n, \sigma \rangle)) \quad \forall s \in dom(\mathcal{X}_i), \langle n, \sigma \rangle \in \mathcal{X}_i(s)$$

  so by evaluating $\langle n, \sigma \rangle$ in this equation, it follows that $\sigma(s_{n-1}) = j$.

  As b_i = `JUMP`, by applying rule (1) in Fig. 3.2, we obtain that $\langle i, \langle n, \sigma \rangle \rangle \Rightarrow \langle \sigma(s_{n-1}), \langle n - 1, \sigma \backslash [s_{n-1}] \rangle \rangle$. Our previous reasoning has shown that $\langle \sigma(s_{n-1}), \langle n - 1, \sigma \backslash [s_{n-1}] \rangle \rangle = \langle j, \langle n', \sigma' \rangle \rangle$. So by applying the induction hypothesis, we have found a $(m+1)$-trace of the form $S_0 \Rightarrow^{m+1} S_{m+1}$, where $S_{m+1} = \langle j, \langle n', \sigma' \rangle \rangle$.

- b_i = `JUMPI`: in this case, from applying $\lambda(b_i, \langle n, \sigma \rangle)$, we obtain that $\langle n', \sigma' \rangle = \langle n - 2, \sigma' \backslash [s_{n-1}, s_{n-2}] \rangle$. In this case, there are two equations associated to b_i, so we don't know which one of the two has been used to obtain that $\langle n', \sigma' \rangle \in \mathcal{X}_j^{m+1}(s') \backslash img(\mathcal{X}_j^m, s')$. We will study both possibilities:

  Let's assume $\langle n, \sigma \rangle$ has been evaluated in

  $$\mathcal{X}_{\sigma(s_{n-1})} \sqsupseteq idmap(\lambda(b_i, \langle n, \sigma \rangle)) \quad \forall s \in dom(\mathcal{X}_i), \langle n, \sigma \rangle \in \mathcal{X}_i(s)$$

  in order to obtain that $\langle n', \sigma' \rangle \in \mathcal{X}_j^{m+1}(s')$ for some s'. Then, $\mathcal{X}_{\sigma(s_{n-1})}^{m+1} = \mathcal{X}_j^{m+1}$ and this implies that necessarily $j = \sigma(s_{n-1})$.

  On the other hand, if we apply rule(2) in Fig. 3.2, it follows that $\langle i, \langle n, \sigma \rangle \rangle \Rightarrow \langle \sigma(s_{n-1}), \langle n - 2, \sigma \backslash [s_{n-1}, s_{n-2}] \rangle \rangle$. Therefore, $S_m \Rightarrow \langle j, \langle n', \sigma' \rangle \rangle := S_{m+1}$. By applying the induction hypothesis, we have generated $(m+1)$-trace of the form $S_0 \Rightarrow^{m+1} S_{m+1}$.

If we assume that $\langle n, \sigma \rangle$ has been evaluated in

$$\mathcal{X}_{pc+size(b_i)} \sqsupseteq idmap(\lambda(b_i, \langle n, \sigma \rangle)) \quad \forall s \in dom(\mathcal{X}_i), \langle n, \sigma \rangle \in \mathcal{X}_i(s)$$

then necessarily $j = i + size(b_i)$. The reasoning from this point is analogous to the one before, as rule (2) and (3) only differ in the value of the program counter index.

- $b_i \notin Jump \cup End$:

  In this case, we could have applied either the equation

  $$\mathcal{X}_{i+size(b_i)} \sqsupseteq idmap(\lambda(b_i, \langle n, \sigma \rangle)) \quad \forall s \in dom(\mathcal{X}_i), \langle n, \sigma \rangle \in \mathcal{X}_i(s)$$

  or the equation

  $$\mathcal{X}_{i+size(b_i)} \sqsupseteq \tau(b_i, \mathcal{X}_i)$$

  In both cases, it follows that necessarily $j = i + size(b_i)$.

  Now we are going to use the Lemma 4.1, knowing that from the equations above, necessarily $\lambda(b_i, \langle n, \sigma \rangle) = \langle n', \sigma' \rangle$. Therefore, $\langle i, \langle n, \sigma \rangle \rangle \Rightarrow \langle i + size(b_i), \langle n', \sigma' \rangle \rangle = \langle j, \langle n', \sigma' \rangle \rangle := S_{m+1}$. By applying the induction hypothesis, we obtain that $S_0 \Rightarrow^{m+1} S_{m+1}$.

The only cases left are the ones corresponding to $b_i \in End$. However, it is impossible that $b_i \in End$: we have assumed there is an equation of the form $\mathcal{X}_j \sqsupseteq f(\mathcal{X}_i)$, but if we study all the possible cases in Fig. 4.2, it is easily proven that there is no the equation for such b_i, and therefore, $\langle n', \sigma' \rangle$ couldn't have been generated.

It has been proved that every possible $\langle n', \sigma' \rangle$ satisfies that there is a $(m+1)$-trace. Therefore, Lemma 6.2 is followed.

□

From the two previous Lemmas, it can easily be deduced that the constraint equation system is complete, according to our semantics.

Theorem 6.1 (Completeness)**.** *Let $P = \{b_0, ..., b_n\}$ be a program, $S_0 = \langle 0, \langle 0, \sigma_\emptyset \rangle \rangle$ and $\mathcal{X}_{CFG} = \{\mathcal{X}_0, ..., \mathcal{X}_n\}$ the least solution for the constraint equation system defined in Section 4.2. For every abstract stack $s \in \mathcal{X}_{CFG_i}(s')$ for some s', it exists a trace $S_0 \Rightarrow^m S_m$, where $S_m = \langle i, s \rangle$ and $m \geq 0$.*

Proof. From Lemma 6.1, we know there exists an index m which verifies that

$$s \in \begin{cases} \mathcal{X}_i^m(s') \backslash img(\mathcal{X}_i^{m-1}, s') & if \quad m > 0 \\ \mathcal{X}_i^0(s'), & if \quad m = 0 \end{cases}$$

Therefore, by using Lemma 6.2, it is followed that $S_0 \Rightarrow^m S_m$.

□

Completeness is guaranteed, however, this result only applies to the semantics we have defined in Figure 3.2. We already know this semantics over-approximates the real one, thus introducing some traces that cannot be generated in a real execution. However, this is our only of overestimation, thus it is also our only source of losing completeness. Therefore, only paths containing conditional branches in our CFG could correspond to non-real executions in our program, just in case one of the conditions would always evaluate to True or False.

Chapter 7

Implementation

The proposed algorithm has been included in EthIR tool, as we have already discussed. EthIR obtains a RBR representation from the generated CFG and this representation can be later used to infer properties of the EVM bytecode. Gastap uses this representation and infers gas upper bounds. Therefore, in order to assess the impact of our analysis, we will make our experiments using Gastap.

Therefore, we will first present a general overview of Gastap tool in Section 7.1, discussing how each component of its architecture works in order to infer upper bounds. In Section 7.2, implementation details of the analysis discussed in this thesis will be provided. Besides, we are going to discuss the results obtained from the code in Figure 7.3 using Gastap, in order to understand how the Gastap output works.

7.1 Gastap architecture

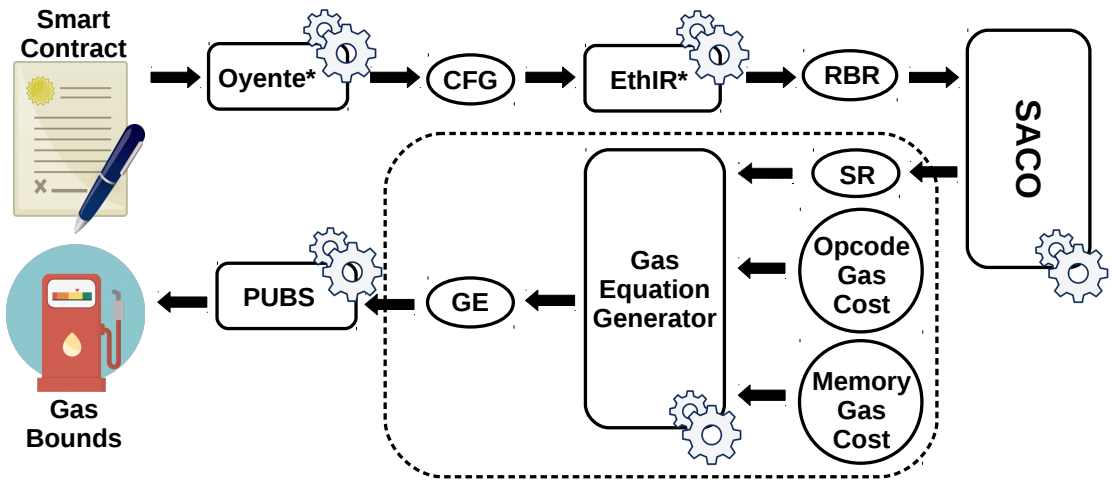


Figure 7.1: Gastap Architecture

In Figure 7.1, the whole Gastap architecture is depicted. This picture shows all intermediate representations obtained through the process. At each step, the output generated from a previous tool and is used as an input for the next one in the flow diagram.

The CFG generator has been implemented on top of the Oyente tool [4]. The aim of Oyente is to find security vulnerabilities and bugs in smart contracts rather than building a CFG of the contract. Hence, the CFG generated by Oyente is not complete and its approach is not sound. Thus, we only use the parser of Oyente, and we have extended it to develop Oyente*. More details of the implementation will be given in Section 7.2.

The EthIR decompiler [1] implements the generation of the high-level *rule-based* representation given a complete CFG of the contract under analysis. For that purpose, each block is translated into a function that receives as parameters stack variables, the memory, the storage and the blockchain data. As we know the size of the stack at the beginning of the block, we present the stack as independent stack variables that we pass onto the function. Inside every function, each bytecode is translated into a high-level rule, using the information we have already gathered from the stack. Finally, an explicit call to the following function is included. This RBR representation is later modified to allow dynamically computing the gas consumption.

Saco [10] is used to generate the cost equations and the size relations needed to obtain the *opcode gas upper-bound* as well as to apply the peak resource analysis used to infer the *memory gas upper-bound*.

EVM gas model is highly complex, so we have divided the gas consumption of each instruction into two parts: *memory gas upper-bound* and *opcode gas upper-bound*. But before understanding the reason behind this distinction, we are going to first understand how cost analysis works.

Cost analysis is a type of static analysis which aims to measure the consumption of a certain resource in a system and give it as a function on the size of the input parameters. Normally, resources are *cumulative*, which means that are only increased through execution, and its upper bound corresponds to the cost related to final state. For instance, the number of calls in an execution is a *cumulative* resource. Cost analysis has been broadly studied, and different generic frameworks based on different approaches have been developed as a result.

However, there are other resources that are acquired and released during the execution. They are known as *non-cumulative* resources. Reasoning about *non-cumulative* resources raises new challenges: the resource consumption upper bound cannot be deduced from the final state of the system, as it could happen on any intermediate step of the execution. For instance, the maximum size a queue can have during an execution.

This cost is known as *peak cost*, and its analysis is known as *peak cost analysis*. A generic framework for performing this analysis can be found in [11].

We would expect gas consumption to be a *cumulative* resource, as it clearly increases for each instruction that is executed. This statement is partially true, because there exists some instructions whose gas consumption depends on *non-cumulative* resources. In order to infer a valid upper bound for gas, we need to make a distinction between the *memory gas cost* and the *opcode gas cost*.

The *memory gas cost* refers to the cost associated to access locations in memory that are beyond previously accessed locations. If an opcode makes such access, then its gas consumption is proportional to the distance to this location. Memory is constantly expanded and reduced through execution, and therefore, we have to perform the *peak cost analysis* to infer valid upper bounds.

The *opcode gas cost* refers to the gas consumption when memory is not taken into account. In this case, we don't deal with *non-cumulative* resources, so we can build the

cost relations from the EVM gas model and apply a standard resource analysis.

The sum of *memory gas upper-bound* and *opcode gas upper-bound* results in a valid upper-bound for gas consumption. As we will study in Section 7.3, if this upper bound is constant, it matches the real gas consumption.

Finally, PUBS [12] is the solver used to solve the cost equations system and to infer the gas upper bound.

7.2 Oyente* implementation

OYENTE is a symbolic execution tool that aims to find vulnerabilities in a smart contract. It takes as an input the EVM bytecode and the global Ethereum state, and it returns the possible vulnerabilities as a symbolic path.

OYENTE is written entirely in Python. It has four main components: **CFGBuilder**, **Explorer**, **Core Analysis** and **Validator**. **CFGBuilder** generates a *skeleton* CFG *i.e* an incomplete CFG that is later used by **Explorer** to execute symbolically from a concrete state. This component uses **Z3** solver, a SMT-solver that can infer whether a conditional branch is probably true or probably false to be taken. This way, it determines which path is more probable and it detects which paths are directly unfeasible, by solving the constraints that have been detected for the condition while generating the CFG. Symbolic traces are generated, so that **Core Analysis** detects which possible vulnerabilities can be found. Finally, **Validator** certifies this decision, filtering possible false positives from previous analysis.

In our case, we are only interested in building our analysis on top of the **CFGBuilder**. In particular, we are just reusing the parser of the bytecode, and also adapting the code in which the *symbolic execution* of EVM bytecode is performed. This way, the impact of bytecodes, that was simulated through the updating function in Section 4.1, is directly adapted from the existing code. Nevertheless, it is important to take into account that this new updating function is much more complex than the one depicted before, as it also records the impact of the bytecode in the memory and the storage. We have changed this information to adapt it to our analysis. For instance, **Z3** solver is no longer used to check unfeasible paths, we just maintain the constraint representation in order to represent unknown stack variables. Most of these changes had already been included in ETHIR [1].

Therefore, we only need to implement the naive algorithm we discussed in Section 4.2. Instead of solving first the constraint equation system and then generate the CFG, we will be generating the CFG implicitly while performing the propagation of values, according to the jump equations.

Before starting the execution, we join the bytecodes that belong to the same block in a structure, so that we can execute all the instructions from a block directly. These blocks are identified by the position of the first bytecode. When a block is copied, a subindex is added to identify the copy. Each block copied has a unique identifier, so in order to assign a new subindex when copying a block, we have included a map that links each original block address with the next index available for copying it. If we are cloning a previously cloned block, we remove its subindex and add the new one available. This has been changed from previous implementation of OYENTE*, that just added a new subindex each time, making it difficult to identify blocks.

For each block generated (copied or not), we are going to maintain a map that links its block address with its associated stack before executing the instructions within the

block. This way, we can identify whether a new block must be generated or link current iteration to a block that had already been generated in our analysis. We will clone the block if its associated stack hasn't been generated before, *i.e* there is no other previous block whose stack is in the same equivalence class with current stack using the same definition we used for the analysis.

We will also simulate memory and storage during symbolic execution. However, we must note that in some cases, we will have to wipe them out, as we are analyzing the contract as a whole, and we cannot keep track of these structures between transactions.

Now, we are going to describe how the cloning and the generation of the CFG is performed. As we have stated before, instead of solving the equation system, we will propagate the abstract states following these equations.

At first, we only consider the block corresponding to address 0 and the initial empty stack. For each block, we execute symbolically all the bytecodes within it and obtain a final stack. Once we reach the end of the block, we can find the following cases:

- Case (1) : we haven't reached a final bytecode. This means next bytecode is the beginning of another block. Therefore, before cloning next block, we retrieve all blocks that are represented by this address (*i.e* the original block and its copies) and compare their associated initial stacks with the final stack from current execution. If another block shares the initial stack, we just update the CFG information by adding an edge from current block to the already generated block. Otherwise, we add a new node to the CFG representing the new cloned block and an edge that points to it from the block before, and repeat the whole process with the new generated block.

  Depending on final bytecode, next block address is obtained in different ways:

  - Last bytecode is `JUMP` : next block address corresponds to the top of the stack before executing `JUMP`.
  - Last bytecode is `JUMPI` : in this case, we have to explore the paths corresponding to a taken branch, or not. Therefore, we recursively explore each of the two cases: we consider as next block address both the jump target address and current program counter plus one.
  - Last bytecode is neither `JUMP` nor `JUMPI`. This means that no branch instruction is performed, and therefore, next block address is the program counter plus one.

- Case (2) : We have reached a final node. The analysis of current path is over, so no further modifications are made.

From this algorithm, we obtain as an output a list of blocks that represent the nodes of the CFG, and their associated edges. If we active *-cfg* flag when invoking *oyente-ethir*, we will produce a .dot file to represent the CFG explicitly, an interesting feature for debugging purposes. The block list is later passed to EthIR, that generates the RBR representation from it.

Example 7.1. *Figure 1.2 corresponds to the CFG of the contract in Figure 3.1. There are two different figures, one with the .dot file when cloning was not considered and the other after using our algorithm. We are going to analyze more in depth what the colors of the node means.*

Function	solc	opcode bound GASTAP	memory bound GASTAP
`addresses(uint256)`	1050	1035	15
`array`	815	800	15
`getAddresses`	∞	1329+292·**nat**(addr-1/32) +75·**nat**(addr+31/32)	6·**nat**(addr)+24+ $\left\lfloor \frac{(6\cdot \mathbf{nat}(addr)+24)^2}{512} \right\rfloor$
`map(address)`	708	693	15
`param`	504	489	15
`sum(uint256)`	∞	$321 + 78 \cdot log_2(2 + 5 \cdot n/4)$	15
`sumConstant`	∞	$369 + 78 \cdot log_2(8.25)$	15
`sumConstructor`	∞	$613 + 78 \cdot log_2(2 + 5 \cdot c/4)$	15
`sumFail(uint256)`	∞	391 + failed(no_rf)	9
`sumFailConstantValue`	∞	2827 + 2/3	15
`sumMapping`	∞	$812 + 78 \cdot log_2(1 + c(\text{maximize\_failed}))$	15
`sumMemory`	∞	$767 + 78 \cdot log_2(1 + c(\text{maximize\_failed}))$	15
`sumMod(uint256)`	∞	$343 + 78 \cdot log_2(2 + 5 \cdot n/7)$	15

Figure 7.2: Gas bounds for `Sums`. Function `nat` defined as `nat(l) = max(0,l)`.

Nodes have different color depending on its type. Green nodes represent terminal nodes, and yellow nodes represent unconditional jumping with a `JUMP` instruction. Red nodes are those blocks that contains no jump instruction, and they are followed by the beginning of another block. Blue nodes represent those nodes whose last bytecode is `JUMPI`. We annotate with a t the edge corresponding to evaluating its condition to True, and with f otherwise.

All the modifications mentioned above can be found in https://github.com/costa-group/EthIR. In particular, the main contributions to the repository have been done in *symexec.py* file, as it is the module in which the symbolic execution is performed. Nevertheless, this repository contains a version with only ETHIR tool, as GASTAP code hasn't been publicly released. An online version of gastap for testing different inputs can be found in https://costa.fdi.ucm.es/gastap/.

Therefore, all commits made for implementing new analysis aren't reflected in this version of the tool. Instead, the contribution statistics can be found in https://github.com/alexcere/EthIR, which is the forked branch in which the analysis was developed. A total of 37 commits have been made to develop the algorithm, generating 8,883 new lines of code and removing 5,249 lines.

7.3 Gas Bounds for Sums Case Study

In this example, instead of analyzing the contract in Figure 3.1, we are going to analyze another different contract we have specifically designed to analyze the strengths and weaknesses of GASTAP tool.

Figure 7.2 shows the comparison between the gas bound provided by **solc**, the Solidity compiler, and in the next two columns the bounds produced by GASTAP for opcode gas and memory gas, respectively, for all public functions in contract `Sums`. Its associated code can be found in Figure 7.3.

First thing we must highlight is that functions with constant gas consumption according to **solc** (`addresses`, `array`...) matches the sum of the gas and memory bounds we

infer with GASTAP. This shows that no precision is lost when analyzing contracts with fixed gas consumption.

In the remaining cases, **solc** cannot determine gas consumption and returns ∞, whereas GASTAP is able to determine this bounds in some of them.

This is really remarkable with function `sumConstant`, that has a fixed upper bound that couldn't be found by **solc**. Before understanding the reasons of this behaviour, we have to understand what `sum` function does. This function receives a `uint` n as a parameter, and contains the following loop:

```
1 sol = 0;
2 uint j = 1;
3 while(sol <= 5*n){
4     j = 3*j + 1;
5     sol = j;
6 }
```

This loop iterates until the value of `sol` is greater than 5n, updating the value of j in each iteration. It is clear that the number of iterations depends directly on the value of n, and therefore, **solc** assumes the gas consumption is ∞. On the other side, GASTAP manages to infer the parametric opcode bound, that clearly depends on the value n. This cost depends on a log_2 function, which is a reasonable upper gas bound, as in each iteration, j grows exponentially. Nevertheless, this is not a trivial estimation: in each iteration, we multiply j by 3, but our bounds depends on log_2 instead of log_3. Besides, inside the logarithm function, we have $5n/4$ instead of $5n$.

Most of the other functions call the `sum` function with values from different structures. `sumConstant` calls with a constant value, thus obtaining a numerical result. This explains why **solc** couldn't obtain this value, as it calls a function with a looping that depends on a parameter.

`sumConstructor` calls it with a public variable (`param`) in the storage that was initialized in the constructor. This value can be changed through a transaction, as we see that we obtain bounds for `param` function. As we cannot guarantee current value when executing a transaction, its bound depends on a parameter that represents it.

`sumMapping` and `sumMemory` use variables that are stored in an map and an array, respectively. We have already noted that persistant information from storage is lost when performing SACO. Therefore, we obtain a *maximize_failed* error when trying to perform the analysis.

In function `sumMod`, we have slightly modified function `sum` above, to analyze how small modifications can affect the bounds obtained. In particular, we have just initialized j to 2 instead of 1. The opcode bound inferred differs from the one obtained before: now we have $5n/7$ instead of $5n/4$. Clearly, this new bound is always smaller than the one obtained before, as we could expect from initializing the loop with a bigger value. This shows our tool greatly adapts its bounds to the information provided, tightening as much as possible this bound.

Finally, we have defined a function called `sumFail`, that intends to generate failed analysis on purpose. At first sight, it doesn't differ much from the ones above:

```
1 function sumFail (uint n) public returns (uint sol){
2     sol = 0;
3     for(uint j = 0; j <= 100; j += n)
```

```
        sol += 10;
    }
```

The main difference is that now we add n to j in each step. This proves fatal for the analysis, as we obtain *no_rf* as a result, that indicated its failure. This is because if n is 0, it theoretically does not terminate.

However, if we now call the contract with a constant value (contract `sumFailConstantValue`), we obtain a valid result. What's the reason of this behaviour? In the first case, we try to add an unknown value to j, crashing when performing the equations. In the second case, this value is known and it isn't modified between iterations, thus generating valid costs.

Note that the results presented in this section do not add the so called *intrinsic gas* cost of the execution as Solidity compiler does. However, GASTAP has a flag to incorporate the transaction fee of 2,300 gas.

```
pragma solidity ^0.4.11;

contract Sums {

uint public param;
uint[] public array;
mapping (address => uint) public map;
address[] public addresses;


 constructor() public {
    param = 3;
    array[0] = 2;
    map[msg.sender] = 4;
 }

 function getAddresses() public view returns (address[]) {
   return addresses;
 }

 function sumConstructor() public returns (uint sol){
    sol = sum(param);
 }

 function sumMemory() public returns (uint sol){
    sol = sum(array[0]);
 }

 function sumConstant() public returns (uint sol){
    sol = sum(5);
 }

 function sumMapping() public returns (uint sol){
    sol = sum(map[msg.sender]);
 }

 function sumFailConstantValue() public returns (uint sol){
    sol = sumFail(3);
 }

 function sum (uint n) public returns (uint sol) {
    sol = 0;
    uint j = 1;
    while(sol <= 5*n){
       j = 3*j + 1;
       sol = j;
    }
 }

 function sumMod (uint n) public returns (uint sol) {
    sol = 0;
    uint j = 2;
    while(sol <= 5*n){
       j = 3*j + 1;
       sol = j;
    }
 }

 function sumFail (uint n) public returns (uint sol){
    sol = 0;
    for(uint j = 0; j <= 100; j += n)
        sol += 10;
 }

}
```

Figure 7.3: Solidity code for `sums` example contract

Chapter 8

Evaluation

In Section 8.1, a brief explanation on how the experiments have been performed is included. Experiments have been separated in two different sections: in Section 8.2, we compare the performance of our analysis with the previous version of Oyente*, and in Section 8.3, we discuss the overall efficiency and effectiveness of Gastap tool including our algorithm.

8.1 Experiments generation

In this subsection, we are going to detail how experiments were carried out. Two type of experiments have been carried out: we have analyzed for which contracts Oyente* fails, and we have also analyzed the overall performance of Gastap tool with our analysis.

Our experimental setup consists on 34,460 contracts taken from the blockchain as follows. We pulled all Ethereum contracts from the blockchain of January 2018 using Etherscan service [7], removed duplicates, and after that, those smart contracts that lead to a compiler error due to a lower version of the compiler.

This process led to obtaining 10,796 files. The whole dataset used can be found at https://github.com/costa-group/EthIR/tree/master/examples/gastap.

In both cases, we have used *Bash scripts* to carry out the experiments. This way, we can execute uninterruptedly the tools for each of our files, classify the files depending on the outcome of the execution and generate log files with the information we need to quantify.

Experiments have been performed on an Intel Core i7-7700T at 2.9GHz x 8 and 7.7GB of Memory, running Ubuntu 16.04. Gastap and EthIR accepts smart contracts written in versions of Solidity up to 0.5.15 or bytecode for the Ethereum Virtual Machine v1.8.18.

The results of Oyente* are discussed in Section 8.2. In this case, we have considered all the files, and the results are expressed in terms of files analyzed instead of contracts. We have considered this approach because there can be some contracts that inherit from other contracts, or can invoke transactions of another contract in the same file. Therefore, dependencies between contracts in the same file can exist, so we don't consider isolated contracts when studying whether the compilation fails or not.

The results of Gastap overall performance can be found in Section 8.3. In this case, we have excluded the files where the decompilation phase fails in any of the contracts it includes, since in that case we do not get any information on the whole file. As a result, we obtain 9,659 files that contain 34,460 contracts. In total, we have analyzed 318,093 public functions (and all auxiliary functions that are used from them).

Type of results	#files old	%files old	#files new	%files new
Files decompiled correctly	9658	89.46%	8737	80,93%
Oyente* error	1102	10,21%	2023	18,74 %
Time out	35	0,32 %	35	0,32%
Unknown error	1	0,01%	1	0,01%
RBR Error	0	0%	0	0%
SACO Error	0	0%	0	0%
Total number of files	10796	100%	10796	100%

Type of Oyente* errors	#files old	%files old	#files new	%files new
Compiler error	1036	9,60 %	1036	9,60%
CFG Generation error	66	0,61%	158	1,46%
Cloning error	0	0%	829	7,68%
Total number of files	1102	10,21%	2023	18,74%

Figure 8.1: (Top) Comparison between old and new analysis results when using EthIR. (Bottom) Reason breakdown for Oyente* errors.

8.2 Oyente* statistics

The results of our analysis in Oyente* are reflected in Figure 8.1. In this figure, we measure the impact our analysis have had in the overall performance.

Columns **#files old** and **#files new** contain the number of files that have been analyzed using the previous version of Oyente* and the version presented in this project. Columns preceded by % represent the impact each possible outcome of the analysis has in the overall dataset.

The number of files that are correctly decompiled have been increased from 8737 to 9658, resulting in a 10.54% improvement. Note that we are comparing only those files that couldn't be analyzed before because an error occurred during the analysis, so previous contracts that were wrongly generated aren't considered. There is also a huge improvement in the number of files that couldn't even generate its CFG: we have gone from 2023 to 1102 (45,53% improvement).

Most of them now correspond to those contracts that cannot be compiled by **solc** compiler, because they correspond to contracts that can only be compiled by newer version of **solc**. By not considering these files in the performance, we obtain that only 1,05% of all the files analyzed fails. This number is smaller than the failure rate of other tools like Vandal [13] (5% of failure rate), Oyente [4] (10% of failure rate) and Rattle [14] (30% of failure rate) or the previous prototype of EthIR [2] (7% of failure rate).

Now we may wonder why our analysis still fails. The files we are interested in inspecting correspond to the 66 files with a CFG generation error. There aren't either many files to consider, so we have studied the error message we obtain from their execution with Oyente* and classified the types of errors:

- *Z3 cannot support too deep recursion*:

  This is the most common error we get among the remaining files. **Z3** cannot handle well recursions when the level is too deep. In those contracts with many different

functions, hundreds of blocks are considered in our analysis. As we keep using the **Z3** representation for unknown stack variables, some of this contracts just fail when generating the CFG.

We decided to keep this representation for a better understanding on the constraints in the conditions in the bytecode. Nevertheless, we are planning to shift to an easier representation of unknown value, as we are not using the SMT-solver for removing possible paths.

- *High level order function*:

  This is the least common error: only two contracts have this error, and both contracts are essentially the same in terms of Solidity code. This is one of the few cases in which the address of a jump bytecode is not obtained through a `PUSHx` instruction directly: the value is previously stored in the memory, loaded from it and applied a mask to compute the jump target address. We cannot keep track of this value, thus reaching a point where analysis just fails.

- *Handling strings*:

  Again, this is really similar to high level order function. We try to store the address before performing the jump, but an error is raised because the memory address where the value is going to be stored is too big for Python to simulate.

- *Non-terminating recursive function*:

  The last case corresponds to a simple recursive function that never ends. In this case, when generating the CFG, we encounter a block that makes an unconditional jump to itself, and pushes several values during its execution. Therefore, no two stacks are the same between the different calls, and the stack will reach its limit of 1024 elements. We haven't taken into consideration this case in our implementation, as in this case, an exception is raised, so the analysis would fail anyway.

8.3 Gastap Statistics

Figure 8.2 shows the final results when analyzing our experiments with Gastap. Columns **#opc** and **%opc** represent the number of functions analyzed for opcode gas bounds, and the percentage of each possible outcome respectively. Columns **#mem** and **%mem** represent the same results for memory gas bounds.

Our results are very promising, as our success rate is 90,24% in case of opcode performance, and 91,95% for memory bounds. In both cases, most of the contracts have a constant gas bound, showing that most contracts contain simple instructions. This behaviour is the one recommended, as we have already discussed, having possible infinite gas consumption leads to *out-of-gas* exceptions.

We also have a considerable number of contracts with parametric gas bounds, which are subject to not be included to blocks in the blockchain due to exceeding the amount of gas allowed for a transaction.

Timeout error has been obtained from those contracts that took more than 30s to analyze. The rest of errors are quite technical due to other intermediate tools used, so we won't go into detail since they are not related to my dissertation. A further discussion will be given in Gastap journal yet to be published.

Type of result	**#opc**	**%opc**	**#mem**	**%mem**
Constant gas bound	266,401	83.75%	274,969	86.44%
Parametric gas bound	20,648	6.49%	17,518	5.51%
Time out	19,935	6.27%	18,086	5.69%
Finite gas bound (maximization error)	9,189	2.89%	7,520	2.36%
Termination unknown (ranking function error)	1,685	0.53%	0	0%
Complex control flow (cover point error)	235	0.07%	0	0%
Total number of functions	318,093	100%	318,093	100%

Phase	$\mathbf{T}_{opcode}$ (s)	$\mathbf{T}_{mem}$ (s)	$\mathbf{T}_{total}$ (s)	**%opc**	**%mem**	**%total**
CFG generation	—	—	20.92	—	—	0.0014%
RBR generation	—	—	1.25	—	—	0.0001%
Size analysis	—	—	132,701	—	—	9.05%
Generation of gas eqs.	175,824	154,529	330,353	11.99%	10.53%	22.52%
Solving gas eqs.	478,506	525,445	1,003,951	32.61%	35.82%	68.43%
Total time GASTAP			1,467,027.17			100%

Figure 8.2: (Top) Statistics of gas usage on the analyzed 34,460 smart contracts from Ethereum blockchain. (Bottom) Timing breakdown for GASTAP on the analyzed 34,460 smart contracts.

As far as efficiency is concerned, results are found in the bottom table in Figure 8.2. In total, the experiments have taken 1,467,027.17 sec (407.5 hours).

It is really significant that our analysis has only taken 0,0014% of the total execution. This fact shows the efficiency of our algorithm, which together with its effectiveness discussed in Section 8.2, proves it can be really useful for other tools that rely on the CFG generation.

Chapter 9

Conclusions and Future Work

Ethereum has become really popular in the last years, becoming its token Ether the second most valuable cryptocurrency, just after Bitcoin. However, we can expect Ethereum to surpass Bitcoin in a near future, as it has a main advantage over Bitcoin: it allows users to deploy dApps on top of it, and program their own smart contracts. There is still many possibilities within smart contracts that haven't probably been considered yet, and we will have to wait some years to see all their potential and possible applications in people's daily life.

Nevertheless, when deploying a contract in Ethereum, we have to be really careful that no vulnerabilities can be triggered when performing a transaction. This is a real danger, and probably the most famous example of it is the *DAO attack*.

The *DAO* implemented a decentralized autonomous organization that let anybody invest Ether in exchange for DAO tokens, so that this Ether could be used for funding in projects that could be voted by the community. An exploit that allowed a user to ask for its investment back multiple times was found by a hacker, allowing them to steal millions of dollars worth Ether. The reason of this bug was simple: the contract returned first the Ether funds to its owner, and then the balance was updated. This episode was heavily discussed in the community, and led to a hardfork resulting in Ethereum Classic to get the Ether back.

One of the main reasons why decentralized technologies are becoming more and more used nowadays is to be able to exchange information knowing that no centralized party can take control over it. If trust over a tool is lost, then the spirit of using it is destroyed, leading to nobody wanting to take part in it.

This is the reason why formal methods are extremely important in this context. Multiple attempts are being made in order to guarantee safety and reliance, and as a result, multiple tools have been developed. Being such a delicate task, we need to ensure that algorithms involved are correct, which can be only achieved by applying formal methods for analysis. This dissertation aims to provide a correct framework that can be useful to develop these algorithms, as control-flow graphs are a necessary tool to reach this goal. Our algorithm presented is simple and easy to implement, yet sound. Besides, having proven completeness with the handling-jumps semantics, gives us which are the cases in which accuracy is lost. Thus, we have control over the cases in which we lose information.

Gastap results in Section 8.3 corroborate the robustness of the analysis: nearly 99% of the total of files considered are correctly decompiled and Gastap manages to correctly infer upper bounds for more than 90% of the functions. We expect these numbers to arise in future developments, when some of the problems addressed in Section 8.2 are finally

overcome. Nevertheless, we cannot expect to have a 100% success rate, as the peculiarities of non-persistent information in EVM code prevent from resolving all the problems.

Further development of this work will be focused in adapting the analysis to newer versions of Solidity code. Solidity is constantly evolving and updating versions, being difficult to maintain an up-to-date tool. Nevertheless, we have already developed a general structure of the analysis, so we would just need to update to newer versions of Solidity.

Bibliography

[1] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In Shuvendu Lahiri and Chao Wang, editors, *16th International Symposium on Automated Technology for Verification and Analysis, ATVA 2018. Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 513–520. Springer, 2018.

[2] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. Running on Fumes: Preventing Out-Of-Gas Vulnerabilities in Ethereum Smart Contracts using Static Resource Analysis. In Pierre Ganty and Mohamed Kaâniche, editors, *13th International Conference on Verification and Evaluation of Computer and Communication Systems, VECoS 2019. Proceedings*, volume 11847 of *Lecture Notes in Computer Science*, pages 63–78, 2019.

[3] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Safevm: A safety verifier for ethereum smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 386389, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Oyente, 2018. Available at https://github.com/melonproject/oyente.

[5] Elvira Albert, Jesús Correas, Pablo Gordillo, Alejandro Hernández-Cerezo, Guillermo Román-Díez, and Albert Rubio. Analyzing Smart Contracts: From EVM to a Sound Control-Flow Graph. Technical report, 2020.

[6] Ethereum. Solidity, 2018. https://solidity.readthedocs.io.

[7] Etherscan. https://etherscan.io, 2018.

[8] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.

[9] Ethereum Virtual Machine Opcodes, 2019. https://ethervm.io/.

[10] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proceedings of 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer, 2014.

[11] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Non-Cumulative Resource Analysis. In *Proceedings of 21st International Conference on Tools and Algorithms*

for the Construction and Analysis of Systems, TACAS 2015, volume 9035 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2015.

[12] 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, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2008.

[13] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A Scalable Security Analysis Framework for Smart Contracts, 2018. arXiv:1809.03981.

[14] Rattle - an evm binary static analysis framework, 2018. https://github.com/crytic/rattle.