Integrating the EVM super-optimizer *gasol* into *real-world* compilers

Integración del superoptimizador de EVM *gasol* en compiladores de Ethereum

Universidad Complutense de Madrid
Facultad de Informática

Máster en Métodos Formales en Ingeniería Informática

Alejandro Hernández Cerezo

Dirigido por:
Elvira Albert Albiol
Albert Rubio Gimeno

Curso 2020-21
Convocatoria: *Junio-Julio 2021*
Calificación: *10*

Abstract

Smart contracts are programs deployed and executed on the blockchain for a monetary fee paid in *gas* and thus smart contract compilers have a clear optimization target: gas usage. Because smart contracts are a young, fast-moving field for which (manually) fine-tuned compilers have not yet been developed, they highly benefit from automated and adaptable approaches. Moreover, smart contracts are effectively immutable, and as-such need a high-level of assurance, which makes them an ideal domain for applying formal methods to them.

Super-Optimization is a technique, proposed by the formal methods community, which attempts to find the best translation of a block of instructions by trying all possible sequences of instructions that produce the same result.

This work presents a framework for super-optimizing smart contracts based on Max-SMT with two main ingredients:

1. the extraction of a *stack functional specification* from the basic blocks of a smart contract, which is simplified using rules that capture the semantics of arithmetic, bit-wise, and relational operations, and

2. the *synthesis of optimized blocks*, which by means of an efficient SMT encoding, finds the bytecode blocks with minimal gas cost whose stack functional specification is equal (modulo commutativity) to the extracted one.

This framework was already implemented in a tool called syrup 1.0, which only considered the necessary encodings to perform super-optimization. This Master Thesis presents an improved version of that tool, called gasol (GAS Optimization tooLkit), which includes more advanced encodings of the problem to speed up the process of super-optimization. Large-scale experiments have been performed to determine realistic settings for including gasol in an EVM compiler and investigate trade-offs between quality of optimizations and required optimization time. Besides, verification techniques have been developed to ensure the process is indeed correct.

Our experimental results indicate these new encodings are useful, and by limiting the timeout of the tool, it is feasible to include gasol into an EVM compiler. In fact, *GASOL* project has been funded by the Ethereum Foundation and they are considering integrating this approach into following versions of the **solc** compiler.

The main results in this project have been included as part of the article submitted to the *ACM Transactions on Software Engineering and Methodology (TOSEM)* for its publication.

Keywords:

Blockchain, Super-Optimization, Verification, Dafny, Ethereum, Formal Methods, Gas, Max-SMT, Smart Contracts .

Resumen

Los contratos inteligentes son programas que se despliegan y ejecutan en una cadena de bloques, pagando una tarifa monetaria en términos de una unidad computacional llamada *gas*. Por tanto, los compiladores de contratos inteligentes tienen una función objetivo clara: el consumo de gas. Dado que el desarrollo de contratos inteligentes es un campo de estudio relativamente novedoso y cambiante, aún no se han desarrollado compiladores óptimos que se puedan beneficiar de técnicas automáticas. Es más, al ser estos contratos inmutables, necesitan ser comprobados de forma minuciosa antes de ser desplegados, lo que los convierte en candidatos ideales para la aplicación de métodos formales.

La *super-optimización* es una técnica propuesta por la comunidad científica de métodos formales que consiste en hallar la mejor traducción de un bloque de instrucciones a partir de probar exhaustivamente todas las secuencias de instrucciones que produzcan el mismo resultado.

Este trabajo presenta un sistema para optimizar contratos inteligentes basada en la técnica Max-SMT, con dos componentes principales:

1. la extracción de una *especificación funcional de pila* a partir de los bloques básicos de un contrato inteligente, la cuál se puede simplificar a partir de reglas que capturan la semántica de las operaciones aritméticas, de bit y de las relaciones entre distintas operaciones.

2. la *síntesis de bloques optimizados*, la cuál se expresa en términos de una codificación SMT eficiente de tal forma que se consiga encontrar la secuencia de *bytecodes* que consume menos *gas* cuya *especificación funcional de pila* concuerda con la inicial, teniendo en cuenta la conmutatividad de las operaciones.

Este sistema ya fue implementado en una herramienta llamada syrup 1.0, la cual se limitaba a considerar únicamente la codificación necesaria para llevar a cabo el proceso de *super-optimización*. Este trabajo presenta una versión mejorada de esa herramienta, a la cuál hemos denominado gasol (GAS Optimization tooLkit). Esta nueva versión considera codificaciones más avanzadas que permiten acelerar el proceso. Se han llevado a cabo numerosos experimentos que han permitido descubrir configuraciones realistas para integrar esta herramienta en un compilador de EVM. Estos experimentos también se han centrado en determinar la relación entre distintas codificaciones y el tiempo que tarda en realizar el proceso. Además, se han desarrollado técnicas de verificación para probar que la transformación es correcta.

Los resultados experimentales muestran que las nuevas codificaciones propuestas son útiles a la hora de abordar el problema, y que a través de limitar el tiempo máximo de ejecución de la misma, se consiguen tiempos plausibles para la integración en compiladores de EVM. De hecho, la *Fundación Ethereum* ha financiado el proyecto *GASOL* y están considerando incluirlo en nuevas versiones de su compilador.

Los resultados principales de este trabajo de fin de máster se han incluido en el artículo que hemos enviado a la revista *ACM Transactions on Software Engineering and Methodology (TOSEM)* para su publicación.

Palabras Clave:

Cadena de bloques, Super-optimización, Yul, Verificación, Dafny, Ethereum, Métodos Formales, Gas, Max-SMT, Contratos inteligentes.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation of this Master thesis

The Ethereum Virtual Machine (EVM) is a stack machine that enables 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 master thesis is to study in detail and improve the performance of an already existing tool: syrup 1.0 [8]. syrup 1.0 is a framework that optimizes stack-related operations by applying semantic simplification rules and finds the optimal sequence of operations when considering basic stack-related opcodes. As such, the main basis of this thesis relies on a very strong previous work, being syrup 1.0 a complete tool by its own. Then, why dedicating a whole master thesis to improve an already *functional* tool?

Many tools developed in academia just serve as proof-of-concept systems that prove certain approach has a potential for being used by showing its gains performing some empirical experiments. And, then, there is usually no real motivation to improve these tools, unless it provides more material for publishing new articles.

In this thesis, the usual way of proceeding is reversed: instead of justifying certain analysis/approach is worth by performing experiments, our motivation relies on trying different approaches until we determine the optimal one for the real use of the approach. Thus, obtaining promising experimental results becomes explicitly our *goal* instead of the *justification* of the approach.

And not only performance is our goal, but rather all issues that are related to making a tool functional:

1. Is our tool reliable?

2. How can we integrate it into *real-world* tools?

At first glance, it may seem that some of these issues have more to do with Software Engineering than Formal Methods and Analysis. However, as we will discuss throughout this thesis, formal methods fit perfectly and are fundamental in order to deal with these questions.

This approach also leads to an unusual structure in the thesis: instead of presenting a main analysis or result that is justified through the different chapters, different issues are tackled using different approaches that are related by sharing a common starting point, but using different techniques.

In fact, all different issues can be subsumed in one final goal: integrating gasol into an EVM compiler. The goal *per se* is probably the least interesting aspect of this Master thesis from a theoretical point of view. Nevertheless, it serves as a hidden purpose that drives the rest of the thesis from the background.

This project is currently being funded by the Ethereum Foundation. GASOL: GAS Optimization tooLkit (also denoted simply as gasol or syrup 2.0) is planned to be added into Solidity's compiler optimizer, considering interactions between the current optimizer and gasol leads to better results than current state-of-the-art. Nevertheless, Ethereum's developers have expressed their strong interest in having a tool that allows proving optimal configurations of instructions so they can enhance their current optimizer.

These remarks have been introduced in this section so that the reader can understand the goals of this thesis, have a broad overview of it and find it easier to follow the structure of the thesis.

1.2 Introduction

While Bitcoin [34] paved the way for cryptocurrencies and the popularity of blockchain technologies, Ethereum [43] elevated their potential by allowing developers to run decentralized applications—so called *smart contracts*—on top of the Ethereum blockchain. Today, many blockchains come with a mechanism to run smart contracts, *e.g.* Facebook's blockchain [28], Tezos [18], or Zilliqa [41]. Basically, a smart contract is a program, whose current state and source code are stored on the blockchain. When invoked, decentralized *miners* execute the smart contract and update the contract state. Through this decentralization the state and smart contract cannot easily be adversely manipulated, making them effectively *immutable* without consensus of the miners and thus they have the potential of facilitating legal, financial and social agreements, e.g., to fulfill employment contracts or to execute bets and wagers. For this to work, smart contracts have to be "correct", and work on reasoning about correctness of smart contracts [26], even repairing them automatically [45], has emerged.

Smart contracts are usually written in a Turing-complete programming language. To prevent an adversarial participant from wasting miners computational resources—up to exploiting non-termination—every execution of a smart contract is subject to a monetary fee paid by the caller. That is, to call a smart contract the caller allots an amount of *gas* to carry out the execution. If the transaction exceeds the allotted gas limit, an *out-of-gas* exception is raised, interrupting the current execution and restoring the previous state. The fee is lost to the caller. In Ethereum, gas is priced in a sub-unit of the cryptocurrency—in *gwei*—a sub-unit of its *Ether*. The *gas model* is specified in Ethereum's yellow paper [43] and gives a precise definition of the gas consumption for each executed instruction. The cost of an execution in Ethereum ranges from cents to a few dollars, except in certain peak periods where it has been ten or even a hundred times more. In order to provide an idea of the impact, we have estimated that the money spent on transactions (excluding the intrinsic gas cost) from 2017 to 2019 is around 157 Million dollars[1].

Smart contracts are also usually written in a high-level language and compiled to a low-level language, which is then deployed on the blockchain. In Ethereum the most

[1]The data is taken from [1] using the gas spent by transactions and the average *gwei* and Ether exchange rate per day.

popular high-level language is Solidity [17], which by the Solidity compiler `solc` is compiled to Ethereum bytecode to be executed on the Ethereum Virtual Machine (EVM), a simple stack-based machine specified in [43]. Now there are several aspects that make smart contract compilers special, and that motivate our work, they

1. are a young, fast-moving field without years of manual fine-tuning and therefore require automated and adaptable approaches,
2. are immutable, deployed only once, and therefore longer compile times may be acceptable,
3. need a high level of assurance, which makes them ideal for applying formal methods, and
4. have a clear cost model and optimization target: gas usage.

Reducing gas costs of smart contracts is a problem of utmost relevance in the blockchain ecosystem, as there are normally between half a million and a million transactions a day. And indeed, the Solidity documentation [17], and posterior documents (e.g. [12, 32, 11]), identify gas-costly patterns and propose replacements with gas-efficient ones. Compilers also optimize the bytecode for minimizing its gas consumption. For example, the flag `optimize` of the `solc` compiler optimizes storage of large constants, the `dispatch` routine and replaces some EVM bytecode sequences with the goal of saving gas. In addition, the newer versions of the compiler include a more sophisticated optimization process. They are able to apply simplification rules based on the semantics of the EVM instructions and infer more complex properties such as dead code, constant addresses, or redundant expressions that are used to modify the bytecode. Still, even when the guidelines are followed and the `optimize` flag is used, the compiled EVM bytecode is not always as efficient as possible. Moreover, adopting these guidelines requires deep understanding of EVM instructions and the gas consumption for the different operations.

Our goal is to develop a generic and adaptable framework to optimize smart contracts, where we focus on optimizations of *basic blocks*, i.e., sequences of instructions without making any change in the control flow. We therefore look at *super-optimization* [27]—a technique proposed over 30 years ago which attempts to find the best sequence of instructions by using exhaustive search to try all possible sequences of instructions that produce the same result. Super-optimization has been used in compiler optimizations, e.g., for LLVM [23] with Souper [39] and recently on Ethereum bytecode [33]. As an exhaustive search problem, the technique is computationally extremely demanding, which is confirmed by the experimental results in [33]. This shows that finding optimal code for a basic block still remains very challenging.

Framework. Figure 1.1 gives an overview over our framework. The input is a *smart contract* (in Solidity or EVM bytecode), and the output is an optimized *smart contract'* that uses less gas for execution. Our goal is to find it with a reasonable amount of resources. To generate the optimized *smart contract'*, we first 1. generate a *control-flow graph* (CFG) from the initial *smart contract*, and then 2. split the blocks of the CFG into smaller *sub-blocks*. Here we split on instructions manipulating state, such as store or log instructions, because those cannot be simply reordered and their result does not modify the stack. An example of a (split) block can be found in Figure 2.1. A block is the basis for our first major data structure 3. the *stack functional specification* (SFS).

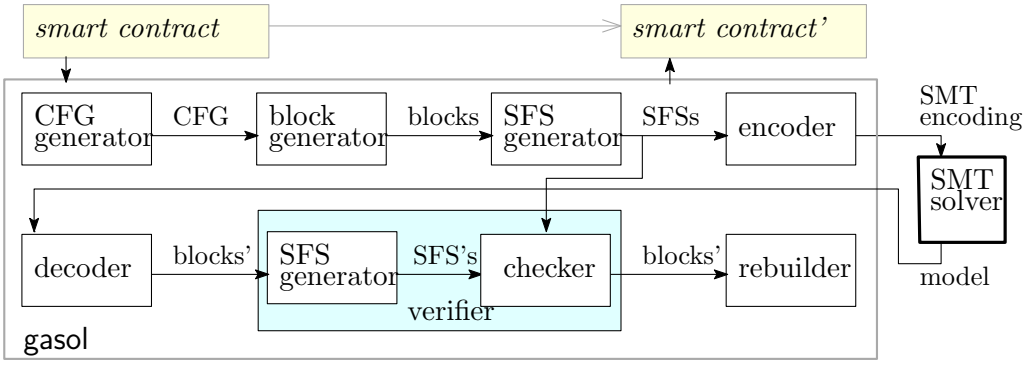

Figure 1.1: Components and interfaces of gasol.

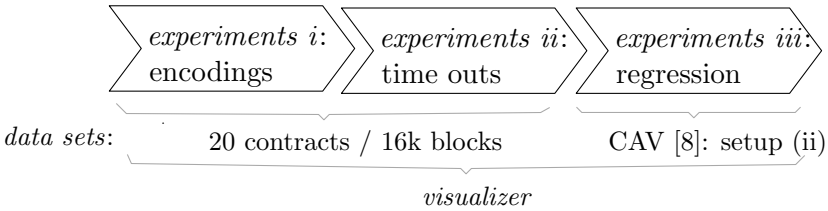

Figure 1.2: Overview over the experiments.

The SFS is considered as our *intermediate representation* and specifies the input and target operational stacks for each block. We compute the SFS by symbolic execution [29]. Hereby, we also apply *simplification rules* capturing the semantics of instructions exploiting algebraic identities. The SFS is the *specification* of our block and we will re-use the SFS in the *verifier*. Section 2 gives technical details of the SFS. Once the SFS is computed for all the blocks, 4. we *encode* the synthesizes of optimal EVM bytecode as a Max-SMT problem. This *Max-SMT encoding* is our second major data structure. Section 3 gives an overview of the encoding and extensions and omits technical details as they can be found in [8]. With this Max-SMT encoding we 5. call an external *Max-SMT solver* as a black-box. In our implementation, we leverage Z3 [15], Barcelogic [9], and MathSAT [14]. If successful, 6. the SMT solver returns a *model*, which we *decode* to get a synthesized *block'*. Here *block'* may be an optimization of the original *block*. It may also be the same block and the solver has shown it to be optimal. In the first case, we 7. verify *block'* is adhering to its specification, *i.e.* the SFS. Therefore we (a) generate an SFS' from the optimized *block'* and (b) *check* that SFS and SFS' are equivalent. This verification step is optional, but gives an additional guarantee of correctness, and details are in Section 4.1. Finally, we 8. *rebuild* the blocks into an optimized *smart contract'*.

Experiments. We implemented our framework for optimizing EVM bytecode in our tool gasol. Our goal is towards integration of our framework into a real-world smart contract compiler, which requires careful consideration of resources to arrive at *smart contract'* in Figure 1.1. Hence, we conducted a range of experiments to improve the performance of our tool and balance trade-offs. Figure 1.2 shows an overview of the experiments in Section 6. To interpret our experiments, we have implemented a *visualizer* and the results are available to be viewed at `http://costa.fdi.ucm.es/syrup-visualizer`.

The goal of the experiments is threefold: *experiments i* (performed on 20 contracts and 16k blocks) aim at finding the best *SMT encoding* from Section 3 and [8]. Therefore we add constraints to aid the solver in finding a solution. Here, the added constraints are

not necessary to specify the problem, but at least one (optimal) solution has to satisfy them. We further balance between simpler but larger constraints versus smaller but more intricate constraints. We compare encodings which differ by

1. adding constraints every solution must satisfy, e.g., constraints that encode that every numerical value in the target stack must be pushed at least once;

2. adding constraints at least one *optimal* solution must satisfy, e.g., to indicate that there cannot be instructions that introduce an element on the stack before a `POP` instruction. Finally, we have

3. different constraints for the gas. Here we either

   (a) *directly* add weight for every chosen instruction in the solution, or

   (b) *group* all instructions by gas cost, and add a delta of gas cost whenever an instruction from a more expensive group is chosen.

Once we found the best encoding, the goal of *experiments ii* is to find the best *timeout* using the same benchmark set as in *experiments i*. The trade-off between finding a good solution and giving time to find this solution is sensitive. If no solution is found, the timeout given is the upper bound of the computation. Thus increasing the timeout has a large effect on the overall time spent on optimizing a contract, even if for many individual blocks the timeout does not matter. Our experiments are with a timeout of 1 sec, 10 sec, 15 sec, 30 sec, and 60 sec. Finally, the goal of *experiments iii* is to make regression tests with the experiments described in [8]. Our comparison looks at the time spent and the quality of solutions.

Contributions. We propose a generic, novel, and realistic framework for *gas optimization* of smart contracts which is based on synthesizing super-optimized basic blocks using SMT solvers and carefully estimating the trade-off between optimization and compile time. Features that distinguish our work from previous approaches, that attack the same or a similar problem [33, 20] and the conference version that preceeded this thesis [8] are:

1. *Stack functional specification (SFS).* We introduce SFSs to specify input and target stack computed by symbolic execution as basis for an SMT encoding to synthesize optimal stack operations. To capture a great part of the *semantics* of the arithmetic, bit-wise, relational, etc. operations we use *simplification rules*.

2. *Synthesis problem using SMT.* We view optimization as a synthesis problem in which a Max-SMT solver is used to synthesize optimal basic blocks which, for the input stack given in the SFS to find a solution, i.e., a sequence of instructions, which produces the target stack.

3. *Verify synthesis.* We verify correctness of the found solution with the original SFS. This gives us a high level of assurance that our encoding and found solution is correct. Occasionally even SMT solvers may have bugs *e.g.* Souper found a bug in Z3 [39]. The equivalence algorithm has been verified using Dafny [24].

4. *Towards application in real-world compilers.* We extend syrup 1.0 from [8] to gasol to facilitate integration with a real-world smart contract compiler and experiments

to determine performance improvements and trade-offs between optimizations and compile time. The reconstruction of the smart contract allows analysis on impact of optimization on whole contracts and allows integration with other optimizations.

5. *Experiments.* We report on and leverage our implementation gasol for experiments to evaluate the performance of different encodings, determine the best time outs, and compare with previous approaches in [8, 33].

This thesis is an extended and improved version of the conference paper published in the proceedings of CAV'20 [8], which has been submitted to the journal TOSEM. We make the following further contributions w.r.t [8]: For (1), we have extended the SFS with more simplification rules from different sources and verified them with [40]. Besides, we have proposed an abstract semantics in order to formalize the process of *symbolic execution*. For (2), we have developed different constraints to improve performance. We have added a new phase (3), the verification of SFS, to provide a higher level of assurance. For (4), we have extended syrup 1.0 to gasol, which enabled us to perform for (5) new experiments for different encodings and trade-offs between optimizations and compile time.

1.3 Structure of the thesis

The chapters in this thesis are organized as follows:

2. Chapter 2 introduces a brief overview on EVM and the notion of *sub-block*. From this notion, the definition of *Stack Functional Specification* is derived. Finally, an optimization step on the SFS based on the semantics of certain EVM instructions is performed. This optimization is based on the application of semantic rules simplification. The complete list of rules can be found in Appendix A.

3. Chapter 3 focuses on the generation of a Max-SMT problem to produce an optimal equivalent *sub-block*. Firstly, the inference of the number of instruction in a block and the maximum stack size is discussed. Then, a refinement of the *Stack Functional Specification* useful for the SMT-generation is introduced: the *Abstract Functional Specification*. After that, an explanation on how to model the stack using quantified variables is discussed. Finally, the Max-SMT basic encoding is introduced, as well as some additional constraints that are useful for improving solver's performance. The complete encoding is fully described Appendix B. Theorems proving the correctness of the additional encoding can be found in Appendix C.

4. Chapter 4 explains in full detail the algorithm that compares the equivalence between *sub-blocks*. A full explanation on its verification in Dafny is also provided.

5. Chapter 5 focuses on the implementation details. This section includes an explanation on the exchange format used among different phases of the optimization process, as well as the produced output. Besides, a case study is included for a *sub-block*. Finally, implementation details of the gasol visualizer are discussed.

6. Chapter 6 introduces the experimental results, following the structure described in previous section.

7. Chapter 7 describes other similar approaches to optimization of smart contracts and super-optimization in general.

8. Chapter 8 concludes the thesis by pointing out the planned future of this Master thesis.

Chapter 2

Stack Functional Specification

In this section we present a functional description of the stack. The *stack functional specification* (SFS) expresses the stack after the execution of a sequence of EVM bytecodes in terms of the elements that were already in the stack. It can be seen as our *intermediate representation* that is used (i) to describe the state of the stack after the execution of each of the blocks that will be optimized, (ii) to apply a preliminary optimization phase based on the semantics of the bytecodes, (iii) to generate the SMT encoding that allows us to synthesize the optimal blocks, and (iv) to verify that the optimize block obtained complies with the original specification, *i.e.* it produces the same stack.

Section 2.1 describes the key aspects of the EVM that will be needed for the remaining sections. Section 2.2 describes the *stack functional specification* and overviews how it can be generated. Section 2.3 shows a subset of the simplification rules used to optimize the SFSs.

2.1 EVM overview

In this section, we discuss the main aspects of the EVM [43] relevant to this master thesis.

The EVM is the virtual machine used by Ethereum in order to run smart contracts. It is a stack-based virtual machine which has its own set of low-level instructions, denoted as EVM bytecode. Programmers usually do not have to work at this level: smart contracts are normally written in a high-level language that is transformed into this representation. The most used ones are Solidity, Vyper or Yul.

The EVM works with three different structures: a *stack* in which basic operations are performed, a volatile memory region denoted as *memory* and a persistent memory region denoted as *storage*. The explicit separation between both memory regions allows programmers to use smart contracts' resources wisely, as smart contracts have an expensive computational cost associated. *Storage* has a much higher computational cost associated, due to its contents being stored within the blockchain.

The *stack* is used for performing all operations. It contains 256-bit words, and its maximum size is limited by 1024 elements. Every operation extracts its operands at the top of the stack and then stores the result on top of it. *Memory* and *storage* are managed through stack operations. Jump instructions also retrieve the jump address from the top of the stack at the moment of execution, which leads to dynamic jump instructions. There are some low-level specific opcodes that are used to manage the stack: `PUSHx v`, `DUPx`, `SWAPx` and `POP`.

In order to avoid non-terminating executions and to measure the computational effort that deploying a contract or committing a transaction has in the system, a unit called *gas* is introduced in the blockchain ecosystem. This unit has no fixed monetary value: when submitting a transaction, the proposer associates a monetary price per gas in terms of the *ether* cryptocurrency. Usually, this price is expressed in terms of smaller units of *ether*, such as the *gwei*.

Each block included in the blockchain cannot exceed a certain gas limit, which in the last included blocks nearly reaches 15 million units of gas. The less amount of gas consumed by a transaction, the more transactions can be added to the block and the less money is spent per transaction. Thus, *gas* optimization has a clear positive impact on the overall network.

Gas execution from a deploying a contract or proposing a transaction is measured in terms of a *gas* model in terms of EVM opcodes, which is described in [43]. Most opcodes have a fixed gas cost associated, whereas there are some opcodes whose gas cost may depend on variable parameters, such as the regions of memory that are occupied at the moment of the execution.

Memory and storage opcodes consume much more gas compared to stack ones. However, they can be seldom optimized due to the fact that we can only get rid of them by detecting unnecessary accesses to them. That's why this thesis focuses on the optimization of stack related variables, and considers memory opcodes as fixed ones.

Another key aspect when considering optimizing EVM opcode is the size of the bytecode of the smart contract. Once smart contracts are deployed, their associated bytecode is stored in the blockchain. This way, their associated methods can be invoked through transactions. Only the code from the `constructor` is not stored: it is executed only once when the contract is deployed.

Even though the optimization goal of this approach is *gas* and blocks are limited by gas consumption, it is also necessary to take into account the length of the produced bytecode. In many cases, small *gas* optimizations that lead to bigger code are not worthy.

2.2 From EVM bytecode to a Stack Functional Specification

As shown in Figure 1.1, the first step of our framework is the generation of the *sub-blocks* that will be analyzed. We consider as our starting point the basic blocks obtained from the control-flow graph (CFG) of the EVM bytecode to be optimized:

Definition 2.1 (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

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

Basically, these basic blocks represent fragments of code that are always executed straightforward. The initial instruction from these blocks corresponds either to the first

```
1  PUSH1 0X03
2  SLOAD
3  PUSH1 0X40
4  MLOAD
5  PUSH1 0X01
6  PUSH1 0XA0
7  PUSH1 0X02
8  EXP
9  SUB
10 DUP1
11 DUP5
12 AND
13 SWAP3
14 AND
15 SWAP1
16 PUSH1 0X8B
17 SWAP1
18 PUSH1 0X00
19 SWAP1
20 LOG3
21 PUSH1 0X03
22 DUP1
23 SLOAD 3
24 PUSH2 0xFFFF
25 NOT
26 AND
27 PUSH1 0x01
28 PUSH1 0xa0
29 PUSH1 0x02
30 EXP
31 SUB
32 SWAP3
33 SWAP1
34 SWAP3
35 AND
36 SWAP2
37 SWAP1
38 SWAP2
39 OR
40 SWAP1
41 SSTORE
```

Figure 2.1: CFG block of contract SimpleChildToken

instruction of the program, a `JUMPDEST` opcode or an opcode whose previous opcode is a jump instruction. The final instruction corresponds to the last instruction of the program, an stop opcode or an opcode whose following opcode is a jump destination.

Note that there exist several tools such as EthIR [7], Mytril [30], Rattle [2], or Madmax [19] that can compute it. As we are only focused on those EVM instructions that operate on the stack, there are bytecode instructions such as `LOGX`, `MSTORE` or `CALLDATACOPY` that we do not optimize. Hence, we use these instructions, whose effects are not reflected on the stack and cannot be reordered, to split the basic blocks into *sub-blocks* that will be optimized.

Definition 2.2 (block-partitioning)**.** *Given a basic block $B = [b_0, b_1, ..., b_n]$, we define its block-partitioning as follows:*

$$blocks(B) = \left\{ B_i \equiv b_i, \ldots, b_j \;\middle|\; \begin{array}{l} (\forall k.i < k < j, b_k \notin Jump \cup Terminal \cup Split \cup \\ \{\mathit{JUMPDEST}\}) \wedge (\ i{=}0 \vee b_{i-1} \in Split \cup \{\mathit{JUMPDEST}\}\ ) \wedge \\ (\ j{=}n \vee b_{j+1} \in Jump \cup Split \cup Terminal\ ) \end{array} \right\}$$

where

$$\begin{aligned} Jump &= \{\mathit{JUMP}, \mathit{JUMPI}\} \\ Terminal &= \{\mathit{RETURN}, \mathit{REVERT}, \mathit{STOP}, \mathit{INVALID}\} \\ Split &= \{\mathit{SSTORE}, \mathit{MSTORE}, \mathit{LOGX}, \mathit{CALLDATACOPY}, \mathit{CODECOPY}, \mathit{EXTCODECOPY}, \\ &\quad \mathit{RETURNDATACOPY}\} \end{aligned}$$

Figure 2.1 shows a basic block from the CFG of the contract SimpleChildToken[1] at the top. It contains a `LOG3` bytecode at program line 20 (L20 for short) and a `SSTORE` instruction at L41. In this case, there will be two sub-blocks to be optimized, the first sub-block contains the EVM instructions from L1 to L19 and the second one from L21 to L40. Note that, though the effects of both bytecodes (`LOG3` and `SSTORE`) are not reflected on the stack, the `SSTORE` instruction does not produce any sub-block as it is the last bytecode of the sequence. Hence, the bytecode that induces the two sub-blocks that will be used to build the functional description of the stack is `LOG3`.

Once we have the partitioned blocks from the CFG, we aim at obtaining a functional description of the output stack (i.e., the stack after executing the sequence of bytecodes in the block). The SFS describes the output stack of the block to be optimized in terms of the input stack and is defined as follows.

[1] `https://etherscan.io/address/0x0d152b9ee87ebae179f64c067a966dd716c50742`

Definition 2.3 (SFS). *Let B be a block and $\mathcal{S}_0$ its initial stack of size n that contains at each position $i \in \{0, \ldots, n-1\}$ a symbolic variable s_i that represents the element stored at position i. The* stack functional specification *of B is the output stack $\mathcal{S}$ of size m that contains at each position $j \in \{0, \ldots, m-1\}$ the element located at position j in the stack after executing the EVM instructions of B. Each element can be either (i) a non-negative integer value, (ii) a variable $s_i \in \mathcal{S}_0$, or (iii) a symbolic expression composed by a functor* `OP` *with k parameters $a_1, \ldots, a_k$ such that each a_i can be either of type (i), (ii) or (iii).*

The latter corresponds to an EVM instruction `OP` *that operates on the stack (other than* `SWAP`*k,* `PUSH`*k,* `DUP`*k, and* `POP`*) using k stack elements $s_i, \ldots, s_{i+k}$.*

Intuitively, the purpose of the SFS is to specify the effect of executing all EVM bytecodes of block B from an initial stack $\mathcal{S}_0$ that simply contains symbolic variables s_i to represent the stack elements (considering s_0 as the top-most element of the stack). Note that the stack is empty before executing a transaction and the number of elements on which each EVM instruction operates and that it produces is fixed and specified in [43]. Thus, the number of elements in the stack at the beginning of each block can be inferred statically. W.l.o.g, we assume that the initial stack size of each block is given within the CFG. The above definition specifies that the output stack may have (item i) non-negative integer values introduced by `PUSH`k, (item ii) symbolic variables restricted to those contained in the input stack $\mathcal{S}_0$, that will be generated by instructions `SWAP`k and `DUP`k, and (item iii) symbolic expressions to represent all remaining EVM instructions (arithmetic, bit-wise operations and hash operations among others). The symbolic expressions generated contain as functors the same operation names as the original EVM instructions and as arguments the symbolic variables, integer values or (recursively) symbolic expressions stored in the stack positions that they consume. Given a block of bytecode instructions, the SFS can be computed by symbolically executing [29, 6] the bytecode instructions from the initial stack $\mathcal{S}_0$. The next example shows how this process works.

Example 2.1. *Consider the first sub-block induced by the partitioning at the underlined instructions in Figure 2.1, which is formed by the EVM bytecodes that go from the bytecode at L1 to the one at L19. Before computing the functional description of the stack, the initial stack is $\mathcal{S}_0 = [s_0, s_1, s_2]$ and $n = 3$. Symbolic execution from $\mathcal{S}_0$ leads to the following output stacks at these selected program points:*

$L1:$ $[3, s_0, s_1, s_2]$
$L4:$ $[\mathtt{MLOAD}(64), \mathtt{SLOAD}(3), s_0, s_1, s_2]$
$L6:$ $[160, 1, \mathtt{MLOAD}(64), \mathtt{SLOAD}(3), s_0, s_1, s_2]$
$L9:$ $[\mathtt{SUB}(\mathtt{EXP}(2, 160), 1), \mathtt{MLOAD}(64), \mathtt{SLOAD}(3), s_0, s_1, s_2]$
$L11:$ $[s_0, \mathtt{SUB}(\mathtt{EXP}(2, 160), 1), \mathtt{SUB}(\mathtt{EXP}(2, 160), 1), \mathtt{MLOAD}(64), \mathtt{SLOAD}(3), s_0, s_1, s_2]$
$L14:$ $[\mathtt{AND}(\mathtt{SLOAD}(3), \mathtt{SUB}(\mathtt{EXP}(2, 160), 1)), \mathtt{MLOAD}(64), \mathtt{AND}(s_0, \mathtt{SUB}(\mathtt{EXP}(2, 160), 1)), s_0, s_1, s_2]$
$L19:$ $[\mathtt{MLOAD}(64), 0, 139, \mathtt{AND}(\mathtt{SLOAD}(3), \mathtt{SUB}(\mathtt{EXP}(2, 160), 1)), \mathtt{AND}(s_0, \mathtt{SUB}(\mathtt{EXP}(2, 160), 1)), s_0, s_1, s_2]$

Note that each output stack represents the SFS after the execution of the sequence of EVM instructions up to the selected program points. Thus, the output stack of the SFS of the analyzed block is

$\mathcal{S} = [\mathtt{MLOAD}(64), 0, 139, \mathtt{AND}(\mathtt{SLOAD}(3), \mathtt{SUB}(\mathtt{EXP}(2, 160), 1)), \mathtt{AND}(s_0, \mathtt{SUB}(\mathtt{EXP}(2, 160), 1)), s_0, s_1, s_2]$

We can see that the stack is updated by inserting 3 at the top at L1 after considering the instruction `PUSH1 0x03`*. At L4 and L9, after the execution of bytecodes* `MLOAD` *and*

(1)	$OP(X_{int}, Y_{int}) = eval(OP, X_{int}, Y_{int})$	(9)	$ISZERO(ISZERO(GT(X,Y))) = GT(X,Y)$
(2)	$OP(X_{int}) = eval(OP, X_{int})$	(10)	$ISZERO(XOR(X,Y)) = EQ(X,Y)$
(3)	$MUL(X,1) = X$	(11)	$AND(AND(X,Y),Y) = AND(X,Y)$
(4)	$DIV(X,0) = 0$	(12)	$OR(X, AND(X,Y)) = X$
(5)	$AND(X,X) = X$	(13)	$AND(ORIGIN, 2^{160}-1) = ORIGIN$
(6)	$GT(1,X) = ISZERO(X)$	(14)	$MUL(SHL(X,1),Y) = SHL(X,Y)$
(7)	$EQ(X,0) = ISZERO(X)$	(15)	$BALANCE(ADDRESS) = SELFBALANCE$
(8)	$ISZERO(LT(0,X)) = ISZERO(X)$		$\cdots$

Figure 2.2: Subset of simplification rules based on the semantics of EVM bytecodes, where X_{int} and Y_{int} are constants.

`SUB` respectively, new symbolic expressions are introduced at the top of the stack with the same name as the EVM op code that they represent. In addition, they have as arguments the corresponding elements of the stack on which the bytecodes operate. Note that, in the case of `MLOAD`, it has an integer value (64) as its argument. However, since the `SUB` bytecode consumes the result obtained from the exponentiation, the symbolic expression corresponding to `SUB` has another expression as its first argument (`EXP`). At L11, the symbolic variable of the initial stack s_0 is duplicated. Finally to obtain the SFS of the analyzed block, the element at the top of the stack (0) is swapped at L19 with the element stored at position 1 (`MLOAD`(64)).

2.3 Optimizations based on the Semantics of the Bytecodes

Before producing the SMT encoding, we apply a first optimization step on the SFS based on the semantics of certain EVM instructions that are not captured by our encoding. These optimizations lead to a simpler SFS representation that will potentially reduce both the bound on the number of instructions and the gas consumed by the block being optimized. The simplification process is applied recursively as the application of a rule may induce new simplifications.

The rules allow us to replace the symbolic expressions of the SFS that match the left-hand side of the rules with the right-hand side. Figure 2.2 shows a selection of simplification rules (the complete list is included in Appendix A). Rules (1) and (2) evaluate the corresponding arithmetic, Boolean or bit-wise operation OP if its arguments are constants; rule (3) operates the identity element and (4) evaluates the `DIV` instruction according to the semantics defined in EVM; rule (5) applies the idempotence of the bytecode `AND`; rules (6) - (10) transform the sequence of bytecodes using the semantics of the involved EVM bytecodes; rules (11) and (12) combine associativity and idempotence of `AND` and `OR`; rule (13) uses the fact that addresses are represented with 20 bytes in EVM and then, applying the mask to obtain the last 20 bytes of a 32-bytes word that contains an address, returns the original one; rule (14) applies an equivalence of the left shifting operation; and rule (15) replaces the sequence that computes the balance bound to the current contract by the analogous new opcode `SELFBALANCE` included in Solidity version higher than 0.5.13.

Example 2.2. *Consider the SFS of the block in Example 2.1. We can apply rule (1) in Figure 2.2 twice. In a first step, it evaluates the exponentiation* $\texttt{EXP}(2, 160)$ *and substitutes the symbolic expression with the result of the operation* $1.461501637 \cdot 10^{48}$*. Hence, we obtain a new SFS* $\mathcal{S} = [\texttt{MLOAD}(64), 0, 139, \texttt{AND}(\texttt{SLOAD}(3), \texttt{SUB}(1.461501637 \cdot 10^{48}, 1)), \texttt{AND}(s_0, \texttt{SUB}(1.461501637 \cdot 10^{48}, 1)), s_0, s_1, s_2]$*. Then, if we evaluate the* `SUB` *expressions of the SFS, we obtain the final SFS of the analyzed block*

$$\mathcal{S} = [\texttt{MLOAD}(64), 0, 139, \texttt{AND}(\texttt{SLOAD}(3), 1.461501637 \cdot 10^{48} - 1), \texttt{AND}(s_0, 1.461501637 \cdot 10^{48} - 1), s_0, s_1, s_2]$$

. Note that without the simplification rules we would not be able to optimize the symbolic expressions of the SFS as they have to appear once in the optimized block to generate the same result. Intuitively, thanks to the rules, we will be able to save the gas consumed by `SUB`*,* `EXP` *and two of the* `PUSH` *bytecodes that locate on the stack the integer values on which they operate and replace the value of the third* `PUSH` *bytecode by the result of the evaluation.*

As the ebso optimizer [33] encodes the semantics of the arithmetic and bit-vector operations, we have used it together with Z3 [15] to verify the correctness of the rules proposed when possible. To this end, we have encoded the search for input to distinguish the left- from the right-hand side of the rules. If no such input can be found, the rule is correct. However, ebso is not able to verify all of them due to its limitations, e.g., it does not handle complex arithmetic operations such as `EXP`, it does not support the EVM instructions included from Solidity versions up to 0.5.0 such as `SHL` or `SELFBALANCE`, and it is not able to simulate the bytecodes that involve addresses such as `ORIGIN` or `ADDRESS`. Out of the rules in Figure 2.2, it is not able to verify (13), (14) and (15).

Chapter 3

Optimal Synthesis using Max-SMT

This section describes our Max-SMT encoding. First, we include a brief discussion in some aspects related to the SFS that must be considered in order to generate the encoding. After that, a refinement of the *Stack Functional Specification* is provided: the *Abstract Stack Functional Specification*. We give an overview of how the stack is modelled in the Max-SMT problem in Section 3.3. Then, we present a brief summary of the complete encoding in Section 3.4, as well as a more detailed explanation of the alternative constraints we proposed to determine the best setting to be used in a real inclusion of the optimizer within a compiler. Finally, in Section 3.4.2 we describe two variants of the optimization problem in terms of soft constraints. The complete formal description of the encoding is provided in Appendix B.

3.1 Bounding the number of instructions and the stack: b_o and b_s

In this section, we will discuss some information that can be inferred from a SFS and describe an alternative representation to deal with when building a encoding.

Number of Instructions: Fixing the bound on the number of instructions of the synthesized code and the needed size of the stack is essential for the result of the synthesis process. Taking a too large number can notably affect the efficiency of the applied optimizers, but taking a too low number can turn the problem into unsatisfiable. In our approach, we take as bound on the number of operations the size of the original program but only counting the *necessary* `POP` operations.

A stack variable s_i in the initial stack S_I is *useless* (i.e., it needs to eventually be popped) if it does not occur in the final stack as element or in an expression. Then, then number of necessary `POP`s coincides with the number of initial stack variables that are useless. Let m be the number of operations of the original code, p the number of `POP` operations and n the necessary ones. Then we take as bound on the number of instructions $\boxed{b_o = m - p + n}$.

Size of stack: Similarly, the size of the stack is bound using the original program. In this case, we define the bound on the size of the stack as the smallest number of stack elements needed to execute the original program. To compute this bound, we infer the *highest stack position* h used by the execution of the program and the number of stack

elements which are *not used*. As we know the number of elements that the stack contains before executing the program and the number of elements that each bytecode consumes and generates, h is computed during the symbolic execution in Sec. 2. A stack variable s_i in the initial stack S_I is *not used* if it appears in the same position in the target stack S_T and it is not used by any of the bytecodes in the block. Then, the *highest non-used stack index* u is defined as the highest index of a not used stack element such that all the stack elements whose indexes are smaller than it are not used neither. Then, the bound on the size of the stack is $\boxed{b_s = h - u}$.

3.2 Abstract Stack Functional Specification

Instead of working directly with the notion of *Stack Functional Specification* presented in Section 2.2, we will introduce a refinement of this notion that will be crucial for defining the Max-SMT problem. The motivation of this definition is based on avoiding composite elements when representing the stack evolution in the Max-SMT problem.

Hence, for each *non-basic* opcode in a *basic-block* and each application of that opcode to certain operands, a new stack variable is introduced. These stack variables are denoted as *fresh stack variables*, so that a distinction can be made between them and the *initial stack variables*.

Every *fresh stack variable* represents the application of an opcode that consumes certain parameters. We need to keep track of the parameters associated to each *fresh stack variable*, so an operator map is introduced to link both. This map may contain recursive definitions when different composite elements are chained, but no infinite recursive definitions can be introduced due to its construction. All elements become shallow as a result of this process. Next example illustrates this behaviour:

Example 3.1. *Let us consider the*

$$SFS := [s_0, \texttt{MUL}(s_2, \texttt{ADD}(s_1, s_0))]$$

obtained from initial stack $\mathcal{S}_0 = [s_0, s_1, s_2]$. *It is clear that the initial stack variables are* s_0, s_1, s_2.

We need to introduce two fresh stack variables: s_4, s_5. *Then, we can define the operator map* $M := \{s_4 \mapsto \texttt{ADD}(s_1, s_0), s_5 \mapsto \texttt{MUL}(s_2, s_4)\}$ *and the final stack can be expressed as* $\mathcal{S} = [s_0, s_5]$.

However, this map might contain redundant pairs: by just generating a new fresh variable for each expression, the same operation could be assigned to different stack variables. In order to avoid this situation, a *minimal* map will be considered as part of our representation. Note that there exists commutative EVM operations, so this must be taken into account when identifying redundant pairs.

This representation will be denoted as *Abstract SFS* or *ASFS*. It is composed of three different components:

1. An initial stack: a list of stack variables $s_0, \ldots, s_n$ with no repeated elements.

2. A final stack: a list that contains a series of either stack variables or numerical values from 0 to $2^{256} - 1$.

3. An *uninterpreted* operation map: a *minimal* map that links every fresh new variable to its corresponding parameters.

The following example shows how the abstract SFS is built from a SFS:

Example 3.2. *Let us consider the*

$$SFS := [s_0, \texttt{MUL}(\texttt{SUB}(s_2, s_1), \texttt{ADD}(s_1, s_0)), 50, \texttt{ADD}(s_0, s_1), \texttt{SUB}(s_1, s_2)]$$

obtained from initial stack $\mathcal{S}_0 = [s_0, s_1, s_2, s_3]$.

Note that `ADD` *is a commutative operation, whereas* `SUB` *is not. Therefore,* $\texttt{ADD}(s_1, s_0) = \texttt{ADD}(s_0, s_1)$ *but* $\texttt{SUB}(s_2, s_1) \neq \texttt{SUB}(s_1, s_2)$.

The corresponding abstract SFS needs to introduce four fresh stack variables: s_4, s_5, s_6 *and* s_7. *It contains the initial stack* $\mathcal{S}_0 = [s_0, s_1, s_2, s_3]$, *the final stack* $\mathcal{S} = [s_0, s_7, 50, s_4, s_6]$ *and the map* $M := \{s_4 \mapsto \texttt{ADD}(s_1, s_0), s_5 \mapsto \texttt{SUB}(s_2, s_1), s_6 \mapsto \texttt{SUB}(s_1, s_2), s_7 \mapsto \texttt{MUL}(s_4, s_5)\}$.

The operation map satisfies a very interesting property: by removing duplicates and considering commutative operations, it holds that every fresh stack variable must appear at some point in the encoding. This fact will be crucial to understand why C_L encoding works in Section C.1.

3.2.1 Properties of the Abstract Stack Functional Specification

In previous section, the notion of the *Abstract SFS* was introduced, as well as some of its properties. In this section, these properties are formalized so we can reason about them in the following chapters.

An *ASFS* must satisfy the following property: the *fresh stack variables* introduced in the map cannot belong to $\mathcal{S}_0$:

Property 3.1. *Let* $ASFS := \langle \mathcal{S}_0, \mathcal{S}, M \rangle$ *be an* ASFS *associated to a* sub-block. *Then,* $dom(M) \cap \mathcal{S}_0 = \varnothing$.

This property is imposed to guarantee there is no confusion among *initial stack variables* and *fresh stack variables*.

Besides, two other interesting properties follow due the construction of an *ASFS*: no infinite recursive definitions are allowed and it is *minimal* w.r.t the map operator.

Firstly, we introduce the notion of *convergence* of the *ASFS*:

Property 3.2. *Let* $ASFS := \langle \mathcal{S}_0, \mathcal{S}, M \rangle$ *be an* ASFS *associated to a* sub-block. *Then,* M *is finite and we can define a strict weak order* $<$ *among elements in* $dom(M)$ *s.t.* $s_i < s_j$ *iff the composite element* s_j *represents in the* SFS *contains the element* s_i *represents.*

Proof. M is clearly finite, as *sub-blocks* can only contain a finite number of operations. We will prove $<$ satisfies all strict weak order properties:

1. *Irreflexivity*: elements in the SFS have finite depth. Therefore, a stack variable that represents a composite element cannot contain itself as a parameter, as this would lead to an element with infinite depth. Thus, it is followed that $s_j \not< s_j$.

2. *Transitivity*: let us take three fresh stack variables s.t. $s_i < s_j < s_k$ and prove that $s_i < s_k$. This fact directly holds, as clearly s_k contains the element s_i represents due to s_j also containing s_i.

3. *Antisymmetry*: Directly followed from *Irreflexivity* and *Transitivity*.

□

This way, the parameters related to every fresh stack variable s_i can be either numerical values, initial stack variables or other fresh stack variables s_j s.t. $s_j < s_i$. Thus, no infinite recursive definitions are allowed.
Minimality is followed from two independent facts: there are no repeated operations in the map and every operation is eventually applied to obtain an element of the final stack *i.e.* no *noise* is introduced.

First property can be stated as follows:

Property 3.3. *Let* $ASFS := \langle \mathcal{S}_0, \mathcal{S}, M \rangle$ *be an* ASFS *associated to a* sub-block. *Then, the ASFS contains no repeated stack variables if every fresh stack variable* s_i *satisfies one of the following properties:*

1. $M(s_i) := \texttt{Op}(d_1, \ldots, d_n)$ *and there is no other fresh stack variable* s_j *in* M *s.t.* $M(s_j) := \texttt{Op}(d_1, \ldots, d_n)$ *, being* `Op` *a non-commutative operation.*

2. $M(s_i) := \texttt{COp}(d_1, d_2)$ *and there is no other fresh stack variable* s_j *in* M *s.t.* $M(s_j) = \texttt{COp}(d_1, d_2)$ *nor* $M(s_j) = \texttt{COp}(d_2, d_1)$*, being* `COp` *a commutative operation.*

whereas second property is stated through the following statement:

Property 3.4. *Let* $ASFS := \langle \mathcal{S}_0, \mathcal{S}, M \rangle$ *be an* ASFS *associated to a* sub-block. *Then, the ASFS introduces no noise if every stack variable* s_i *in the domain of* M *satisfies any of the following properties:*

1. $s_i \in \mathcal{S}$

2. *There exists a functor* `Op` *and another stack variable* s_j *s.t.* $s_j \neq s_i$, $s_j \in dom(M)$ *and* $M(s_j) = \texttt{Op}(v_0, \ldots, s_i, \ldots, v_k)$.

First property can be directly ensured from the construction of the *ASFS*, whereas the second holds trivially due to *SFS* always containing *meaningful* values.

3.3 Modeling the stack

An element of utmost importance in our encoding is the representation of the stack and the elements it contains. In the model, the stack only has non-negative integer constants in the domain $\{0, \ldots, 2^{256} - 1\}$ (they are handled as 256-bit words independently if they represent a negative number or not [43]), initial stack variables $s_0, \ldots, s_{k-1}$ and fresh variables $s_k, \ldots, s_v$. These fresh variables represent the output of every non-stack related opcode that appears in the SFS, i.e., they are abstracting the symbolic expressions that appear in the SFS. We have a minimal mapping that binds a new fresh variable to each symbolic expression, i.e., we use the same variable for the expressions that occur more than once.

In order to distinguish between constants and the variables s_i, we assign to every variable s_i, with $i \in \{0, \ldots, v\}$, the constant $2^{256} + i$. Thus, the following constraint is applied:

$$S_V = \bigwedge_{0 \leqslant i < v} s_i = 2^{256} + i$$

Let us now show how we model the stack that is produced along the execution of the EVM instructions. First, we have to fix a bound on the number of operations b_o and the size of the stack b_s. Although we can apply different heuristics, we get sound bounds by considering the initial number of operations and the maximum number of stack elements involved in the block (as mentioned in Section 2 this can be statically computed). Now, we have to express a stack of at most size b_s after executing j operations with $j \in \{0, \ldots, b_o\}$. To do that, we use existentially quantified variables $x_{i,j} \in \mathbb{Z}$ with $i \in \{0, \ldots, b_s - 1\}$ and $j \in \{0, \ldots, b_o\}$ to express the word at position i of the stack after executing the first j operations of the code, where $x_{0,j}$ denotes the element on the top of the stack.

In addition, to complete the modeling of the stack we generate propositional variables $u_{i,j}$ with $i \in \{0, \ldots, b_s - 1\}$ and $j \in \{0, \ldots, b_o\}$, to denote the *utilization* of the stack, i.e., the elements of the stack that are active. Variable $u_{i,j}$ indicates that the word at position i of the stack after executing the first j operations exists (if it is set to true) or not (if it is set to false).

3.4 Encoding of instructions

Let $\mathcal{I}$ be the set of instructions occurring in our problem. The set $\mathcal{I}$ is split in three subsets $\mathcal{I}_S \uplus \mathcal{I}_U \uplus \mathcal{I}_C$, where:

- $\mathcal{I}_S$ contains the stack operations: `PUSH`, that introduces an up to 32-bytes element on top of the stack; `POP` that removes the top of the stack; `DUP`k, with $k \in \{1, \ldots, 16\}$ that copies the $k-1$ element of the stack on top of the stack; and `SWAP`k, with $k \in \{1, \ldots, 16\}$ that swaps the top of the stack with the k element of the stack,
- $\mathcal{I}_U$ contains the non-commutative uninterpreted functions occurring in the SFS,
- $\mathcal{I}_C$ contains the commutative uninterpreted functions occurring in SFS.

Let m_ι be the carnality of $\mathcal{I}$ and θ a mapping from the set of instructions in $\mathcal{I}$ to consecutive different non-negative integers in $\{0, \ldots, m_\iota - 1\}$. In order to encode the selected instructions at every step j, we introduce the existentially quantified variables $t_j \in \{0, \ldots, m_\iota\}$, with $j \in \{0, \ldots, b_o - 1\}$ where for every instruction $\iota \in \mathcal{I}$, if $t_j = \theta(\iota)$ then we have that the operation executed at step j is ι. In addition, we use associated existentially quantified variables $a_j \in \{0, \ldots, 2^{256} - 1\}$, with $j \in \{0, \ldots, b_o - 1\}$, to represent the value pushed at the top of the stack when $t_j = \theta(\texttt{PUSH})$.

Once the variables are defined, we introduce the constraints that model the impact that each operation has in the stack. We have to model the effect of choosing one of the operations at each step. Hence, we add a clause for each $\iota \in \mathcal{I}$ at each step $j \in \{0, \ldots, b_o - 1\}$ of the form

$$t_j = \theta(\iota) \Rightarrow C_\iota(j)$$

where $C_\iota(j)$ expresses the new state of the stack after applying the instruction ι at step j.

For instance, to encode $C_{\texttt{PUSH}}(j)$ we generate the following constraint:

$$t_j = \theta(\texttt{PUSH}) \Rightarrow 0 \leq a_j < 2^{256} \wedge \neg u_{b_s-1,j} \wedge u_{0,j+1} \wedge x_{0,j+1} = a_j \wedge \mathit{Move}(j, 0, b_s - 2, 1)$$

The predicate *Move* states that every variable $x_{i,j+1}$ is the same as $x_{i-1,j}$, i.e., the values stored in the stack after the `PUSH` instruction are kept in the stack and that the elements

are moved one position (as the variable on the top of the stack $x_{0,j+1}$ is equal to a_k, the value it has just been pushed). In addition, we ensure that the stack does not overflow by means of the utilization variables: $\neg u_{b_s-1,j}$ ensures that the stack is not complete, it is able to locate the new push element and hence, it does not overflow. Similarly, $u_{0,j+1}$ states that the top of the stack after executing the `PUSH` instruction is active, as it now contains the pushed value.

When $\iota \in \mathcal{I}_U \uplus \mathcal{I}_C$, we have to generate the corresponding clause according to the information that is contained in the SFS: the top elements on the stack must match the ones needed for executing the corresponding instruction and the result is stored on top of the stack at step $j+1$. Additionally, commutative instructions can consume stack elements in two different orders, so we have to consider the two combinations when defining $C_\iota(j)$ if $\iota \in \mathcal{I}_C$. In what follows, let $C_\mathcal{I}$ be the set of all clauses generated to encode the instructions.

Finally, we also need to represent the state of the stack at the beginning of the block and after performing all operations inside it. Hence, we introduce a constraint B to describe how the input stack $[s_0, \ldots, s_{k-1}]$ is when entering the block and a constraint E to describe how the final stack $[f_0, \ldots, f_{w-1}]$ is at the end of its execution.

$$\begin{aligned} B &= \textstyle\bigwedge_{0 \leqslant \alpha < k} (u_{\alpha,0} \ \wedge \ x_{\alpha,0} = s_\alpha) \ \wedge \ \bigwedge_{k \leqslant \beta \leqslant b_s - 1} \neg u_{\beta,0} \\ E &= \textstyle\bigwedge_{0 \leqslant \alpha < w} (u_{\alpha,b_o} \ \wedge \ x_{\alpha,b_o} = f_\alpha) \ \wedge \ \bigwedge_{w \leqslant \beta \leqslant b_s - 1} \neg u_{\beta,b_o} \end{aligned}$$

B initializes at step $j = 0$ the utilization variables $u_{\alpha,0}$ to express that the words from 0 to α will be active and assigns to the variables $x_{\alpha,0}$ the corresponding stack value s_α. In addition, it states that the rest of the words are not active. Similarly, constraint E defines the state of the stack in the last step $j = b_o$. Then, by combining all the constraints defined above, we express the whole encoding C_{SFS}. Thus, $C_{SFS} = S_V \ \wedge \ C_\mathcal{I} \ \wedge \ B \ \wedge \ E$.

3.4.1 Additional constraints

Our previous encoding is sufficient to obtain models that compute the given SFS. However, other constraints can be added to our encoding in order to provide the solver more information that may help to find models faster. A full explanation on why these constraints are indeed useful is provided in Appendix C.

The proposed constraints in this section can be classified in three different groups according to its impact. The first group corresponds to constraints that are redundant, i.e., every solution of our problem must satisfy them. Therefore, the resulting encoding considering them is equivalent to the initial one. We propose and analyze two new constraints:

C_L Every uninterpreted operation must be used at least once. This property is equivalent to stating that every operation in $\mathcal{I}_U \uplus \mathcal{I}_C$ must be assigned for at least one t_j. It is part of the default encoding:

$$\textstyle\bigwedge_{\iota \in \mathcal{I}_U \uplus \mathcal{I}_C} \ \bigvee_{0 \leqslant j < b_o} \ t_j = \theta(\iota) \tag{3.1}$$

This encoding will be denoted as *Uninterpreted opcodes at least once* or C_L in what follows.

C_N Every numeric value that appears as an input of an uninterpreted function or in the final stack must be *pushed at least once*. This constraint is related to the previous

one: if every uninterpreted operation must be used, then it means all their input values must appear at some point on the stack. Hence, PUSH instructions must be executed to introduce them. We use P to denote the set of numeric input values for a given block, so that the following constraint is considered:

$$\bigwedge_{k \in P} \bigvee_{0 \leqslant j < b_o} (t_j = \theta(\texttt{PUSH}) \wedge a_j = k) \tag{3.2}$$

This encoding will be denoted as *Numerical values pushed at least once* or C_N in what follows.

The second group corresponds to constraints that may not be satisfied by every possible solution, but at least one optimal solution must satisfy them. These constraints greatly affect the exploration, as now we will be able to find fewer solutions that satisfy the constraints. Hence, the resulting encoding problem is not equivalent as the initial one, but every optimal model of it is also an optimal model of the initial problem.

Only C_R encoding belongs to this group:

C_R The instruction just before POP cannot introduce an element in the stack. In our current model, this means that the instruction before a POP can only be another POP or SWAPk. We denote the set of instructions that do not introduce an element in the stack by Q. The following set of constraints is proposed:

$$\bigwedge_{0 \leqslant j < b_o - 1} (t_{j+1} = \theta(\texttt{POP}) \Rightarrow \bigvee_{\iota \in Q} (t_j = \theta(\iota))) \tag{3.3}$$

The rationale for this constraints set is that producing an element with an opcode and then popping it is meaningless, as we could just have directly applied POP to all operands consumed by that opcode and obtain the same stack. Formally, this assertion can be proven by checking that the gas cost associated with the generation of the element and the subsequent POP operation is lower than or equal to applying POP a number of times equal to the number of items consumed by the opcode that generates the element.

This encoding will be denoted as *Restricted opcodes before POP* or C_R in what follows.

The last group corresponds to those constraints that may lead to non-optimal solutions, but in general are satisfied by optimal solutions. This way, they serve as an heuristic to guide the search and try to obtain better results.

Only C_U encoding is included in this group. This encoding has been tested using dataset (i) from Section 6.1, and we have not detected any loss of optimality in any sub-block. However, by building an artificial set-up, we have shown that optimality could be lost in extreme cases by providing a counterexample in Section C.4.

C_U Every uninterpreted operation with gas cost greater or equal than 3 must be used *at most once*. Instead of applying these operations more than once, we could have applied the corresponding DUPk to obtain the stack variable.

$$\bigwedge_{\iota \in \mathcal{I}_U \uplus \mathcal{I}_C} \bigwedge_{0 \leqslant j < b_o} (t_j = \theta(\iota) \Rightarrow \bigwedge_{0 \leqslant i < b_o, i \neq j} (t_i \neq \theta(\iota))) \tag{3.4}$$

The rationale for this constraints set is that in many cases, there exists a reordering of instructions that preserves optimality and ensures the element produced by that operation is still in the stack, so we can just apply a `DUPk` operation to obtain it.

The reason why the cost must be greater than or equal to 3 is that otherwise it is better to apply this function again instead of using a `DUPk` instruction. For instance, `CALLDATAVALUE` opcode gas cost is 2 and consumes no input, so that the optimal solution of a block that only contains this opcode twice is `CALLDATAVALUE CALLDATAVALUE`, whose associated cost is 4. Instead, if the constraint would be applied, an optimal solution would be `CALLDATAVALUE DUP1`, whose associated cost is 5.

This encoding will be denoted as *Uninterpreted opcodes at most once* or C_U in what follows.

3.4.2 Optimization using Max-SMT

Now that we know that every model that satisfies the constraints in C_{SFS} described in Section 3.4 provides a block that computes the SFS, we want to obtain the optimal solution, i.e., a sequence of EVM instructions that produces the same stack but that consumes the least amount of gas possible. Since the cost of the solution can be expressed in terms of the cost of every of the instructions we select in all t_j, we will introduce *soft constraints* expressing the cost of every selection. A (partially weighted) Max-SMT problem is an optimization problem where we have two different components: (i) an SMT formula which establishes the *hard constraints* of the problem, and (ii) a set of pairs $\{[C_1, \omega_1], \ldots, [C_m, \omega_m]\}$, where each C_i is an SMT clause and ω_i is its weight, that establishes the *soft constraints*. The hard constraints have to be satisfied while sum of the weights of the soft constraints that are not satisfied has to be minimized.

Our Max-SMT optimization problem is encoded as follows: we include the SMT formula C_{SFS} previously described as *hard constraints*, and a set of *soft constraints* such that sum of the weights of the falsified soft constraints coincides with the cost (in terms of gas) of the operations taken in every step. Hence, the model found, which is the optimal solution, is also optimal in terms of gas cost.

In the EVM, every operation has an associated gas cost, which in general is constant, but in some few cases may depend on the particular arguments it is applied to or on the state of the blockchain. All these operations that are non-constant are considered as uninterpreted, and hence we cannot change the operands on which they are applied. Therefore, omitting the non-constant part cannot affect which is the optimal solution.

We propose two different *soft constraints* encodings in order to study which one works better. The first one considers all instructions $\iota \in \mathcal{I}$, its associated gas cost v_i and all existentially quantified variables t_j that appear in C_{SFS}. For every possible combination of them, we consider the Max-SMT problem

$$O_{SFS} = C_{SFS} \ \wedge \bigwedge_{0 \leqslant j < b_o} \bigwedge_{\iota \in \mathcal{I}} [t_j \neq \theta(\iota) \ , \ v_j \ ] \tag{3.5}$$

This means that for every model that satisfies C_{SFS}, the only non-satisfied soft constraints are those where $t_j = \theta(\iota)$, meaning that the total sum of the falsified soft constraints matches its cost.

The second set of *soft constraints* can be seen as a refined dual model of the previous one, where instead of using inequalities in the soft constraints, equalities are applied. In this case, the reasoning becomes more tricky, as now the non-satisfied clauses related to an assignment are those in which $t_j = \theta(\iota)$ does not appear.

Thus, we need to split our set of instructions I in $p+1$ disjoint sets $W_0 \uplus \ldots \uplus W_p$ where all instructions in W_i have the same constant cost cost_i, and such that the costs are strictly increasing, i.e., $cost_0 = 0$ and $\mathsf{cost}_{i-1} < \mathsf{cost}_i$ for all $i \in \{1 \ldots p\}$. The following Max-SMT problem O'_{SFS} is obtained

$$O'_{SFS} = C_{SFS} \ \wedge \bigwedge_{0 \leqslant j < b_o} \ \bigwedge_{1 \leqslant i \leqslant p} \ [\bigvee_{\iota \in W_0 \uplus \ldots \uplus W_{i-1}} t_j = \theta(\iota) \ , \ w_i \ ] \tag{3.6}$$

Therefore, if the selected instruction at step j is ι (i.e. $t_j = \theta(\iota)$) for some $\iota \in W_i$ then we accumulate the weight w_α of all soft clauses with $\alpha \in \{1 \ldots i\}$, which as said sums cost_i, and hence we accumulate the cost of executing the instruction ι.

Chapter 4

Verification

4.1 Lightweight verification of optimization

Our tool gasol includes a *lightweight verifier* as its last component (see Figure 1.1). It aims to (lightweight) verify the correctness of the optimization by checking that the optimized block generated by the tool complies with the specification (SFS) of the original block. Therefore, the part of extraction of the CFG and synthesis of the SFS from the blocks belongs to the *trusted base code* and only the posterior phase (i.e., the generation of the encoding and the solving process to find the optimal solution) are verified. The verification process consists in proving the *equivalence* between the SFS of the original block and the SFS of the optimized one, i.e., that the blocks produce the same representation of the stack. The verifier hence mainly allows us to find bugs in the encoding and even some of them that may exist in the SMT solvers. However, it also allowed us to fix bugs in the generation of the SFS as this process is applied again on the optimized code.

First of all, in order to apply the verifier, we have to "decode" the model returned by the SMT solver. Given an SMT encoding, the solver finds a *model* that satisfies the hard constraints and that minimizes the sum of the weights bound to the soft constraints (see Section 3.4.2) of the encoding. The *model* is a sequence of assignments that, among others, binds:

- a non-negative integer value $x \in \{0, \ldots, m_\iota\}$ to each of the variables t_j, the existentially quantified variables that represent the instruction executed at step j, and
- a value $y \in \{0, \ldots, 2^{256} - 1\}$ to the variables a_j, that represent the values pushed on top of the stack at step j.

Hence, we obtain the optimized block by decoding the model using the mapping θ defined in Section 3.4 and obtaining the corresponding numerical value in case that t_j is `PUSH`. Once that the optimized block is decoded, we generate its SFS in order to check that it complies the specification of the original block, i.e., that the SFS of the optimized block is *equivalent* to the one of the original block from which it has been synthesized.

Definition 4.1 (Equivalent SFS)**.** *Let $\mathcal{S}$ be a SFS of size n obtained from the initial stack $\mathcal{S}_0$ of size j, and let $\mathcal{S}'$ a second SFS of size m obtained from the initial stack $\mathcal{S}'_0$ of size k, we say that $\mathcal{S}$ and $\mathcal{S}'$ are* equivalent *(denoted as $\mathcal{S} \equiv \mathcal{S}'$) iff:*

(i) $j = k$, and

(ii) $n = m$ *and* $\forall i \in \{0, \ldots, n-1\}$ $s_i = s'_i$*, modulo commutativity for the symbolic expressions.*

For two SFSs to be equivalent, two conditions have to be hold: (i) as the SFSs are expressed in terms on the initial stacks, both of them need to have the same length; and (ii) all the elements of the two SFS have to be identical if they are integers values or symbolic variables. For the case of the symbolic expressions they have to contain the same functor (EVM instruction) `OP` and the arguments need to be (recursively) equal in both expressions considering the two possible orders in which the arguments may appear for the commutative instructions (e.g. `ADD`, `MUL` or `AND`).

Therefore, given a block B, its SFS $\mathcal{S}$ and its decoded optimized block B' together with its SFS $\mathcal{S}'$, B' is *verified* if $\mathcal{S}' \equiv \mathcal{S}$.

Example 4.1. *Let us consider the block shown in Figure 2.1 whose simplified SFS (described in Example 2.2) is* $\mathcal{S} = [\texttt{MLOAD}(64), 0, 139, \texttt{AND}(\texttt{SLOAD}(3), 1, 461501637 \cdot 10^{48} - 1), \texttt{AND}(s_0, 1, 461501637 \cdot 10^{48} - 1), s_0, s_1, s_2]$

The decoded optimized block obtained from the solver is the sequence of EVM instructions B' = `DUP1`, `PUSH20` $1.461501637 \cdot 10^{48} - 1$, `AND`, `PUSH1` 3, `SLOAD`, `PUSH20` $1.461501637 \cdot 10^{48} - 1$, `AND`, `PUSH1` 139, `PUSH1` 0, `PUSH1` 64, `MLOAD` *(for simplicity the pushed values have been represented as integer values instead of hexadecimal values). If we compute the SFS of the optimized block B' we obtain that* $\mathcal{S}' = [\texttt{MLOAD}(64), 0, 139, \texttt{AND}(1.461501637 \cdot 10^{48} - 1, \texttt{SLOAD}(3)), \texttt{AND}(1.461501637 \cdot 10^{48} - 1, s_0), s_0, s_1, s_2]$*. Thus, the optimized block B' is* verified *as* $\mathcal{S} \equiv \mathcal{S}'$*. Note that the arguments of the symbolic expressions involved in the bytecode* `AND` *appear in different orders in* $\mathcal{S}$ *and* $\mathcal{S}'$*. However, as the bytecode* `AND` *is commutative, the expressions* $\texttt{AND}(1.461501637 \cdot 10^{48} - 1, \texttt{SLOAD}(3))$ *and* $\texttt{AND}(\texttt{SLOAD}(3), 1.461501637 \cdot 10^{48} - 1)$ *produce the same result.*

Finally, when all the optimized blocks have been verified, we can reconstruct the bytecode of the optimized smart contract. We replace all the instructions within the blocks in the CFG of the contract by their optimized versions. Note that the size of the blocks may have changed and, hence, the jump addressed need to be updated accordingly. Such new jump addresses can be computed statically. Hence, the old jump addresses can be replaced by the new ones with a straightforward data-flow analysis that binds the addresses with their corresponding `PUSH` instruction.

4.2 Verification of *ASFS* equivalence in Dafny

In this section, the *SFS* equivalence algorithm is verified using a verification tool called Dafny. Instead of working directly with the *stack functional specification*, previous algorithm is adapted to the *ASFS*. It follows exactly the same idea: check all elements in the final stack are equivalent, and also that fresh variables refer to the same elements. The project source files can be found at `https://github.com/alexcere/sfs-equivalence-dafny`.

Dafny [24] is a programming language developed by Microsoft that allows specifying and verifying the functional correctness of programs. It supports the imperative programming paradigm, and allows specifying dynamic allocation and inductive datatypes. It also allows users to introduce pre and postconditions and termination metrics in order to prove the correction of programs.

Dafny syntax resembles the syntax of other conventional languages, such as C++ or C#. The built-in specification constructs are easy to follow and understand, so a proper explanation of them will be skipped to focus on the implementation provided in this section. A full explanation of the general behaviour of Dafny can be found in [38].

We just need to highlight the clear distinction between `predicate` and `function` declarations in Dafny, that contributes only to the specification and has no code equivalence whatsoever, and `method`, that includes the code to verify. Dafny also allows a hybrid between both approaches: `function method`, which allows defining a method and its specification at the same time, following the syntax of functions.

Knowing this distinction will allow us to distinguish those declarations that are just useful for specification and those for implementation and verification.

Dafny code can be compiled into popular programming languages, such as C#, JavaScript or Go. The C# implementation has been generated and included in the github repository. However, it has not been tested yet with real examples. The algorithm used in gasol corresponds to an implementation done entirely in Python.

4.2.1 *ASFS* specification

The specification of the *ASFS* is heavily based on the definitions and properties presented in Section 3.2. First of all, we need to build the algebraic datatype for representing stack elements:

```
datatype BasicTerm = Value(val: int) | StackVar(id: int)
```

Note that we make a clear distinction between numerical values and stack variables. Besides, stack variables id correspond to the subindex used when denoting stack variables s_i.

Thus, the elements of the *ASFS* can be defined as follows:

```
datatype ASFS = ASFS(input: seq<BasicTerm>, dict: map<int,StackElem>,
    output: seq<BasicTerm>)
```

The datatype definition does not give any further information on the properties an *ASFS* fulfills. Thus, we need to introduce predicates to specify the properties of this representation.

First of all, let us discuss the requirements of the initial stack $\mathcal{S}_0$: it only contains stack variables that are no repeated.

```
predicate initialInputIsWellDefined (input: seq<BasicTerm>)
{
    allVarsAreStackVar(input) ∧ noRepeatedStackVar(input, {})
}
```

The final stack or output stack $\mathcal{S}$ may contain both stack variables or numerical values. The only limitation is that stack variables either appear at the initial stack or as keys in the dict:

```
predicate outputIsWellDefined(inputStack: seq<BasicTerm>, dict: map<int,
    StackElem>, output: seq<BasicTerm>)
{
```

```
    ∀ elem • elem in output ⟹ match elem {case Value(x) ⇒ true case
        StackVar(id) ⇒ id in dict ∨ id in idsFromInput(inputStack)}
}
```

Now let us focus on the *operator map*. This representation is very tricky, as many properties need to be considered to obtain a representation to work with.

First of all, we need to state that every value in the map is well-defined *i.e.* it can be either a numerical value or a stack variable from $\mathcal{S}_0$ or from M keys:

```
predicate idsInDictAreWellDelimited(inputStack: seq<BasicTerm>, dict:
    map<int, StackElem>)
{
    ∀ key • key in dict ⟹
        match dict[key]
            case Op(l) ⇒
                (∀ i • 0 ≤ i < |l| ⟹ match l[i]{
                        case Value(x) ⇒ true
                        case StackVar(id2) ⇒ (id2 in idsFromInput(
                            inputStack) ∨ id2 in dict)})
            case COp(x1, x2) ⇒
                match x1 {
                        case Value(x) ⇒ true
                        case StackVar(id2) ⇒ id2 in idsFromInput(
                            inputStack) ∨ id2 in dict
                    }
                    ∧ match x2 {
                        case Value(x) ⇒ true
                        case StackVar(id2) ⇒ id2 in idsFromInput(
                            inputStack) ∨ id2 in dict
                    }
}
```

This previous predicate is needed to ensure convergence can be defined properly. Predicate `dictElementConverges` ensures Property 3.2 holds. Representing $<$ relation directly in Dafny is a really difficult task and not worthy. Instead, a *previously_ids* set is generated when invoking the predicate. This way, given a fresh variable s_i we check all its associated parameters. If any of them is another fresh variable s_j, we add s_i to that set and call the predicate recursively. Before proceeding, it is checked whether $s_j \in$ *previously_ids*. In that case, a loop has been found that can lead to infinite recursive definitions. Otherwise, it is followed that $s_j < s_i$ and this fact ensures the predicate eventually must converge. From this point, this reasoning can be repeated for every fresh stack variable and the convergence is followed.

```
predicate dictElementConverges(inputStack: seq<BasicTerm>, dict: map<
    int, StackElem>, key: int, previously_ids: set<int>)
decreases dict.Keys - previously_ids
requires previously_ids ≤ dict.Keys
requires key in dict
requires idsInDictAreWellDelimited(inputStack, dict)
```

```
{
    match dict[key]
        case Op(l)      ⇒
            key ∉ previously_ids ∧
            ∀ i • 0 ≤ i < |l| ⟹ match l[i] {
                case Value(x) ⇒ true
                case StackVar(x1) ⇒ if x1 in idsFromInput(inputStack)
                   then true else dictElementConverges(inputStack, dict,
                    x1, previously_ids + {key} )}
        case COp(el1, el2) ⇒
            key ∉ previously_ids ∧
            match el1 {
                case Value(x) ⇒ true
                case StackVar(x1) ⇒ if x1 in idsFromInput(inputStack)
                   then true else dictElementConverges(inputStack, dict,
                    x1, previously_ids + {key} )}
            ∧
            match el2 {
                case Value(x) ⇒ true
                case StackVar(x1) ⇒ if x1 in idsFromInput(inputStack)
                   then true else dictElementConverges(inputStack, dict,
                    x1, previously_ids + {key} )}
}
```

In order to specify Property 3.4, the function `dependentIds` is introduced. This function collects all fresh stack variables ids that depend on another fresh stack variable recursively, including itself. Thus, Property 3.4 is equivalent to stating that the union set obtained for all elements in $\mathcal{S}$ must correspond to the domain of M.

```
function method dependentIds(inputStack: seq<BasicTerm>, dict: map<int,
    StackElem>, key: int, previously_ids: set<int>) :  (sol: set<int>)
decreases dict.Keys - previously_ids
requires previously_ids ≤ dict.Keys
requires key in dict
requires idsInDictAreWellDelimited(inputStack, dict)
requires dictElementConverges(inputStack, dict, key, previously_ids)
ensures previously_ids ≤ sol
ensures sol ≤ dict.Keys
{
    match dict[key]
        case Op(l)      ⇒
            (set x,el | (el in l ∧ match el {case Value(x) ⇒ false
               case StackVar(x1) ⇒ if x1 in idsFromInput(inputStack)
               then false else true}
            ∧ x in dependentIds(inputStack, dict, getId(el),
               previously_ids + {key})) • x) + previously_ids + {key}
        case COp(el1, el2) ⇒
            (set x | (match el1 {case Value(x) ⇒ false case StackVar(x1
               ) ⇒ if x1 in idsFromInput(inputStack) then false else
```

```
            true}
        ∧ x in dependentIds(inputStack, dict, getId(el1),
            previously_ids + {key})) • x)
        +
        (set x | (match el1 {case Value(x) ⇒ false case StackVar(x1
            ) ⇒ if x1 in idsFromInput(inputStack) then false else
            true}
        ∧ x in dependentIds(inputStack, dict, getId(el1),
            previously_ids + {key})) • x)
        + previously_ids + {key}
}
```

Finally, all predicates and functions discussed above come together in the definition of a well-defined *map* operator. Only two properties were still left: Properties 3.1 and 3.3. They are directly stated in the following predicate:

```
predicate dictIsWellDefined(inputStack: seq<BasicTerm>, dict: map<int,
    StackElem>)
{
    (dict.Keys * idsFromInput(inputStack) = {}) ∧
        idsInDictAreWellDelimited(inputStack, dict)
    ∧ (∀ key • key in dict ⟹ dictElementConverges(inputStack, dict,
        key, {}))
    ∧ (∀ id • id in dict ⟹ match dict[id] {
        case Op(l) ⇒ Op(l) ∉ (dict - {id}).Values
        case COp(x1, x2) ⇒ COp(x1, x2) ∉ (dict - {id}).Values ∧ COp(x2
            , x1) ∉ (dict - {id}).Values })
}
```

Joining all requirements above, the *ASFS* can be fully specified:

```
predicate isASFS(asfs: ASFS)
{
    match asfs
        case ASFS(input, dict, output) ⇒ initialInputIsWellDefined(
            input) ∧ dictIsWellDefined(input, dict) ∧
            outputIsWellDefined(input, dict, output)
            ∧ (set x,id | id in dict ∧ x in dependentIds(input, dict,
                id, {}) • x) = dict.Keys
}
```

4.2.2 Equivalence verification

After the *ASFS* is fully specified, the equivalence algorithm can be introduced.

The first step is to include the definition of equivalence as a predicate in Dafny, so the output of the proposed algorithm can be verified according to this specification.

The specification is quite straightforward following Definition 4.1. Dafny finds it difficult to prove termination with mutually recursive functions in which only one of them actually decreases the termination measure and the other does not modify it. That is

why the `isEquivalent` method constantly repeats the same code for comparing two stack elements.

Basically, the comparison between two basic terms always follows the same scheme: if both of them are variables, then they must be the same. If both of them are stack variables, one of the following two possibilities must hold: either they both represent initial stack variables and share the same position in their respective stacks or they are fresh stack variables and represent the same operation. Otherwise, they cannot be equivalent.

```
case (Value(x1), Value(x2)) ⇒ x1 = x2
case (StackVar(x1), StackVar(x2)) ⇒
    if (x1 in idsFromInput(input1) ∧ x2 in idsFromInput(input2))
        then getPos(input1, x1) = getPos(input2, x2)
    else if (x1 in dict1 ∧ x2 in dict2)
        then compareStackElem(input1, input2, dict1, dict2, x1, x2,
            prev_ids1 + {key1}, prev_ids2 + {key2})
    else false
case (Value(x1), StackVar(x2)) ⇒ false
case (StackVar(x1), Value(x2)) ⇒ false
```

In order for this definition to be fully correct, a comparison between input parameters from functors must be set. This is the reason why we introduced an explicit separation between commutative and non-commutative operators: this way, it is states that commutative operators *always* correspond to binary operators, and thus, conmutativity is easier to check.

Also, when comparing stack elements, we have to ensure that the definition eventually converges. `dictElementConverges` states this fact. However, we need to provide an explicit `decreases` declaration, and thus, the same convergence sets must be considered as parameters.

Four different possibilities arise when comparing two entries at the stack:

```
predicate compareStackElem(input1: seq<BasicTerm>, input2: seq<
    BasicTerm>, dict1: map<int, StackElem>, dict2: map<int, StackElem>,
                        key1: int, key2: int, prev_ids1: set<int>,
                            prev_ids2: set<int>)
decreases |dict1.Keys - prev_ids1|
requires initialInputIsWellDefined(input1)
requires initialInputIsWellDefined(input2)
requires idsInDictAreWellDelimited(input1, dict1)
requires idsInDictAreWellDelimited(input2, dict2)
requires prev_ids1 ≤ dict1.Keys
requires prev_ids2 ≤ dict2.Keys
requires key1 in dict1 ∧ key2 in dict2
requires dictElementConverges(input1, dict1, key1, prev_ids1)
requires dictElementConverges(input2, dict2, key2, prev_ids2)
requires dict1.Keys * idsFromInput(input1) = {}
requires dict2.Keys * idsFromInput(input2) = {}
{
    match (dict1[key1], dict2[key2])
        case (Op(l1), Op(l2)) ⇒
```

```
            |l1| = |l2| ∧
            ∀ i • 0 ≤ i < |l1| ⟹
                match (l1[i], l2[i]) {
                    // ... comparison of elements
                }
        case (COp(el11, el12), COp(el21, el22)) ⇒
            // Checking all permutations
            (match (el11, el21) {
                    // ... comparison of elements
            } ∧
            match (el12, el22) {
                    // ... comparison of elements
            }) ∨
            (match (el11, el22) {
                    // ... comparison of elements
            } ∧
            match (el12, el21) {
                    // ... comparison of elements
            })
        case (COp(x1, y1), Op(l2)) ⇒ false
        case (Op(l1), COp(x2, y2)) ⇒ false
}
```

With this comparison between fresh stack elements, the equivalence notion is easily derived:

```
predicate areEquivalent(asfs1: ASFS, asfs2: ASFS)
requires isSFS(asfs1)
requires isSFS(asfs2)
{
    match (asfs1, asfs2)
        case (ASFS(input1, dict1, output1), ASFS(input2, dict2, output2
            )) ⇒ |input1| = |input2| ∧ |output1| = |output2|
        ∧ (∀ i • 0 ≤ i < |output1| ⟹ match (output1[i], output2[i])
        {
            case (Value(x1), Value(x2)) ⇒ x1 = x2
            case (StackVar(x1), StackVar(x2)) ⇒
                if (x1 in idsFromInput(input1) ∧ x2 in idsFromInput(
                    input2))
                    then getPos(input1, x1) = getPos(input2, x2)
                else if (x1 in dict1 ∧ x2 in dict2)
                    then compareStackElem(input1, input2, dict1, dict2,
                        x1, x2, {}, {})
                else false
            case (StackVar(x1), Value(x2)) ⇒ false
            case (Value(x1), StackVar(x2)) ⇒ false
        })
}
```

Now, let us consider the implementation and verification of the algorithm. The implementation is heavily based on the specification: two different methods have been defined resembling the two predicates used before. Whenever we detect two elements are not equivalent, we directly return false. Otherwise, at the end of all comparisons, the equivalence is ensured.

The first method compares recursively two fresh stack variables. The idea is really similar to the corresponding predicate, so we will not include the corresponding Dafny's code.

The second method explicitly compares two *ASFS*. No much is left to discuss, we will just introduce the implementation of the equivalence algorithm. Dafny manages to verify the implementation is indeed correct.

```
method areEquivalentASFS(asfs1: ASFS, asfs2: ASFS) returns (b: bool)
requires isASFS(asfs1)
requires isASFS(asfs2)
ensures b = areEquivalent(asfs1, asfs2)
{
    match (asfs1, asfs2)
        case (ASFS(input1, dict1, output1), ASFS(input2, dict2, output2
           )) ⇒
           if(|input1| ≠ |input2| ∨ |output1| ≠ |output2|){
               return false;
           }
           else {
               var i := 0;
               while i < |output1|
                decreases |output1| - i
               invariant 0 ≤ i ≤ |output1|
               invariant (∀ j • 0 ≤ j < i ⟹ match (output1[j],
                  output2[j])
                  {
                      case (Value(x1), Value(x2)) ⇒ x1 = x2
                      case (StackVar(x1), StackVar(x2)) ⇒
                          if (x1 in idsFromInput(input1) ∧ x2 in
                             idsFromInput(input2))
                              then getPos(input1, x1) = getPos(input2,
                                 x2)
                          else if (x1 in dict1 ∧ x2 in dict2)
                              then compareStackElem(input1, input2,
                                 dict1, dict2, x1, x2, {}, {})
                          else false
                      case (StackVar(x1), Value(x2)) ⇒ false
                      case (Value(x1), StackVar(x2)) ⇒ false
                  })
               {
                  match (output1[i], output2[i])
                  {
                      case (Value(x1), Value(x2)) ⇒
                          if(x1 ≠ x2) {
```

```
                        return false;
                    }
                case (StackVar(x1), StackVar(x2)) ⇒
                    if (x1 in idsFromInput(input1) ∧ x2 in
                        idsFromInput(input2))
                    {
                        if getPos(input1, x1) ≠ getPos(input2, x2
                            ){
                            return false;
                        }
                    }
                    else if (x1 in dict1 ∧ x2 in dict2)
                    {
                        var aux := compareDictElems(input1,
                            input2, dict1, dict2, x1, x2, {}, {});
                        if ¬aux{
                            return false;
                        }
                    }
                    else {
                        return false;
                    }
                case (StackVar(x1), Value(x2)) ⇒ return false;
                case (Value(x1), StackVar(x2)) ⇒ return false;
            }
            i := i + 1;
        }
        return true;
    }
}
```

Chapter 5

Implementation

A main contribution of this thesis is the development of the gasol tool. Nevertheless, other *sub-projects* have been developed as side projects: a web page hosted in COSTA group web page that contains an interactive version of the experiments and also Dafny's verification of the equivalence algorithm presented in Section 4.2. Specification and verification in Dafny was already discussed in that section, so it is skipped in this chapter.

In Section 5.1, the key aspects of gasol implementation are discussed. The gasol repository can be found in `https://github.com/costa-group/gasol-optimizer`.

Section 5.2 introduces many of the concepts discussed in the previous section applied to a concrete *sub-block*. The whole process is formalized, including the application of semantic rules, *ASFS* representation, the Max-SMT encoding and the interpretation of the final model. This *sub-block* has been directly extracted from an example smart contract in Remix [37].

In Section 5.3, the implementation of a interactive web page to show the experimental results is described. The full code, as well as the csv files that contain the information related to the experiments can be accessed in `https://github.com/alexcere/Syrup-Dash-Visualizer`. The web page can be accessed at `http://costa.fdi.ucm.es/syrup-visualizer`.

5.1 gasol implementation

gasol has a precursor in syrup 1.0, which in turn has a precursor in ebso [33]. Ebso was the first approach that combined SMT solvers with superoptimization of EVM code. This tool barely manages to optimize any code, due to its encoding relying too much in high-level order constructions. It is implemented in Ocaml, an ideal language for studying the correctness of certain approaches. In fact, this version contains the semantics of the EVM and manages to prove correctness of some of the semantics rules depicted in Appendix A. Ebso implementation can be found at `https://github.com/juliannagele/ebso`.

syrup 1.0 is also implemented in Ocaml and its code can be found in `https://github.com/mariaschett/syrup-backend/tree/master/lib`. This implementation includes the basic encoding provided in Section 3.4 and manages to improve greatly the results from ebso.

However, when designing gasol, we decided to implement it from scratch in Python. This way, it could be easily integrated with other tools developed by the COSTA group based on EVM analysis and optimization.

gasol accepts directly the assembly json-based representation obtained from the **solc** compiler or a *basic-block*. gasol implementation is clearly divided into two phases: the *SFS* generation and optimization, and solving Max-SMT problem. The *SFS* generation is heavily based on Ethir [7], an already function tool for generating a high-level rule-based representation from a complete Control Flow Graph. Thus, we will not discuss the implementation of this phase in detail.

Let us focus on the generation of the Max-SMT encoding problem. Firstly, we need to set the exchange format we are assuming as an input for this phase. As a result from previous phase, a json file is generated for each sub-block in the code. This json files are contained in the folder `/tmp/costabs/gasol/jsons`. They follow the same structure:

- max_progr_len: represents the parameter b_0.
- max_sk_sz: represents the parameter b_s.
- src_ws: list of stack variables (strings) that represent the initial contents in the stack. Each element is of the form *s(i)* and no index is repeated.
- vars: stack variables that are going to be used for representing the SFS.
- init_progr_len: lower bound for parameter b_0 that guarantees the Max-SMT problem is satisfiable. This bound can be inferred in certain cases when rules have been applied and we know beforehand simplifications lead to smaller *sub-blocks*.
- current_cost: gas associated to block without optimization.
- user_instrs: a lists of maps that represent the operator map of the *ASFS*. Each map contains the following information:
  - outpt_sk: generated fresh stack variable that represents the application of the function we are defining to the arguments in inpt_sk. It always contains exactly one stack variable.
  - commutative: boolean value that states whether the operation is commutative or not.
  - inpt_sk: list of stack elements that the function receives as input arguments to generate the output (output_sk).
  - gas: associated gas for the opcode.
  - disasm: EVM opcode.
  - opcode: opcode number.
  - id: id assigned to the opcode. As the same disassembly operation can appear several times to represent different outputs, id is used to distinguish among these operations.
- tgt_ws: list of elements that represent the output stack.

Note that there are several fields in this file that represent *abstract stacks* or *operation parameters*. In both cases, the leftmost element represents the top of the stack.

These fields match the ones presented in Section 3.2, so they can be easily derived from this representation. Once this information is read into the program, we need to generate

the SMT encoding. Our implementation is adapted to Z3, Barcelogic and OptiMathSAT. Barcelogic has no Python API, so for each of them we are going to generate a file with extension smt2 that contains the constraints.

The generation is quite straightforward following the constraints in Table A.1. Some options and syntax are slightly different depending on the solver considered. For instance, soft constraints in Barcelogic need to add "!" symbol after *assert-soft* declaration. An example of the contents of a smt2 file is included in Section 5.2.

Once the problem has been formulated, the solver is invoked through the corresponding command. The solver output is obtained and from it, it is determined whether we have found a model or not.

If such model was found, next step is rebuilding the solution. Knowing the conversion of θ dict makes the process fairly easy because it is just needed to take into account that `NOP` opcodes are dismissed and also that `PUSH` instructions must be transformed into the corresponding `PUSHx` instruction depending on the length of the word.

Results from this last phase are stored in `/tmp/costabs/gasol/solutions`. For each contract contained in the file, three different folders are generated as a result:

1. *disasm*: contains the solution in disassembly format of each *sub-block*.
2. *evm*: contains the generated bytecode of each sub-block generated in hexadecimal format. This representation is the one uploaded into the blockchain.
3. *total_gas*: contains the gas consumption of each generated block. It is useful to detect whether the equivalent *sub-block* gas is less than the initial one.

Future developments of the tool will focus on its integration on **solc** version. A prototype is currently being developed, that obtains as an input the disassembly asm version of the code and produces the same representation as an output. This representation requires more intermediate constructs to be taken into account. For instance, other variations of `PUSH` instructions are considered as disassembly instructions, such as `PUSH tag`, `PUSH data` or `PUSH DEPLOYADDRESS`. At the point the asm json version is generated, the compiler has not resolved the concrete value they represent, and thus, it introduces this new representation.

Nevertheless, adding these new constructions does not affect the theoretical framework developed in this thesis.

5.2 Case Study: *Owner* contract

In this section, a simple example is studied in full detail to understand the work-flow presented in last section. In order to do so, let us consider the simple Solidity program, which corresponds to *2_Owner.sol* contract presented as an example in Remix web page [37].

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.7.1;

/**
 * @title Owner
 * @dev Set & change owner
```

```
 */
contract Owner {

    address private owner;

    // event for EVM logging
    event OwnerSet(address indexed oldOwner, address indexed newOwner);

    // modifier to check if caller is owner
    modifier isOwner() {
        // If the first argument of 'require' evaluates to 'false',
           execution terminates and all
        // changes to the state and to Ether balances are reverted.
        // This used to consume all gas in old EVM versions, but not
           anymore.
        // It is often a good idea to use 'require' to check if
           functions are called correctly.
        // As a second argument, you can also provide an explanation
           about what went wrong.
        require(msg.sender == owner, "Caller is not owner");
        _;
    }

    /**
     * @dev Set contract deployer as owner
     */
    constructor() {
        owner = msg.sender; // 'msg.sender' is sender of current call,
           contract deployer for a constructor
        emit OwnerSet(address(0), owner);
    }

    /**
     * @dev Change owner
     * @param newOwner address of new owner
     */
    function changeOwner(address newOwner) public isOwner {
        emit OwnerSet(owner, newOwner);
        owner = newOwner;
    }

    /**
     * @dev Return owner address
     * @return address of owner
     */
    function getOwner() external view returns (address) {
        return owner;
    }
```

```
}
```


From this contract, 23 *sub-blocks* are delimited. Let us consider *sub-block* 133. The *ASFS* associated to this block is:


```
{"init_progr_len": 19, "max_progr_len": 19, "max_sk_sz": 5,
  "vars": ["s(0)", "s(1)", "s(4)", "s(5)"],
  "src_ws": ["s(0)", "s(1)"], "tgt_ws": [220, "s(4)"],
  "user_instrs": [
    {"id": "AND_0", "opcode": "16", "disasm": "AND",
      "inpt_sk": [1461501637330902918203684832716283019655932542975,
         "s(5)"],
      "outpt_sk": ["s(4)"], "gas": 3, "commutative": true},

    {"id": "CALLDATALOAD_0", "opcode": "35",
      "disasm": "CALLDATALOAD", "inpt_sk": ["s(1)"],
      "outpt_sk": ["s(5)"], "gas": 3,
      "commutative": false}], "current_cost": 54}
```


This is an interesting example to analyze. Firstly, it contains two uninterpreted functions, one commutative and one non-commutative. Besides, $b_0 = 19$, which gives us big room for improvement. If we analyze the initial *sub-block* instructions, it can be shown that rules have been applied before generating this representation:

$c :=$ `DUP2 : ADD : SWAP1 : DUP1 : DUP1 : CALLDATALOAD :`
`PUSH20 1461501637330902918203684832716283019655932542975 : AND : SWAP1 :`
`PUSH1 32 : ADD : SWAP1 : SWAP3 : SWAP2 : SWAP1 : POP : POP : POP : PUSH2 220`

Both `ADD` instructions do not appear in the *uninterpreted* operations of the *ASFS*. In this case, they have been removed due to final `POP` operations removing the obtained values from both instructions. The *ASFS* representation prevents them from appearing as *uninterpreted* operations, as no *noise* is allowed (Property 3.4).

The following phase corresponds to the generation of the Max-SMT problem. For the sake of simplicity and brevity, declarations are skipped and only instructions constraints for t_0 are included, including only `SWAP1` and `DUP1`:


```
; Logic used: Unquantified linear integer arithmetic
(set-logic QF_LIA)

; Variables assignment

(assert (= |s(0)| 11579208923731619542357098500868790785326998466564056
 403945758400791312963993 6))
(assert (= |s(1)| 11579208923731619542357098500868790785326998466564056
403945758400791312963993 7))
(assert (= |s(4)| 11579208923731619542357098500868790785326998466564056
403945758400791312963993 8))
(assert (= |s(5)| 11579208923731619542357098500868790785326998466564056
403945758400791312963993 9))
```

```
; Instructions constraints

(assert (and (<= 0 t_0) (< t_0 13)))

; Stack constraints

; After a nop, the following instruction must be also a NOP
(assert (=> (= t_0 2) (= t_1 2)))

  ; PUSH constraint
(assert (=> (= t_0 0) (and (<= 0 a_0) (< a_0
   1157920892373161954235709850086879078532699846656405640394575840079131296399360) (not u_4_0) u_0_1 (=
   x_0_1 a_0)
(and (and (= u_1_1 u_0_0) (= x_1_1 x_0_0)) (and (= u_2_1 u_1_0) (=
   x_2_1 x_1_0))
(and (= u_3_1 u_2_0) (= x_3_1 x_2_0)) (and (= u_4_1 u_3_0) (= x_4_1
   x_3_0))))))

; POP constraint
(assert (=> (= t_0 1) (and u_0_0 (not u_4_1) (and (and (= u_0_1 u_1_0)
(= x_0_1 x_1_0)) (and (= u_1_1 u_2_0) (= x_1_1 x_2_0)) (and (= u_2_1
   u_3_0)
(= x_2_1 x_3_0)) (and (= u_3_1 u_4_0) (= x_3_1 x_4_0))))))

; NOP constraint
(assert (=> (= t_0 2) (and (and (= u_0_1 u_0_0) (= x_0_1 x_0_0))
(and (= u_1_1 u_1_0) (= x_1_1 x_1_0)) (and (= u_2_1 u_2_0) (= x_2_1
   x_2_0))
(and (= u_3_1 u_3_0) (= x_3_1 x_3_0)) (and (= u_4_1 u_4_0) (= x_4_1
   x_4_0)))))

; DUP1 constraint
(assert (=> (= t_0 3) (and (not u_4_0) u_0_0 u_0_1 (= x_0_1 x_0_0) (
   and
(and (= u_1_1 u_0_0) (= x_1_1 x_0_0)) (and (= u_2_1 u_1_0) (= x_2_1
   x_1_0))
(and (= u_3_1 u_2_0) (= x_3_1 x_2_0)) (and (= u_4_1 u_3_0) (= x_4_1
   x_3_0))))))

; SWAP1 constraint
(assert (=> (= t_0 7) (and u_1_0 u_0_1 (= x_0_1 x_1_0) u_1_1 (= x_1_1
   x_0_0)
true (and (and (= u_2_1 u_2_0) (= x_2_1 x_2_0)) (and (= u_3_1 u_3_0)
(= x_3_1 x_3_0)) (and (= u_4_1 u_4_0) (= x_4_1 x_4_0))))))

; Commutative constraints
```

```
(assert (=> (= t_0 11) (and u_0_0 u_1_0 (or (and
(= x_0_0 1461501637330902918203684832716283019655932542975)
(= x_1_0 |s(5)|))(and (= x_0_0 |s(5)|)
(= x_1_0 1461501637330902918203684832716283019655932542975)))
u_0_1 (= x_0_1 |s(4)|) (and (and (= u_1_1 u_2_0) (= x_1_1 x_2_0))
(and (= u_2_1 u_3_0) (= x_2_1 x_3_0)) (and (= u_3_1 u_4_0) (= x_3_1
   x_4_0)))
(not u_4_1))))

; Non-commutative constraints
(assert (=> (= t_0 12) (and (and true (and u_0_0 (= x_0_0 |s(1)|)))
   u_0_1
(= x_0_1 |s(5)|) (and (and (= u_1_1 u_1_0) (= x_1_1 x_1_0)) (and (=
   u_2_1 u_2_0)
(= x_2_1 x_2_0)) (and (= u_3_1 u_3_0) (= x_3_1 x_3_0)) (and (= u_4_1
   u_4_0)
(= x_4_1 x_4_0))) (and true) (and true))))

; Initial stack constraints
(assert (and u_0_0 (= x_0_0 |s(0)|)))
(assert (and u_1_0 (= x_1_0 |s(1)|)))
(assert (not u_2_0))
(assert (not u_3_0))
(assert (not u_4_0))

; Final stack constraints
(assert (and u_0_19 (= x_0_19 220)))
(assert (and u_1_19 (= x_1_19 |s(4)|)))
(assert (not u_2_19))
(assert (not u_3_19))
(assert (not u_4_19))

; Soft constraints
(assert-soft (or (= t_0 2)) :weight 2 :id gas)
(assert-soft (or (= t_0 2) (= t_0 1)) :weight 1 :id gas)

; Stack: {'PUSH': 0, 'POP': 1, 'NOP': 2, 'DUP1': 3, 'DUP2': 4, 'DUP3
   ': 5, 'DUP4': 6, 'SWAP1': 7, 'SWAP2': 8, 'SWAP3': 9, 'SWAP4': 10}
; Comm: {'AND_0': 11}
; Non-Comm: {'CALLDATALOAD_0': 12}
```

In this example, some implementation key aspects surface. First of all, *smtlib format* does not allow to include directly variables in a format $s(i)$. Instead, bars need to be added at the beginning and the end of those values so the solver can accept them.

Secondly, it may seem strange that *true* values appear at some *and* clauses. Clearly, they do not add any information, but at the level of implementation, it is important to add them whenever we are working with $\bigwedge$ operations. SMT solvers do not accept empty *and* nor *or* clauses, and C_{SFS} can contain some empty $\bigwedge$ in some declarations.

The easiest way to avoid this type of errors is always adding the *identity element* in any of these arbitrary long constraints.

Finally, note that the last comment includes the corresponding theta representation. It has been added so constraints are understood more easily.

A solution for this model is generated and contains the following t_j and a_j:

```
sat

(objectives (gas 14))

( (t_0 1) )  ( (a_0 0) )
( (t_1 12) ) ( (a_1 0) )
( (t_2 0) )
   ( (a_2 1461501637330902918203684832716283019655932542975) )
( (t_3 11) ) ( (a_3 0) )
( (t_4 0) )  ( (a_4 220) )
( (t_5 2) )  ( (a_5 220) )
( (t_6 2) )  ( (a_6 220) )
( (t_7 2) )  ( (a_7 220) )
( (t_8 2) )  ( (a_8 220) )
( (t_9 2) )  ( (a_9 0) )
( (t_10 2) ) ( (a_10 220) )
( (t_11 2) ) ( (a_11 220) )
( (t_12 2) ) ( (a_12 220) )
( (t_13 2) ) ( (a_13 220) )
( (t_14 2) ) ( (a_14 220) )
( (t_15 2) ) ( (a_15 220) )
( (t_16 2) ) ( (a_16 220) )
( (t_17 2) ) ( (a_17 220) )
( (t_18 2) ) ( (a_18 220) )
```

Many of the t_j elements are assigned to $\theta(\texttt{NOP})$: in fact, the *sub-block* has been reduced from 19 instructions to only 5. This leads from an initial gas consumption of 54 units of gas to just 14.

The final solution corresponds to the next sequence:

$c' :=$ `POP : CALLDATALOAD :`
`PUSH20 1461501637330902918203684832716283019655932542975 : AND : PUSH1 220`

Thanks to the previous simplifications, many `SWAP` and `POP` opcodes have been rendered useless in the generation of the final stack, and thus, the program size has been reduced greatly.

5.3 Web page implementation

A web page has been designed to host the experimental results, so users can interact with different graphs and see the experimental section in full detail. It has been developed

using the Dash [36] library in Python.

Dash is an open-source framework for building web-based analytic applications and dashboards. It runs Flask applications and renders components using React. The main advantage of using Dash is that it allows creating complex apps easily directly from Python code. No Javascript or HTML code are needed, rather Dash contains different libraries with interactive web-based components.

It also allows including graphs from Plotly, one of the most famous Python libraries to generate graphs. In fact, graphs from Chapter 6 have been generated using this library.

Another main advantage of using Dash is that it provides simple reactive decorators for binding data analysis to Dash user interfaces. This way, when an input element changes, the Python code receives the new value of the input.

This component is the main basis on how the web page works. Each time an option is changed, a method is invoked with the new selected options, and the corresponding data is processed.

Let us see how it works with one of the sections in the web-page. In first section, two components can be modified: *solver* and *encoding*. Whenever any of them is modified, the following method is triggered:

```
@app.callback([Output('encoding-time', 'figure'), Output('encoding-gas', 'figure'),
               Output('encoding-statistics', 'figure')],
              [Input('solver', 'value'), Input('encoding', 'value')])
def update_stage_one(selected_solvers, selected_encodings):
    time_figure = plot_time(selected_solvers, selected_encodings)
    gas_figure = plot_gas(selected_solvers, selected_encodings)
    statistics_figure = plot_statistics(selected_solvers, selected_encodings)
    return time_figure, gas_figure, statistics_figure
```

in which *selected_solvers* parameter contains the selected values of component *solver*. As a result, the method returns the new generated graphs generated from the information. These graphs are generated processing a list of csv files contained in the project with the relevant information.

The best way to link selected options and the correspondent data they represent is by naming the csv files in a fixed way: for each combinations of solver and encoding, a csv file is contained in the project. For instance, the corresponding file for Z3 solver and *Initial encoding* is *initial_configuration_z3.csv*.

With this naming convention, generating the new graph is quite simple. For instance, for comparing times between encoding, the following method is used:

```
def plot_time(folder_name, encodings):
    fig = go.Figure()
    for encoding in encodings:
        times = np.empty(0, dtype=float)
        labels = []
        for name in folder_name:
            csv_name = str(DATA_PATH) + "/" + encoding + "_" + name + ".csv"
            df = pd.read_csv(csv_name)
            arr = df['time'].to_numpy() / 60
            times = np.append(times, arr)
            labels.extend([solver_name[name]] * len(arr))
        fig.add_trace(go.Box(y=times, x=labels, name=encoding_names[encoding]))
    fig.update_layout(
        yaxis_title='Times per contract (minutes)',
        boxmode='group' # group together boxes of the different traces for each
```

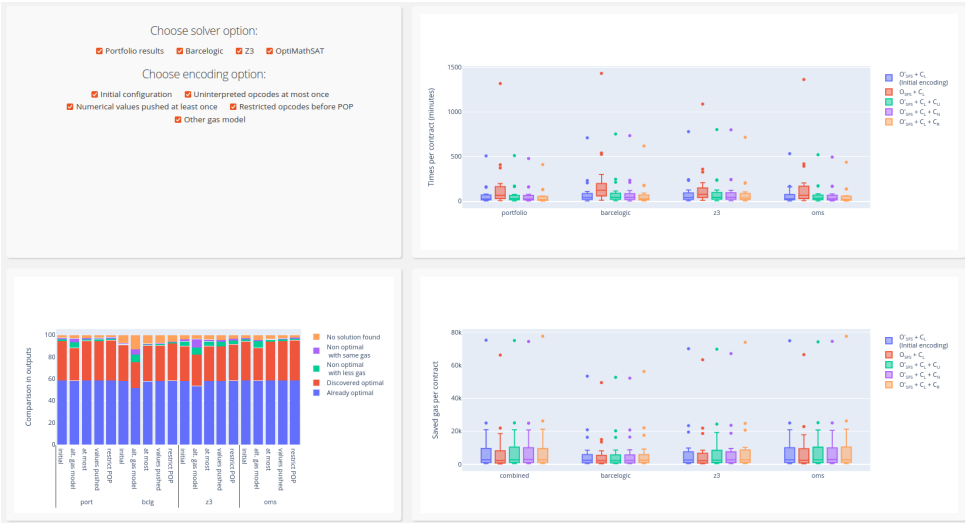

Figure 5.1: Usual structure in every section: configuration selection (top left), time comparison (top right), output comparison (bottom left) and gas comparison (bottom right)

```
            value of x
    )
    return fig
```

In order to deploy the app online, a gunicorn [16] server has been used. Gunicorn is a pre-fork web server model in which a master creates forks to handle each request. The main advantage of using gunicorn is that it allows to deploy apps directly by just specifying the number of workers and the app it has to deploy. In our case, no many people are expected to access the web page at the same time, and thus, only four workers have been fixed.

Once the implementation details are clear, let us discuss the structure of the web page. The web page is divided in three main sections, following the structure in Chapter 6: determining the best encoding, determining the most suitable timeout and comparison with syrup 1.0.

Most subsections follow the same structure: compare the performance of different configurations using all solvers and the best results from them and a classification depending on whether results have been optimized and whether they have proven to be optimal. Figure 5.1 shows an example of this structure.

However, the first section contains additional graphs. These graphs correspond to determine if C_U or C_N work better combined with C_R in certain situations. The situations analyzed are depicted in Figure 5.2. By manual inspection of these graphs, promising situations have been identified in the first subsection. Second subsection checks whether they do really improve time results significantly.

The remaining phases are explained in more detail in Chapter 6.

Figure 5.2: Section 1.2: Comparison of different static parameters for determining optimal combinations of encodings (top). Section 1.3: Comparison of selected candidate parameters to check if gas is saved using different encodings (bottom)

Chapter 6

Experimental Evaluation

This section presents the results of our experimental evaluation using gasol, GAS Optimization tooLkit that implements the approach presented in this thesis.

6.1 Experimental setup

Our tool GASOL: GAS Optimization tooLkit uses two external components:

- EthIR [7] to generate the CFGs of the analyzed contracts and,
- as SMT solvers, Z3 [15] version 4.8.7, Barcelogic [9], and OptiMathSAT (OMS) [14] version 1.6.3 (which is the optimality framework of MathSAT),

We carry out our experiments on two versions of gasol:

- syrup 1.0: corresponds to the predecessor of gasol that uses the basic encoding in Section 3.4 plus C_L in Section 3.4.1 with a fixed timeout of 15 min per block.
- gasol: corresponds to the latest version of our tool that includes the new encodings presented described in Section 3.4.1 and the timeouts described in Section 6.3.

The main components of gasol are implemented in Python. Our tool accepts smart contracts written in versions of Solidity up to 0.8.3 and EVM bytecode v1.10.1[1]. The experiments have been performed on an Intel Core i7-7700T at 4.2 GHz x 8 and 64 GB of memory, running Ubuntu 16.04. The aim of the experiments is twofold: on one hand we aim at finding the right balance between optimization and performance that allows us to integrate our tool as part of a real compiler, and on the other hand we want to assess the gas gains and the optimization time. To these ends, our experimental evaluation is separated in three steps (as shown in Figure 1.2):

(i) In the first step in Section 6.2, we analyze 5 different configurations that vary in the constraints used in order to find out which encoding obtains the best results, i.e., a larger amount of saved gas and more blocks being optimized.

(ii) In the next step in Section 6.3, once we have found the best configuration, we execute it with 5 different timeouts (1s, 10s, 15s, 30s, and 60s) in order to find the best precision vs. performance trade-off.

[1]Latest versions released up to March 2021

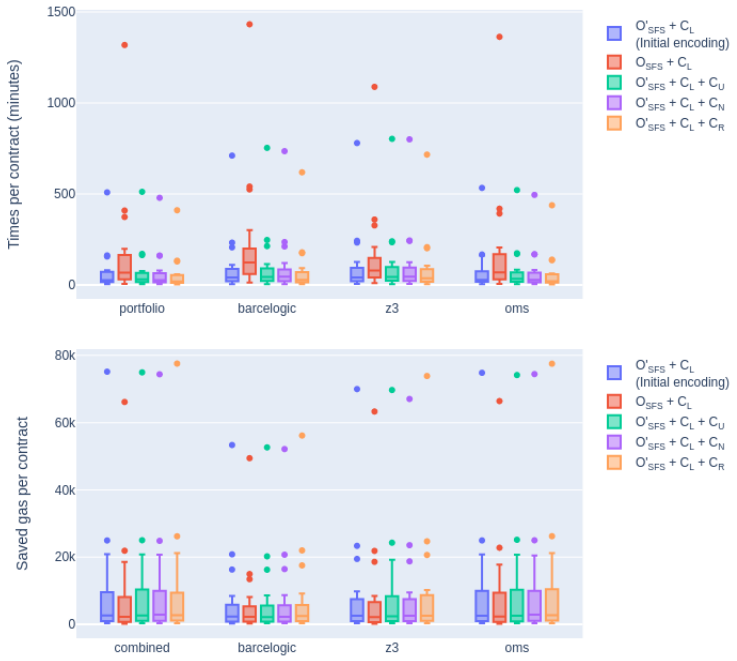

Figure 6.1: Comparison of the encodings studied depending on the time spent (top) and the gas saved (bottom) per contract.

(iii) In the final step in Section 6.4, we execute gasol with the best encoding configuration and the timeout selected from the previous steps and compare the gains in terms of performance and gas savings w.r.t. syrup 1.0.

The results obtained can be visualized at `http://costa.fdi.ucm.es/syrup-visualizer`.

As regards the considered data set, our focus in on analyzing the most called contracts. According to [21], this corresponds to the most relevant case study as many Ethereum contracts are not used. We consider two groups of benchmarks:

(a) For experiments (i) and (ii) above, we use a subset of 20 smart contracts taken from the 100 most called contracts deployed on Ethereum blockchain[2] whose source code is available. The smart contracts analyzed were selected uniformly depending on the number of blocks. In order to get realistic results, we analyze for each contract, the same bytecode stored on the blockchain, i.e., we compile the source code with the same optimization configuration used when they were deployed. In total, we have 16 261 blocks to analyze.

(b) For experiment (iii), we consider a larger set and analyze the 128 most called contracts[3]. This results in a data set of 46 966 blocks.

6.2 Analysis of the encodings

Our aim in this section is to investigate which of the 4 different encodings described in Section 3.4.1 and of the 2 gas encodings described in Section 3.4.2 obtains better results

[2]up to Ethereum blockchain block number 11 795 097 2021-02-05 08:04:14 UTC
[3]up to Ethereum blockchain block number 9 193 265 until 2019-12-31 23:59:45 UTC

both in terms of saved gas and total time spent. Figure 6.1 shows the results obtained in terms of time and gas when using the three solvers and the best results by considering them as a portfolio. Let us analyze the main results when comparing the two gas encodings in Section 3.4, including also C_L (equation 3.1 in Section 3.4.1) in both cases. As it can be seen, it is clear that $O'_{SFS} + C_L$ (blue box) works better than $O_{SFS} + C_L$ (red box) both in terms of time spent and gas saved. In the case of portfolio, $O'_{SFS} + C_L$ needs 39.33% less time to analyze the contracts and optimizes 16.76% more gas. Thus, we will directly assume this gas encoding for evaluating the remaining instruction encodings. As the whole encoding matches the configuration of syrup 1.0, we will also refer to it as *Initial encoding*.

Figure 6.1 shows that including C_R (yellow box) obtains better results than only considering *Initial encoding*, both in time spent and gas saved for the three solvers. In fact, when considering results from portfolio, it needs 21.64% less time to analyze the contracts and optimizes 2.7% more gas. The results of adding the two other encodings (C_U and C_N) are similar to the ones in *Initial encoding*: $O'_{SFS} + C_L + C_U$ (green box) takes 0.98% longer, whereas $O'_{SFS} + C_L + C_N$ (purple box) takes 3.8% less time. In terms of saved gas, in both cases the amount of gas saved is less than 1%. However, we investigate if the success of applying such encoding many depend on the characteristics of the SFS. In particular, we try now to determine possible scenarios in which, depending on parameters of the initial SFS, adding one of these encodings works better than only considering the initial one. These parameters consist of: the number of EVM instructions b_0 of the block analyzed, the number of uninterpreted operations, the number of necessary `PUSH` instructions and a lower bound inferred statically that represents the number of necessary EVM instructions without taking into account `SWAP`k instructions. We now rerun the experiments adding C_R by default, since C_R has been already proven to be very effective and will be part of the final encoding. We also consider only OptiMathSAT at this point, as it usually gets the best results from the solver. We have selected those blocks that take 80% less time compared to the same block when only considering C_R and studied the parameters from these contracts in order to find some patterns that could affect the time saved per encoding. We found out that when $b_0 \leq 15$, adding C_N saves a significant amount of time (14.77%) w.r.t the version not considering it. Also, if we denote the lower bound of EVM instructions as lb, when $lb/b_0 \geq 0.7$, adding C_N saves 5.4% of total time. We studied this quotient as the bigger this relation is, the lesser the percentage of `SWAP`k can appear in a valid sequence of opcodes. Therefore, we add these specific cases to our new default encoding. This final encoding leads to savings of 26.33% in terms of time and 3.40% in terms of gas compared to the initial one. Hence, it improves over having C_R only and therefore, we will assume this encoding by default in the remaining sections, denoting the whole new encoding as *Final encoding*.

6.3 Analysis of the time

In this second step, we consider the dataset (a) again and analyze the results produced by gasol (shown in Figure 6.2) when considering the portfolio results on the 16 261 basic blocks with a timeout of 1 second (s), 10s, 15s, 30s and 60s. As expected, as long as the timeout is increased, more gas is saved but also more time is spent. Our objective is to find the best trade-off between them for the future inclusion of the chosen setup in a compiler. Table 6.1 shows these results in detail. Rows **Avg. gas** and **Std. Dev.**

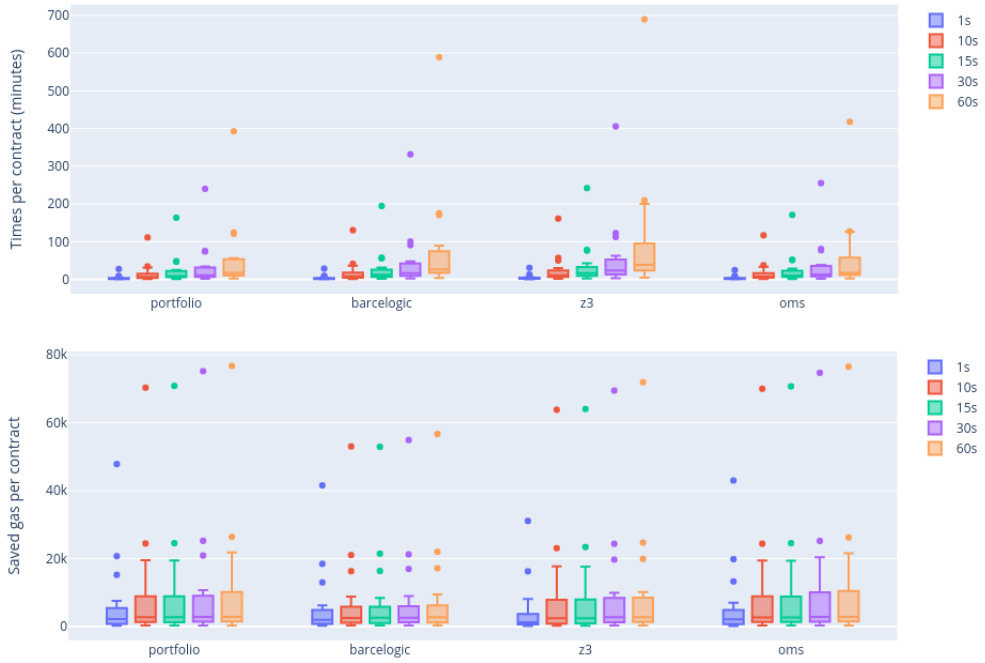

Figure 6.2: Comparison of the different timeouts studied depending on the time spent (top) and the gas saved (bottom) per contract.

gas correspond to the amount of gas saved on average per contract and the standard deviation respectively. Rows **Avg. time** and **Std. Dev. time** correspond to the same parameters when considering the time in minutes spent per contract. Row **% Losses in gas** refers to percentage of total gas not optimized when comparing each version with the most expensive one (60s timeout) and row **% Time saved** displays the time saved per timeout when compared with 60s.

Let us first focus on the 1s results. It manages to save 92.54% of time and optimize 64.94% of the total optimized gas. This means we succeed to get most of the contracts optimized in a fraction of the total time, leading to an average of 3.72 minutes (min) per contract. Therefore, it would be a feasible option to consider this timeout for integrating optimization in a compiler. However, 10s outshines the results from this category: only 8.50% of the gas is not optimized, whereas it still saves 71.55% of the total time. The time per contract increases to 14.19 min, which is a large amount of time for optimizing in a compiler. Nevertheless, as smart contracts are immutable and are stored indefinitely in the blockchain, a final longer compilation time can be assumed as the improvement in the optimization part clearly pays off. Therefore, we assume this *10s* timeout by default in our setup. Remaining timeouts take too long and do not improve gas saving too much, so we can directly dismiss them.

After fixing the encoding and the timeout, we have run the verifier with the solutions obtained in the portfolio setting. 15 685 blocks have been verified, resulting in 100% of them being verified correctly. This phase takes 0.85s, which corresponds to a negligible 0.0076% of the total execution time.

Tout	Avg. gas	Std. Dev. gas	Avg. time	Std. Dev. time	% Losses in gas	% Time saved
1s	6095.3	11154.21	3.72	6.07	35.06%	92.54%
10s	8587.55	15908.73	14.19	24.66	8.50%	71.55%
15s	8620.1	16017.41	20.80	36.20	8.16%	58.30%
30s	9038.15	16970.71	30.40	53.77	3.07%	39.04%
60s	9385.55	17360.89	49.87	88.08	0%	0%

Table 6.1: Comparison in performance among different timeouts

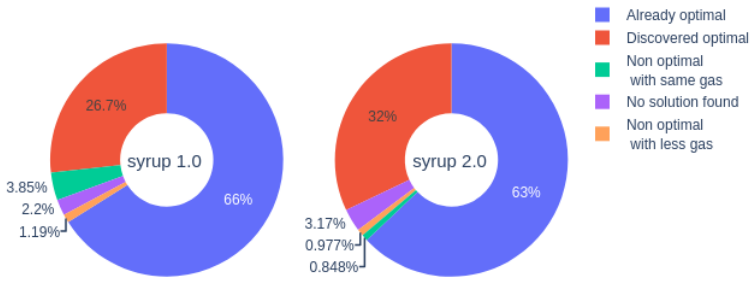

Figure 6.3: Comparison between syrup 1.0 and gasol at block level.

6.4 Analysis of the most called contracts in precision and time

In our last phase, we compare the results obtained by gasol with those obtained by syrup 1.0 (as they were reported in [8]) on the analysis of 46 966 blocks belonging to the 128 (most called) smart contracts (i.e., benchmark (b) in Section 6.1). As already mentioned, syrup 1.0 includes *Initial encoding* and a timeout set to 15 min, whereas gasol includes *Final encoding* and a timeout set to 10s.

Figure 6.3 shows the information related to the analysis of the blocks considering the solvers as a portfolio. There are small differences between both versions as about 95% of the results are obtained within 10-15s. The number of blocks that the solvers prove to be already optimal (i.e., those that cannot be optimized because they already consume the minimal amount of gas) are reduced by 3%. This is due to two factors: the modification of the timeouts and the integration of new simplification rules. Note that, there are blocks that were already optimal according to the previous setting, but now can be further optimized using some of the new simplification rules. Those blocks are now included in the set of *Discovered optimal*. On the other hand, due to the new encodings proposed in Section 3.4.1, we increase the number of blocks that we optimize and are proven to be optimal by 6%, and decrease the number of blocks where the solvers find a solution that consumes less (0.2%) or the same amount of gas (near to 2.88%) than the original block, but they are not able to prove optimality. Finally, the cases in which the solvers are not able to find a model are increased in 0.97% because of the reduction of the timeout.

Figure 6.4 shows the results obtained in terms of gas savings by each of the 128 smart contracts analyzed for the portfolio. As it can be seen, gasol optimizes more gas in most

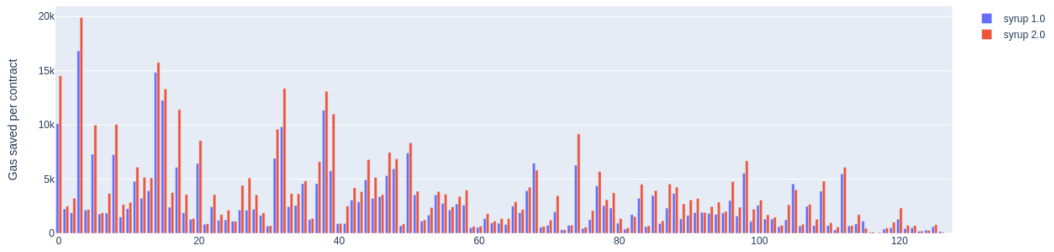

Figure 6.4: Comparison of the gas savings achieved by syrup 1.0 and gasol.

of the contracts, saving 30.3% more of gas because of the improvements included in the new setup. From such savings, 66% of them are achieved due to the application of the new simplification rules (we have included 33 new rules) described in Section 2.3 and Appendix A, and 34% due to the optimization process carried out by the solvers. Note that the simplification rules induce also more optimizations at the stack level. On average, the contracts saved 3161.57 units of gas in syrup 1.0 and now they save 4512.25 units of gas. In addition, we also reduce the time used by each contract. In the previous version, the portfolio setting of the solvers used on average 309,70 minutes (note that we had a timeout of 15 minutes) and now they are executed in 4,55 minutes per contract. This measure may fluctuate depending on the size and the number of blocks that the analyzed contract has. However the results obtained show that it is a realistic setting to be integrated in a real compiler. The minimum time that a contract has needed to be analyzed is less than a second while the maximum time needed has been 37.5 minutes for a contract with more than 5,000 blocks.

Chapter 7

Related Work

There are currently several approaches to gas optimization of Ethereum smart contracts. The closest to ours is the blockchain superoptimizer ebso [33], based on unbounded superoptimization [20]. The goal of ebso is the same as ours: find gas-optimal blocks in the CFG of a smart contract. While the approach of [33] would not be applicable within a compiler due to a high number timeouts, our optimization tool performs very efficiently. This efficiency is due to the fundamental differences with [33]: 1. we use the SFS to solve the optimization problem as a synthesis problem, in which the optimizations related to the semantics of the EVM instructions are carried out within the SFS through simplification rules, and thus 2. do not encode the semantics of the arithmetic and bit-vector operations in the SMT problem, which allows us to express the problem using only existential quantification, and 3. we use Max-SMT instead of repeated calls to the SMT solver. The Solidity compiler [3] also includes an optimizer that can work at two levels: assembly code and Yul representation [4]. In the case of assembly code, the optimizer splits the code in basic blocks and applies simplification rules as we do on the SFS side. In addition, the Yul optimizer is able to perform other optimizations beyond super-optimization such as function inlining, and remove redundant expressions or dead code. In spite of the fact that all SFS simplification rules included in gasol are also implemented by the Solidity optimizer[1], gasol is able to apply them in a higher number of cases due to the symbolic execution carried out during the generation of the SFS that goes beyond matching the rule patterns. For instance, in the sequence of EVM instructions `PUSH1 3 PUSH1 1 SWAP1 SLOAD SWAP1 DIV`, by applying symbolic execution gasol realizes that the divider is 1 and hence the instruction `DIV` has no effect. This is not detectable by just applying the rule patterns. Analyzing the contracts in benchmark (a) described in Section 6 without any optimization of the compiler, gasol applies 3412 simplification rules. On the other hand, if gasol analyzes the optimized version of these contracts, it identifies 306 rules that are not captured by the Solidity optimizer.

The system Gasol: Gas AnalysiS and Optimization tooL for Ethereum smart contracts [5] [2], developed by the COSTA group, incorporates an automatic optimization for storage operations that consists of replacing accesses to the storage (`SSTORE` and `SLOAD`) by equivalent accesses to memory locations (`MSTORE` and `MLOAD`), when a static analysis identifies that such transformation is sound and efficient. This optimization is not achievable by our approach as it involves the modification of multiple blocks, and also

[1] https://github.com/ethereum/solidity/blob/develop/libevmasm/RuleList.h

[2] We use different fonts to distinguish both tools. We are planning to integrate them in the same tool in the future.

requires an analysis that identifies the patterns and the soundness of the transformation. On the other hand, GASOL is not able to perform the intra-block optimizations that we are achieving. Therefore, the optimizations in GASOL are orthogonal (and complementary) to those achievable by means of superoptimization. The tool Gasper [12] replaces 7 expensive patterns in Solidity contracts with respect to useless code (dead code, opaque predicates), and loops (e.g., expensive operations in loops). In contrast to our approach, Gasper does not automatically identify expensive patterns. Similarly, Brandstaetter *et al.* [11] analyze the applicability of "optimization strategies" like loop unrolling, parallel computation, reordering tests, exploiting algebraic identities from software engineering on 3k Solidity smart contracts, and supply a tool to detect and partially optimize the code. Again, their work contrasts to ours, as they do not rely on automation to determine expensive patterns.

There are two rule-based approaches to optimize straight line code. First, Chen *et al.* [13] identify 24 anti-patterns by manual inspection and provide a tool, GasReducer, to eliminate these anti-patterns. Our tool subsumes all of their anti-patterns concerning stack layout/commutativity. Their work shows how important stack optimizations are, but also the difficulty of capturing interleavings in anti-patterns. However, we cannot support 2 anti-patterns identified in [13]: we are not able to collapse n `JUMPDEST`s nor replace `OP STOP` by `STOP` for `OP` not a jump instruction, as those relate to control flow. Second, Schett *et al.* [40] generate peephole optimization rules for EVM bytecode based on optimizations found in ebso [33]. Both approaches are—because of the fixed set of rules—less flexible than our approach for optimizing the stack. However, we do leverage this work on anti-patterns and rules in our framework: they are found in our simplification rules for optimization while generating the SFS.

There is work that experimentally proves that the gas model for some EVM instructions is not correctly aligned with respect to the observed computational costs [44], which can lead to gas-related attacks [35]. Our work is parametric in the gas model used, and new adjustments to the gas model of Ethereum are easily integrated.

Superoptimization has also been successfully employed, *e.g.*, when optimizing machine code with TOAST [10] leveraging answer set programming, or for LLVM [23] with the superoptimizer Souper [39]. Similar to our verifier in Section 4.1, Souper validates soundness, e.g., with Alive [25]. Mukherjee *et al.* [31] extend Souper for LLVM by heuristics to prune the search space to reduce calls to SMT solver to check the equivalence between a candidate program and the original program. We avoid these repeated calls by using Max-SMT and by shifting the search into the solver.

Finally, we believe our approach extends to bytecode of other smart contract languages with a clear cost model. Tezos's Michelson [22] is also a stack-based language, but features high-level data types, like lists, sets, and maps. Move [42] for Facebook's blockchain also has a stack-based machine model—but also features typed locals. To adapt the presented approach, the SMT encodings would need to be extended to incorporate types and locals, and data types need to be handled in the SMT encoding and the SMT solvers, to support complex theories such as sets and lists.

We have adopted a lightweight verification approach in which the outcome of each optimization is verified, rather than formally verifying the optimization tool itself. The latter would be a rather complex process and besides, as it relies on the use of SMT solvers, the whole pipeline of tools used for the optimization could not be verified. This way we have been able to detect and fix many bugs in the implementation and we have a high level of assurance of its correctness by now.

Chapter 8

Conclusions and Future Work

We have presented a novel method for gas super-optimization of smart contracts that combines symbolic execution with an effective Max-SMT encoding. Our focus is on the stack operations (`DUP`, `SWAP`, `PUSH` and `POP`) because these bytecode operations allow for multiple reorderings, simplifications, and cover the major part of the potential optimizations; while reading and/or writing on memory or storage can be seldom optimized (unless the same value is written, or read, consecutively).

In spite of this, the same methodology we have formalized for the stack could be extended to optimize the memory and storage bytecode operations.

Basically, the symbolic execution phase would extract a functional specification also for memory and for storage that would be analogous to our SFS and that could include storage-related optimizations (e.g., detecting unnecessary storage). The SMT encoding for these operations would be similar to ours but, for soundness, would have to maintain the order among the memory and storage accesses. It is part of our future work to implement also the super-optimizations for memory and storage and experimentally evaluate if there is significant gain.

We have also presented a theoretical framework to reason the process is correct. So far, *ASFS* equivalence and some equivalence rules have been fully proven using verification tools. Nevertheless, we are planning to formally prove the of correctness other phases, such as the *SFS* generation or the translation from *SFS* to *ASFS*.

Proving correctness is crucial for integrating gasol into widely used tools. Vulnerabilities in smart contracts lead to major economic loses that cannot be overcome due to their *immutability*. Hence, ensuring and minimizing risks becomes an essential requirement for any tool related to smart contracts.

gasol integration into Yul compiler is also one of the main lines of research for a near future. This approach has shown it is totally viable and worthy in terms of time spent and gas saved. Max-SMT solver ensures optimality in many of the analyzed examples, and thus, the data extracted from the experiments provides a insightful view into the strengths and weaknesses of state-of-the-art optimizers. We have agreed to provide this information to the Ethereum foundation so that they can enhance their compiler and detect possible *under-optimized* situations. Our approach has proven to be able to apply certain rules in situations Yul compiler cannot, so we expect this information to be really valuable once processed and analyzed.

Other crucial issue that has been considered is ensuring the process is somehow *deterministic*. The usage of SMT solvers prevent gasol from ensuring this property holds, as arbitrary heuristics and decisions are applied throughout the process of solving the Max-

SMT problem. Nevertheless, we are planning to generate a log file along the optimization containing the solution of the Max-SMT problem per *sub-block*. A flag will be added to gasol to accept this log, check the information contained is indeed correct, and generate the same optimized code from it.

Bibliography

[1] Etherscan. `https://etherscan.io`, 2018.

[2] Rattle - An EVM Binary Static Analysis Framework, 2018. `https://github.com/crytic/rattle`.

[3] Solidity documentation, 2021. `https://docs.soliditylang.org/en/latest/index.html`.

[4] Yul documentation, 2021. `https://docs.soliditylang.org/en/latest/yul.html`.

[5] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. In Armin Biere and David Parker, editors, *Proceedings of 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2020*, volume 12079 of *Lecture Notes in Computer Science*, pages 118–125, 2020.

[6] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Don't run on fumes—parametric gas bounds for smart contracts. *Journal of Systems and Software*, 176:110923, 2021.

[7] 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 *LNCS*, pages 513–520. Springer, 2018.

[8] Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria A. Schett. Synthesis of super-optimized smart contracts using max-smt. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 177–200. Springer, 2020.

[9] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic SMT solver. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, July 7-14, 2008, Proceedings*, pages 294–298, 2008.

[10] Martin Brain, Tom Crick, Marina De Vos, and John P. Fitch. TOAST: applying answer set programming to superoptimisation. In *Logic Programming, 22nd International Conference, ICLP 2006, Proceedings*, volume 4079 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2006.

[11] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski. Characterizing Efficiency Optimizations in Solidity Smart Contracts. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 281–290, 2020.

[12] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *SANER*, pages 442–446. IEEE Computer Society, 2017.

[13] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. Towards saving money in using smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 81–84, 2018.

[14] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013. Proceedings*, pages 93–107, 2013.

[15] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[16] Gunicorn Developers. Gunicorn. `https://gunicorn.org/`, 2021.

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

[18] L M Goodman. Tezos: A Self-Amending Crypto-Ledger Position Paper. page 18, 2014.

[19] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *PACMPL*, 2(OOPSLA):116:1–116:27, 2018.

[20] Abhinav Jangda and Greta Yorsh. Unbounded superoptimization. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, pages 78–88, 2017.

[21] Lucianna Kiffer, Dave Levin, and Alan Mislove. Analyzing ethereum's contract topology. In *Proceedings of the Internet Measurement Conference 2018, IMC 2018*, pages 494–499, 2018.

[22] Nomadic Labs. Michelson: The language of Smart Contracts in Tezos. `https://www.michelson-lang.com`, 2018.

[23] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, March 2004. IEEE Computer Society.

[24] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[25] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Practical Verification of Peephole Optimizations with Alive. *Commun. ACM*, 61(2):84–91, January 2018.

[26] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS*, pages 254–269. ACM, 2016.

[27] Henry Massalin. Superoptimizer - A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 122–126, 1987.

[28] Libra Association Members. White paper: The Libra Blockchain. page 29, 2019.

[29] Christophe Meudec. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.

[30] Bernhard Mueller. Smashing Ethereum Smart Contracts for Fun and Real Profit.(2018). In *The 9th annual HITB Security Conference*, 2018.

[31] Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. Dataflow-based pruning for speeding up superoptimization. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–24, November 2020.

[32] Mayukh Mukhopadhyay. *Ethereum Smart Contract Development*. Packt publishing, 2018.

[33] Julian Nagele and Maria A Schett. Blockchain superoptimizer. In *Preproceedings of 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019)*, 2019. `https://arxiv.org/abs/2005.05912`.

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

[35] Daniel Pérez and Benjamin Livshits. Broken metre: Attacking resource metering in EVM. *CoRR*, abs/1909.07220, 2019.

[36] Plotly Project. Dash open source. `https://plotly.com/dash/`, 2021.

[37] Remix Project. Remix ide. `https://remix.ethereum.org/`, 2021.

[38] Rubén Rafael Rubio Cuéllar. Verificación de algoritmos y estructuras de datos en dafny. Trabajo de Fin de Grado en Ingeniería Informática y Matemáticas (Universidad Complutense, Facultad de Informática, curso 2015/2016), 2016.

[39] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A Synthesizing Superoptimizer. *arXiv:1711.04422 [cs]*, November 2017.

[40] Maria A. Schett and Julian Nagele. Populating the Peephole Optimizer of a Smart Contract Compiler. In Bruno Bernardo and Diego Marmsoler, editors, *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *OpenAccess Series in Informatics (OASIcs)*, pages 3:1–3:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[41] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with Scilla. In *34th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2019)*, 2019.

[42] The LibraBFT Team. Move: A Language With Programmable Resources. Technical report, Novi, 2020. `https://developers.libra.org/docs/state-machine-replication-paper`.

[43] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2019.

[44] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing ethereum's gas mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2019, Stockholm, Sweden, June 17-19, 2019*, pages 310–319, 2019.

[45] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart Contract Repair. *ACM Transactions on Software Engineering and Methodology*, 29(4):27:1–27:32, September 2020.

Appendix A

Simplification rules

This section contains the complete list of simplification rules integrated in gasol. The rules marked as $\otimes$ in Figure A.1 correspond to those that were previously implemented in the first prototype of gasol [8] while the others have been added to GASOL: GAS Optimization tooLkit. The rules marked with $\odot$ have not been verified by ebso optimizer [33] due to its limitations. Note that we have omitted the commutative cases.

(1)⊗	$OP(X_{int}, Y_{int}) = eval(OP, X_{int}, Y_{int})$	(34)⊗	$ISZERO(ISZERO(ISZERO(X))) = ISZERO(X)$
(2)⊗	$OP(X_{int}) = eval(OP, X_{int})$	(35)	$ISZERO(XOR(X,Y)) = EQ(X,Y)$
(3)⊗	$ADD(X,0) = X$	(36)⊗	$ISZERO(ISZERO(GT(X,Y))) = GT(X,Y)$
(4)⊗	$SUB(X,0) = X$	(37)⊗	$ISZERO(ISZERO(LT(X,Y))) = LT(X,Y)$
(5)	$SUB(X,X) = 0$	(38)	$ISZERO(ISZERO(EQ(X,Y))) = EQ(X,Y)$
(6)⊗	$MUL(X,0) = 0$	(39)⊙	$SHL(X,0) = 0$
(7) ⊗ ⊙	$MUL(X,1) = X$	(40)⊙	$SHL(0,X) = X$
(8)⊙	$MUL(SHL(X,1),Y) = SHL(X,Y)$	(41)⊙	$SHR(0,X) = X$
(9)⊙	$MUL(X,SHL(Y,1)) = SHL(Y,X)$	(42)⊙	$SHR(X,0) = 0$
(10)⊗	$DIV(X,X) = 1$	(43)	$NOT(NOT(X)) = X$
(11)⊗	$DIV(X,1) = X$	(44)⊗	$XOR(X,X) = 0$
(12)	$DIV(X,0) = 0$	(45)	$XOR(X,0) = X$
(13)⊙	$DIV(X,SHL(Y,1)) = SHR(Y,X)$	(46)	$XOR(X,XOR(X,Y)) = Y$
(14)⊗	$MOD(X,1) = 0$	(47)⊗	$OR(X,0) = X$
(15)⊗	$MOD(X,X) = 0$	(48)⊗	$OR(2^{256}-1,X) = 2^{256}-1$
(16)	$MOD(X,0) = 0$	(49)⊗	$OR(X,X) = X$
(17) ⊗ ⊙	$EXP(X,0) = 1$	(50)	$OR(X,AND(X,Y)) = X$
(18) ⊗ ⊙	$EXP(X,1) = X$	(51)	$OR(OR(X,Y),Y) = OR(X,Y)$
(19)⊙	$EXP(1,X) = 1$	(52)⊗	$OR(OR(Y,X),Y) = OR(Y,X)$
(20)⊙	$EXP(0,X) = ISZERO(X)$	(53)	$OR(X,NOT(X)) = 2^{256}-1$
(21)⊙	$EXP(2,X) = SHL(X,1)$	(54)⊗	$AND(X,0) = 0$
(22)⊗	$GT(0,X) = 0$	(55)⊗	$AND(X,X) = X$
(23)⊗	$GT(1,X) = ISZERO(X)$	(56)⊗	$AND(2^{256}-1,X) = X$
(24)	$GT(X,X) = 0$	(57)⊗	$AND(AND(X,Y),Y) = AND(X,Y)$
(25)⊗	$LT(X,0) = 0$	(58)	$AND(AND(Y,X),Y) = AND(Y,X)$
(26)⊗	$LT(X,1) = ISZERO(X)$	(59)	$AND(X,OR(X,Y)) = X$
(27)	$LT(X,X) = 0$	(60)	$AND(X,NOT(X)) = 0$
(28)⊗	$EQ(X,X) = 1$	(61)⊙	$AND(ORIGIN,2^{160}-1) = ORIGIN$
(29)⊗	$EQ(X,0) = ISZERO(X)$	(62)⊙	$AND(CALLER,2^{160}-1) = CALLER$
(30)	$EQ(1,ISZERO(X)) = ISZERO(X)$	(63)⊙	$AND(ADDRESS,2^{160}-1) = ADDRESS$
(31)	$ISZERO(SUB(X,Y)) = EQ(X,Y)$	(64)⊙	$AND(COINBASE,2^{160}-1) = COINBASE$
(32)⊗	$ISZERO(GT(X,0)) = ISZERO(X)$	(65)⊙	$BALANCE(ADDRESS) = SELFBALANCE$
(33)⊗	$ISZERO(LT(0,X)) = ISZERO(X)$		

Figure A.1: Simplification rules of gasol

Appendix B

Complete SMT Encoding

Table B.1 shows our complete SMT encoding. The first three constrains S_V, B and E, are used to model the stack and its elements. The stack only contains initial stack variables $s_0, \ldots, s_{k-1}$, integer values in the domain $\{0, \ldots, 2^{256} - 1\}$ and fresh variables $s_k, \ldots, s_v$. The constraint S_V is applied in order to distinguish between constants and the variables s_i. Let b_s be the size of the stack and b_o the number of operations. Then, we have to model a stack of b_s elements and their values after executing j operations where $j \in \{0, \ldots, b_o\}$. The quantified variables $x_{i,j} \in \mathbb{Z}$ with $i \in \{0, \ldots, b_s - 1\}$ and $j \in \{0, \ldots, b_o\}$ express the word at position i of the stack after executing the first j operations of the code, where $x_{0,j}$ encodes the word on the top of the stack. In addition, the propositional variables $u_{i,j}$ with $i \in \{0, \ldots, b_s - 1\}$ represent that the elements of the stack are active after executing the first j EVM instructions. The constraint B is used to describe the initial stack and constraint E to describe the final stack.

To simplify the definitions of the constraints, we define the auxiliary predicate:

$$Move(j, \alpha, \beta, \delta) = \bigwedge_{\alpha \leqslant i \leqslant \beta} \; u_{i+\delta,j+1} = u_{i,j} \;\wedge\; x_{i+\delta,j+1} = x_{i,j}$$

Given an instruction step j with $0 < j \leq b_o$, two stack positions α and β and a shift amount $\delta \in \mathbb{Z}$, with $0 \leq \alpha$, $0 \leq \alpha + \delta$, $\beta < b_s$ and $\beta + \delta < b_s$, the constraint defines that the stack after executing $j + 1$ instructions between positions α and β is the same as the stack after executing the j instruction but with a shift of δ.

In the encoding of the instructions in Table B.1 we use existentially quantified variables t_j, with $j \in \{0, \ldots, b_o - 1\}$ that encode the selected instruction at step j. If $t_j = \theta(\iota)$ then we have that the operation executed at step j is ι. The constraints presented in the Section *Instruction model constraints* encode the selection of all possible instructions. The existentially quantified variables $a_j \in \{0, \ldots, 2^{256} - 1\}$, with $j \in \{0, \ldots, b_o - 1\}$, express the value pushed at the top of the stack when $t_j = \theta(\texttt{PUSH})$. The instruction `NOP` represents the no-operation, maintaining the state of the stack as it was without making any modifications. To avoid redundant results, i.e., those that have `NOP` in intermediate positions we add the constraint $C_{\texttt{fromNOP}}$ to express that once a `NOP` is chosen, the rest of the instructions have to be `NOP`.

Constraints C_U and C_C encode the (non-commutative and commutative respectively) uninterpreted functions that are abstracted from the symbolic expressions of the SFS where f is a uninterpreted function, and $o_0, \ldots, o_{n-1}$ the arguments that it takes. The complete encoding of the selection of an instruction at each step j is generated with C_I.

C_{SFS} expresses the complete encoding by combining all the constraints that involved the modeling of the stack and the instructions.

Stack model constraints

$$
\begin{array}{rcl}
S_V & = & \bigwedge_{0 \leqslant i < v} s_i = 2^{256} + i \\
B & = & \bigwedge_{0 \leqslant \alpha < k} (u_{\alpha,0} \ \wedge \ x_{\alpha,0} = s_\alpha) \ \wedge \ \bigwedge_{k \leqslant \beta \leqslant b_s - 1} \neg u_{\beta,0} \\
E & = & \bigwedge_{0 \leqslant \alpha < w} (u_{\alpha,b_o} \ \wedge \ x_{\alpha,b_o} = f_\alpha) \ \wedge \ \bigwedge_{w \leqslant \beta \leqslant b_s - 1} \neg u_{\beta,b_o}
\end{array}
$$

Instruction model constraints

$$
\begin{array}{rcll}
C_{\texttt{PUSH}}(j) & = & t_j = \theta(\texttt{PUSH}) & \Rightarrow \ 0 \leq a_j < 2^{256} \wedge \neg u_{b_s-1,j} \wedge u_{0,j+1} \wedge x_{0,j+1} = a_j \wedge \\
 & & & Move(j, 0, b_s - 2, 1) \\
C_{\texttt{DUP}k}(j) & = & t_j = \theta(\texttt{DUP}k) & \Rightarrow \ \neg u_{b_s-1,j} \wedge u_{k-1,j} \wedge u_{0,j+1} \wedge x_{0,j+1} = x_{k-1,j} \wedge \\
 & & & Move(j, 0, b_s - 2, 1) \\
C_{\texttt{SWAP}k}(j) & = & t_j = \theta(\texttt{SWAP}k) & \Rightarrow \ u_{k,j} \wedge u_{0,j+1} \wedge x_{0,j+1} = x_{k,j} \wedge u_{k,j+1} \wedge \\
 & & & x_{k,j+1} = x_{0,j} \wedge \ Move(j, 1, k - 1, 0) \wedge \\
 & & & Move(j, k + 1, b_s - 1, 0) \\
C_{\texttt{POP}}(j) & = & t_j = \theta(\texttt{POP}) & \Rightarrow \ u_{0,j} \wedge \neg u_{b_s-1,j+1} \wedge Move(j, 1, b_s - 1, -1) \\
C_{\texttt{NOP}}(j) & = & t_j = \theta(\texttt{NOP}) & \Rightarrow \ Move(j, 0, b_s - 1, 0) \\
C_U(j, f) & = & t_j = \theta(f) & \Rightarrow \ \bigwedge_{0 \leqslant i \leqslant n-1} (u_{i,j} \ \wedge \ x_{i,j} = o_i) \ \wedge u_{0,j+1} \ \wedge \ x_{0,j+1} = r \ \wedge \\
 & & & Move(j, n, min(b_s - 2 + n, b_s - 1), 1 - n) \ \wedge \\
 & & & \bigwedge_{b_s - n + 1 \leqslant i \leqslant b_s - 1} \neg u_{i,j+1} \\
C_C(j, f) & = & t_j = \theta(f) & \Rightarrow \ u_{0,j} \wedge u_{1,j} \ \wedge \\
 & & & ((x_{0,j} = o_0 \wedge x_{1,j} = o_1) \vee (x_{0,j} = o_1 \wedge x_{1,j} = o_0)) \wedge \\
 & & & u_{0,j+1} \wedge x_{0,j+1} = r \wedge Move(j, 2, b_s - 1, -1) \wedge \neg u_{b_s-1,j+1}
\end{array}
$$

$$
C_{\texttt{fromNOP}} \ = \ \bigwedge_{0 \leqslant j < b_o - 1} \ t_j = \theta(\texttt{NOP}) \Rightarrow t_{j+1} = \theta(\texttt{NOP})
$$

$$
\begin{array}{rcl}
C_{\mathcal{I}} & = & C_{\texttt{fromNOP}} \ \wedge \ \bigwedge_{0 \leqslant j < b_o} 0 \leq t_j \leq m_\iota \ \wedge \\
 & & C_{\texttt{PUSH}}(j) \ \wedge \ C_{\texttt{DUP}k}(j) \ \wedge \ C_{\texttt{SWAP}k}(j) \ \wedge \ C_{\texttt{POP}}(j) \ \wedge \\
 & & C_{\texttt{NOP}}(j) \ \wedge \ \bigwedge_{f \in \mathcal{I}_U} C_U(j, f) \ \wedge \ \bigwedge_{f \in \mathcal{I}_C} C_C(j, f))
\end{array}
$$

Complete SMT encoding

$$
C_{SFS} \ = \ S_V \ \wedge \ C_{\mathcal{I}} \ \wedge \ B \ \wedge \ E
$$

Gas consumption constraint

$$
O'_{SFS} \ = \ C_{SFS} \ \wedge \ \bigwedge_{0 \leqslant j < b_o} \ \bigwedge_{1 \leqslant i \leqslant p} \ [\ \bigvee_{\iota \in W_0 \uplus \ldots \uplus W_{i-1}} t_j = \theta(\iota) \ , \ w_i \ ]
$$

Additional Constraints

$$
\begin{array}{rcl}
C_L & = & \bigwedge_{\iota \in \mathcal{I}_U \uplus \mathcal{I}_C} \ \bigvee_{0 \leqslant j < b_o} \ t_j = \theta(\iota) \\
C_N & = & \bigwedge_{k \in P} \ \bigvee_{0 \leqslant j < b_o} (t_j = \theta(\texttt{PUSH}) \wedge a_j = k) \\
C_U & = & \bigwedge_{\iota \in \mathcal{I}_U \uplus \mathcal{I}_C} \ \bigwedge_{0 \leqslant j < b_o} (t_j = \theta(\iota) \Rightarrow \bigwedge_{0 \leqslant i < b_o, i \neq j} (t_i \neq \theta(\iota))) \\
C_R & = & \bigwedge_{0 \leqslant j < b_o - 1} (t_{j+1} = \theta(\texttt{POP}) \Rightarrow \bigvee_{\iota \in M} (t_j = \theta(\iota)))
\end{array}
$$

Table B.1: gasol Max-SMT encoding.

In the Section *Gas consumption constraint* in Table B.1, O'_{SFS} encodes the optimization problem as a Max-SMT problem. The EVM instructions are split in $p+1$ disjoint sets $W_0 \uplus \ldots \uplus W_p$ depending on the gas that they consume, i.e., all the instructions that consume the same amount of gas $cost_i$ belong to the same set W_i. In addition, the costs are strictly increasing: $cost_0 = 0$ and $\mathsf{cost}_{i-1} < \mathsf{cost}_i \ \forall i \in \{1 \ldots p\}$. An alternative encoding O'_{SFS} of the gas consumption is shown bellow. This proposal considers all the instructions and its associated gas cost v_i.

$$O_{SFS} = C_{SFS} \ \wedge \ \bigwedge_{0 \leqslant j < b_o} \ \bigwedge_{\iota \in \mathcal{I}} \ [t_j \neq \theta(\iota) \ , \ v_j \ ]$$

Finally, the last four constraints shown in table B.1 describe the additional constraints presented in Section 3.4.1.

Appendix C

Proofs of the encodings

In this Appendix, we present a proof for each of the additional encodings provided in Section 3.4.1: C_L, C_N, C_U and C_R. Depending on the group in which they were classified, the proofs will follow these ideas:

1. Redundant constraints: we will prove that any model of the Max-SMT problem must satisfy these constraints. These proofs are based on properties that the constraints in C_{SFS} satisfy. Proofs in Sections C.1 and C.2 follow this idea.

2. Constraints satisfied by at least an optimal solution: given an arbitrary solution, we will build another one that is equivalent and preserves optimality. Instead of dealing with the constraints directly, we will work with the generated *sub-blocks* of instructions. Proof in Section C.3 follow this idea.

3. *Heuristic* constraints: we will present a *counterexample* on why C_U does not always hold in Section C.4. This same *counterexample* hints why using this heuristic does not usually affect optimality. Finally, empirical results will be presented in order to corroborate this idea.

Before proving the lemmas, we will introduce two auxiliary lemmas that explicitly state how *meaningful* stack variables and numerical values are introduced in the model. A *meaningful* stack variable or numerical value corresponds to the assignment of its corresponding value in the encoding to a variable $x_{i,j}$ that also verifies $u_{i,j} = \top$. These values are introduced into the model through a $C_{\texttt{PUSH}}$ for numerical values, and through C_U, C_C for stack variables.

Lemma C.1. *Let $ASFS := \langle \mathcal{S}, \mathcal{S}_0, M \rangle$ be the Abstract SFS derived from a block and C_{SFS} the hard constraints associated for the Max-SMT problem. Let v be a numerical value and i, j two indices that verify the following properties:*

1. $i \in \{0, \ldots, b_s - 1\}$, $j \in \{0, \ldots, b_0 - 1\}$
2. $x_{i,j+1} = v$
3. $u_{i,j+1} = \top$
4. $\forall k, l.\ [k \in \{0, \ldots, b_s - 1\} \land l \in \{0, \ldots, j\} \land u_{k,l} = \top \implies x_{k,l} \neq v]$

Then, $t_j = \theta(\texttt{PUSH})$ and $a_j = v$.

Proof. The result is followed from the way C_{SFS} constraints are designed. Let us consider the instruction model constraints from Table B.1.

Firstly, the assignment of $x_{i,j+1}$ cannot be followed from a *Move* constraint. If so, there would exist a value δ such that $u_{i,j+1} = u_{i-\delta,j} = \top$ and $x_{i,j+1} = x_{i-\delta,j}$. A contradiction is followed from this fact and (4).

Now we can focus on the remaining clauses in those predicates. The only clause that introduces both that $u_{i,j} = \top$ and $x_{i,j} = v$ is $C_{\texttt{PUSH}}(j)$ with $a_j = v$. No other value can be assigned to t_j: by checking every possible instruction model constraint, it cannot be followed that $x_{i,j} = v \wedge u_{i,j} = \top$. The result is derived.

□

Lemma C.2. *Let $ASFS := \langle \mathcal{S}_0, \mathcal{S}, M \rangle$ be the Abstract SFS derived from a block and C_{SFS} the hard constraints associated to the Max-SMT problem. Let s_m be a stack variable and i, j two indices that verify the following properties:*

1. $i \in \{0, \ldots, b_s - 1\}$, $j \in \{0, \ldots, b_0 - 1\}$
2. $x_{i,j+1} = s_m$
3. $u_{i,j+1} = \top$
4. $\forall k, l.\ [k \in \{0, \ldots, b_s - 1\} \wedge l \in \{0, \ldots, j\} \wedge u_{k,l} = \top \implies x_{k,l} \neq s_m]$

Then, $s_m \in dom(M)$ and $t_j = \theta(\texttt{Op})$, with $M(s_m) = \texttt{Op}(\ldots)$

Proof. This proof follows the same idea as Lemma C.1. Firstly, we will reason that $s_m \in dom(M)$. But due to Property 3.1, this assertion is equivalent to prove that $s_m \notin \mathcal{S}_0$. If $s_m \in \mathcal{S}_0$, that would mean $x_{i,0} = s_m \wedge u_{i,0} = \top$ and then (4) is contradicted regardless of the chosen j.

The only clause that can introduce both $u_{i,j+1} = \top$ and $x_{i,j+1} = s_m$ is of the form $x_{i,j+1} = r$. The corresponding constraint can only be either $C_C(j)$ or $C_U(j)$. As $r = s_m$, necessarily $t_j = \theta(\texttt{Op})$, with $M(s_m) = \texttt{Op}(\ldots)$. Again, from the remaining constraints, it cannot be followed that $x_{i,j+1} = s_m \wedge u_{i,j+1} = \top$.

□

C.1 Proof of C_L encoding

Before proceeding with the C_L, we will introduce a first lemma based on the structure of the *Abstract SFS*. As it was discussed in Section 3.2.1, its associated *operator* map is minimal, meaning in particular no noise can be introduced.

Lemma C.3. *Let $ASFS := \langle \mathcal{S}_0 := [s_0, \ldots, s_{k-1}], \mathcal{S} := [f_0, \ldots, f_{w-1}], M \rangle$ be the* Abstract SFS *derived from a block and C_{SFS} the hard constraints associated to the Max-SMT problem. Let $s_k \in dom(M)$. Then, every model that satisfies C_{SFS} contains a variable $x_{i,j}$ s.t $i \in \{0, \ldots, b_s - 1\}$, $j \in \{1, \ldots, b_0\}$, $u_{i,j} = \top$ and $x_{i,j} = s_k$.*

Proof. Let us consider a model T that satisfies C_{SFS}. We shall proceed by induction on the shape of *minimal ASFS* depicted in Property 3.4:

Case base: let us assume $s_k \in \mathcal{S}$. Then, there exists an index $0 \leq i \leq w - 1$ s.t. $f_k = s_k$. E constraint must be satisfied by model T, so in particular, it is followed that $x_{k,b_0} = f_k = s_k$. Also, as $k < w$, it also holds that $u_{k,w} = \top$.

Induction Hypothesis: Let us assume the property holds for a stack variable s_t s.t. $s_t \neq s_k$ and $M(s_t) = \texttt{Op}(v_0, \ldots, s_k, \ldots, v_k)$.

Inductive Case: Applying the **Induction Hypothesis**, we know there exists at least one variable $x_{i,j}$ s.t. $x_{i,j} = s_t$. It may happen that several $x_{i,j}$ satisfy this property, so let us consider one specific $x_{i,j}$ s.t $u_{i,j} = \top$ and minimizes j; *i.e.* either $j = 0$ or $j > 0 \wedge \forall l, m.\, [0 \leq l \leq b_s - 1 \wedge 0 \leq m < j \wedge u_{l,m} \implies x_{l,m} \neq s_t]$. In fact, $j > 0$ must hold, as $j = 0$ would imply that $s_t \in \{s_0, \ldots, s_{k-1}\}$ and we know that $dom(M) \cap \{s_0, \ldots, s_k\} = \varnothing$, according to Property 3.1.

From Lemma C.2, it is followed that necessarily $t_{j-1} = \theta(\texttt{Op})$. `Op` can be either non-commutative or commutative.

In the former case, let h be the position of s_k in the sequence of parameters of operation `Op`. In order to satisfy constraint $C_U(j-1, \texttt{Op})$, it is followed that $u_{h,j} = \top$ and $x_{h,j} = s_k$. The result is followed directly.

In the later case, h can be either 0 or 1. Let us assume $h = 0$ (case $h = 1$ is analogous). In order to satisfy constraint $C_C(j-1, \texttt{Op})$, either $x_{0,j} = s_k \wedge x_{1,j} = o_1$ or $x_{0,j} = o_1 \wedge x_{1,j} = s_k$. Besides, $u_{0,j} = \top$ an $u_{1,j} = \top$, so the result is followed in both cases. □

This proof may seem complex at first glance, but the idea behind it is quite reasonable: the first time stack variable s_k appears in the model corresponds to the first application of the corresponding *uninterpreted function*. It is important to emphasize four crucial facts from this proof:

1. In general, the introduction of a variable in a model $x_{i,j} = s_k$ does not mean that $t_{j-1} = \theta(\texttt{Op})$. Once it has appeared previously in the model, it can be obtained through the constraint associated to `DUPk`. That is why we need to reason on the first time a stack variable appears in the model.

2. The lemma above only can be followed if the corresponding $u_{i,j} = \top$. Otherwise, we could have taken any model in which there exists a previously $u_{i,j} = \bot$ and assign the corresponding $x_{i,j} = s_k$. This assignment would not affect the satisfiability of the model. However, nor does it add any information to prove C_L encoding is correct.

3. By limiting to the first application of the stack variable in the model, we have also deduced that $t_{j-1} = \theta(\texttt{Op})$. The same reasoning can be applied to the **case base** and the result still holds. This is the foundation of why C_L encoding is correct.

4. Defining the set of pushed values in an *analogous* constructive way as Property 3.4 will allow us to reuse the same reasoning as above to derive C_N encoding.

Point 3 is explicitly stated in the following lemma:

Lemma C.4. *Let B be a basic block, $ASFS := \langle \mathcal{S}_0, \mathcal{S}, M \rangle$ an associated Abstract SFS and T a model that satisfies the hard constraints of the correspondent Max-SMT problem. Then, for every $\iota \in \mathcal{I}$ there exists an index $i \in \{0, \ldots, b_0 - 1\}$ s.t. $t_i = \theta(\iota)$ in T.*

Proof. For every $\iota \in \mathcal{I}$, there exists only one stack variable s_k s.t. $M(s_k) = \iota$. Applying Lemma C.3, the result holds directly if $s_k \notin \mathcal{S}$. But if $s_k \in \mathcal{S}$, we can also identify the minimum index j s.t. there exists a i that verifies $x_{i,j} = s_k$ and $u_{i,j} = \top$. And applying the same reasoning as in the proof of the Lemma, necessarily $t_{j-1} = \theta(\iota)$. □

Thus, the correction of C_L is directly followed as a corollary:

Theorem C.1. *Let B be a basic block. C_L holds for any model derived from the corresponding C_{SFS}.*

Proof. Let us take a model T that satisfies C_{SFS}. For every $\iota \in \mathcal{I}_C \uplus \mathcal{I}_C$, we can apply Lemma C.4, so it is followed that there exists an index $j \in \{0, \ldots, b_0 - 1\}$ s.t. $t_j = \theta(\iota)$. Hence, C_L holds.

□

C.2 Proof of C_N encoding

Recall the definition of P in Subsection 3.4.1. No formal definition was provided at that point, as it is really easy to identify the numerical values associated to the definition of an *ASFS*. Nevertheless, in order to prove C_N encoding, we need to formalize this notion:

Definition C.1. *Let $ASFS := \langle \mathcal{S}_0, \mathcal{S}, M \rangle$ be a ASFS associated to a sub-block B. Then, we define P_{ASFS} as the set of numerical values that verifies either one of the following conditions:*

1. $v \in \mathcal{S}$
2. *There exists a functor* `Op` *and stack variable s_i s.t. $s_i \in dom(M)$ and $M(s_i) =$* `Op`$(d_0, \ldots, v, \ldots, d_k)$.

The proof is splitted in two different lemmas, following the same idea as Lemmas C.3 and C.4:

Lemma C.5. *Let $ASFS := \langle [s_0, \ldots, s_{k-1}], [f_0, \ldots, f_{w-1}], M \rangle$ be the Abstract SFS derived from a block and C_{SFS} the hard constraints associated to the Max-SMT problem. Let P be the set of numerical values depicted in Definition C.1 and $v \in P$. Then, every model that satisfies T contains a variable $x_{i,j}$ s.t $i \in \{0, \ldots, b_s - 1\}$, $j \in \{1, \ldots, b_0\}$, $u_{i,j} = \top$ and $x_{i,j} = v$.*

Proof. Let us take a generic element $v \in P$. According to Definition C.1, at least one of the following conditions must hold:

(1) holds: It is followed that $v \in \mathcal{S}$. Then, there exists an index $0 \leq i \leq w - 1$ s.t. $f_k = v$. E constraint must be satisfied by model T, so in particular, it is followed that $x_{k,b_0} = f_k = v$. Also, as $k < w$, it also holds that $u_{k,w} = \top$.

(2) holds: According to Lemma C.4, there exists at least an index $j \in \{0, \ldots, b_0 - 1\}$ s.t. $t_j = \theta(\texttt{Op})$. Hence, the right clause of the corresponding $C_U(j, f)$ or $C_C(j, f)$ constraint must hold. In particular, there exists an index i, $i \in \{0, \ldots, b_s - 1\}$ s.t. $u_{i,j+1} = \top$ and $x_{i,j+1} = v$.

□

Once it has been proven that every v is a *meaningful* numerical value, we can reason it must have appeared in the model due to a `PUSH` operation:

Lemma C.6. *Let $ASFS := \langle \mathcal{S}_0, \mathcal{S}, M \rangle$ be the* Abstract SFS *derived from a block and C_{SFS} the hard constraints associated for this problem. Let P be the set of numerical values depicted in Definition C.1 and $v \in P$. Then, every model that satisfies T contains an index j, $j \in \{0, \ldots, b_0 - 1\}$ s.t. $t_j = \theta($`PUSH`$)$ and $a_j = v$.*

Proof. In Lemma C.5, it was proven there exists indices i, j s.t. $i \in \{0, \dots, b_s - 1\}$, $j \in \{1, \dots, b_0\}$, $u_{i,j} = \top$ and $x_{i,j} = v$. Index j is lower bounded by 0, and thus, we can choose the minimum j s.t. $x_{l,k} \neq v$ for any $l \in \{0, \dots, b_s - 1\}, 0 \leq k < j$. Also, note that $j > 0$, as $x_{l,0}$ is always assigned to a stack variable. From this point, let us fix i, j s.t the property holds and j is minimal. Lemma C.1 can be applied and the result is directly followed. □

The correctness of the encoding is directly followed from this lemma:

Theorem C.2. *Let B be a basic block. C_N holds for any model derived from C_{SFS}.*

Proof. Let us take a model T that satisfies C_{SFS} and let P be the set of input values. For every $v \in P$, we can apply Lemma C.6, so it is followed that there exists an index $j \in \{0, \dots, b_0 - 1\}$ s.t. $t_j = \theta(\texttt{PUSH}) \wedge a_j = v$. Hence, C_N holds.

□

C.3 Proof of C_R encoding

We first introduce the following auxiliary lemma:

Lemma C.7. *Let $\mathcal{S}_0$ be an arbitrary stack of length m and $b^{\delta,\alpha}$ be any EVM instruction with $\alpha = 1$ and $\delta \leq m$. Then the output stack after applying $b^{\delta,\alpha}$ followed by `POP` to $\mathcal{S}_0$ is the same as the output stack after applying `POP` δ times. Besides, if the gas g associated with $b^{\delta,\alpha}$ verifies that*

$$2 * \delta < g + 2 \tag{C.1}$$

then the gas consumed by the new sequence is strictly smaller than the starting one.

Proof. Let $\mathcal{S}_0 = [s_0, \dots, s_{m-1}]$ be the initial stack and $\mathcal{S}_1 := [s_0, \dots, s_{m-1-\delta}, s_m]$ the the output stack after applying $b^{\delta,\alpha}$, where s_m is the new stack variable introduced by b. The output stack after considering `POP` is denoted as $\mathcal{S}_2 := [s_0, \dots, s_{m-1-\delta}]$.

On the other hand, applying δ times `POP` to $\mathcal{S}_0$, leads to the output stack $\mathcal{S}'_1 := [s_0, \dots, s_{m-1-\delta}]$. Then, it holds that $\mathcal{S}_2 = \mathcal{S}'_1$.

Note that the gas associated to `POP` opcode is 2. Then, the first sequence consumes $g + 2$ units of gas. The second sequence consumes $2 * \delta$ units of gas. Thus, if Equation C.1 holds, it is clear that the transformation does not consume more gas.

□

In fact, all opcodes with $\alpha = 1$ in the gas model presented in [43] satisfy Equation C.1. This means that the transformation can be applied to any sequence without losing optimality. This helps us prove Theorem C.3:

Theorem C.3. *Let $B := b_i \dots b_j$ be a basic block s.t. there exists an index k, with $i \leq k < j$ that verifies $b_k = b^{\delta,\alpha}$ with $\alpha = 1$ and $b_{k+1} =$ `POP`. Then, there exists another block $B' := b'_i \dots b'_l$ s.t. SFS associated to B and B' are equivalent according to Definition 4.1. B' satisfies that there is no index k', with $i \leq k' < j$ that verifies $b_{k'} = b^{\delta,\alpha}$, $\alpha = 1$ and $b_{k'+1} =$ `POP`. Besides, its gas consumption is strictly smaller than the one in B.*

Proof. In order to obtain B', for each index k that verifies $b_k = b^{\delta,\alpha}$ with $\alpha = 1$ and $b_{k+1} = \texttt{POP}$, we replace both EVM instructions by a sequence of δ `POP` opcodes. Lemma C.7 states that both sequences lead to the same output stack, and thus, the transformation preserves the SFS at each replacement. Besides, the gas associated with this transformation is less than the starting one, so the total gas of B' cannot be greater or equal than the gas consumed by B. Then, B' satisfies all properties listed in the lemma. □

Note that B' could be larger in terms of number of opcodes than B, if $\delta > 2$. This could lead C_R constraints to be too restrictive and prevent us from finding an optimal solution, as the length of any solution found by the tool is upper bounded by b_0. Nevertheless, we can prevent this situation by considering artificial opcodes `POP`x, that pops directly the top x elements from the stack using just one instruction. In practice, we could not find any examples of a sequence containing $b^{\delta,\alpha}$ with $\delta > 2$, $\alpha = 1$ followed by `POP` in the blocks in our benchmarks, so we have not included this special case in our encoding.

C.4 Counter-example of C_U encoding

Let us consider the $ASFS := \langle \mathcal{S}_0, \mathcal{S}, M \rangle$, in which $\mathcal{S}_0 := [\,]$, $\mathcal{S} := [s_1, 0, \ldots, 0, s_1]$ is a final stack of 34 elements; and $M := \{s_0 \mapsto \texttt{Op}(64), s_1 \mapsto \texttt{Op'}(s_0)\}$. Both `Op` and `Op'` spend 4 units of gas, so C_U encoding can be applied to them. By considering the *Initial encoding*, the optimal gas has been proven to be 112 using OptiMathSat. However, by considering $O'_{SFS} + C_L + C_U$, the optimal gas has been proven to be 113 using the same solver. Thus, adding C_U encoding leads to a loss of optimality.

Why has optimality been lost? The final $\mathcal{S}$ has been purposefully designed to be really big and contain the same fresh stack variable as top of the stack and bottom of the stack. This way, it is not worthy to perform once `Op` and `Op'` and then use `DUPk` and `SWAPk` operations to move the value to the top of the stack. Instead, it is cheaper to introduce value 64 and perform each operation twice.

In general, using Max-SMT solvers lead to solutions in which the application of `SWAPk` instructions is done in a really effective and minimal way. It is usually not worthy performing the same operations twice, but instead duplicating the result of the operation. Despite our example having a final stack size of 34 elements, only one unit of gas is lost due to this behaviour.

In fact, when performing the first experimental phase described in Section 6.2, we have measured the number of examples in which optimality is lost: all *sub-blocks* in which an optimal equivalent is obtained by both encodings (roughly 90 % of all analyzed *sub-blocks*) have been compared according to their associated gas cost. No loss of optimality has been detected this way. This behaviour is due to the fact that *sub-blocks*'s stack is usually small. Therefore, we can conclude that C_U serves as a truly reliable heuristic.