GASOL: GAS Optimization tooLkit

GASOL 0.1[1]

Elvira Albert Pablo Gordillo Alejandro Hernández
Albert Rubio

December 21, 2021

[1]This work was funded partially by the Ethereum Foundation under Grant ID FY21-0372

Abstract

This document describes the main results obtained after Stage 2 of the GASOL project. It improves the previous version of GASOL by considering memory and storage instructions. The next items summarize the main achievements:

1. We have developed a new version of the super-optimization tool that optimizes storage and memory operations;
2. We have extended the SMT model defining a pre-order between the uninterpreted functions that appear in the specification of the stack;
3. We have generalized some of its components to enable byte-size optimization criteria;
4. We have tested the tool on the same dataset used in the previous version. We have downloaded the last 30 verifed smart contracts from Etherscan that have been compiled using the version 8 of solc and whose source code was available on June 21, 2021, we have compiled them using the version 0.8.9 of solc and compared experimentally the gains against the previous versions of GASOL;
5. We have made the tool available in a github repository and added instructions for its usage.

Our experiments on 12,378 blocks from the analyzed real contracts achieve gains of 16.42% in gas *wrt.* the previous version of the optimizer without memory handling, and gains of 3.28% in bytes-size over code already optimized by solc.

Chapter 1

The New Architecture of GASOL 0.1

Figure 1.1 displays the architecture of GASOL 0.1, white components are borrowed from other tools, while gray components correspond to the new developments of this phase of the project (either completely new, like **DEP**, or novel extensions for memory handling of previous implementations, like **SPEC**, **SIMP** and **SMT**). The **input** to GASOL 0.1 is a smart contract (either its source in Solidity or its compiled EVM bytecode [11]), a selection of the optimization criteria (currently we are supporting gas consumption and size in bytes), and system settings (this includes compiler options for invoking the solc compiler and GASOL 0.1 settings like the timeout per block of instructions). The **output** of GASOL 0.1 is an optimized bytecode program and optionally a report with detailed information on the optimizations achieved (e.g., number of blocks optimized, number of blocks proven optimal, gas/size reduction gains, optimization time, among others).

The first component, labeled **SOLC** in the figure, invokes the Solidity compiler solc to obtain the bytecode in their assembly json exchange format [1]. Working on this exchange format has many advantages, one is that we can enable the optimizer of solc [3] and start the superoptimization from an already optimized bytecode. Besides, the format has been designed to be a usable common denominator for EVM 1.0, EVM 1.5 and Ewasm. Hence, we argue it is a good source for superoptimization as different target platforms will be able to use our tool equally. The assembly json format provides the EVM bytecode of the smart contract, metadata that relates it with the source Solidity code, and compilation information such as the version used to generate the bytecode. The output yield by GASOL 0.1 can also be returned in assembly json format so that it can be used by other tools working on this format in the future. From the assembly json, the next component **BLK** partitions the bytecode given by solc into a set of sequences of loop-free bytecode instructions, named *blocks*, which correspond to the blocks of the CFG and also computes the size of the stack when entering each block.[1] We omit details of this step as it is standard in compiler construction and, for the case of the EVM, has been already subject of other analysis and optimization papers (see, e.g. [6, 9, 8] and their references).

[1] In EVM, it is possible to reach a block with different stack sizes, and all such sizes can be statically computed. We will refer to the minimum or maximum when needed.

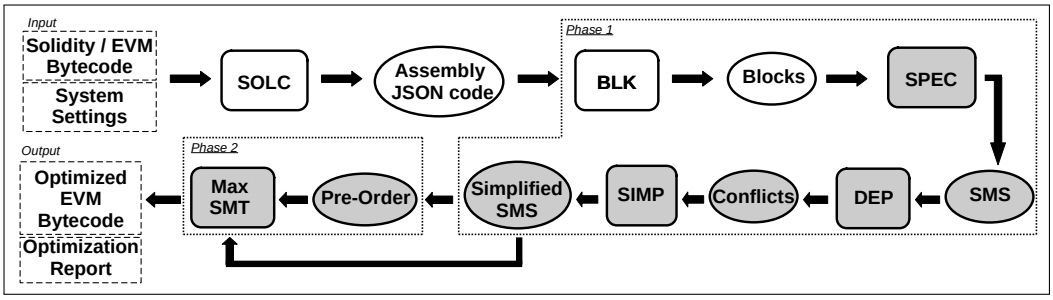

Figure 1.1: Architecture of GASOL 0.1

The next component **SPEC** synthesizes a functional specification of the operand stack and of the memory and storage (SMS for short) for each block of bytecode instructions. This is done by symbolically executing the bytecodes in the block to extract from them what the contents of the operand stack and of the memory/storage are after executing them. The description of this component is given in Section 2.1. Next, **DEP** establishes the dependencies among the memory accesses from which a pre-order, that determines when a memory access needs to be performed before another one, is generated. For instance, subsequent load accesses, which are not interleaved by any store, do not have dependencies among them, while they do have with subsequent write accesses to the same positions. This phase is described in Sections. 2.2 (dependencies) and 2.3 (pre-order). In the next component **SIMP**, we apply simplification rules on the SMS. We include all stack simplification rules of GASOL 0.1, as well as the additional rules we have developed for memory/storage simplifications. For instance, successive write accesses that overwrite the same memory position are simplified to a single one provided the same memory location is not read by any other instruction between them. The description of this component is given in Section 2.2. Finally, we generate a **Max-SMT** encoding from the (simplified) SMS that incorporates the pre-order established by the component **DEP** and from which the optimized bytecode is obtained. The description of this component is given in Chapter 3.

Chapter 2

Synthesis of Stack and Memory Specifications

This chapter describes the first stage of the optimization (components **SPEC**, **SIMP** and **DEP**) that consists in synthesizing from a loop-free sequence of bytecode instructions a *simplified* specification of the stack and of the memory/storage (with the dependencies) that the execution of such bytecodes produces.

2.1 Initial Stack and Memory/Storage Specification

For each block, we synthesize its Stack and Memory Specification (SMS) by symbolically executing the instructions in the sequence. Function τ in Figure 2.1 defines the symbolic execution for the memory/storage operations (1-4) and includes two representative stack opcodes (5-6). The first parameter of τ is a bytecode instruction and the second one is the SMS data structure $< \mathcal{S}, \mathcal{M}, \mathcal{S}t >$ whose first element corresponds to the stack ($\mathcal{S}$), the second one to the memory ($\mathcal{M}$), and the third one to the storage ($\mathcal{S}t$). The stack $\mathcal{S}$ is a list whose position $\mathcal{S}[0]$ corresponds to the top of the stack. At the beginning of executing a block, the stack contains the minimum number of elements needed to execute the block represented by symbolic variables s_i, where s_i models the element at $\mathcal{S}[i]$. The resulting list $\mathcal{M}$ ($\mathcal{S}t$ resp.) will contain the sequence of memory (storage resp.) accesses executed by the block. By abuse of notation, we often treat lists as sequences. Both $\mathcal{M}$ and $\mathcal{S}t$ are empty before executing the block symbolically. As an example, the symbolic execution of `SSTORE` removes the two top-most elements from $\mathcal{S}$, and adds the symbolic expression $\texttt{SSTORE}(\mathcal{S}[0], \mathcal{S}[1])$ to the storage sequence. Similarly, `SLOAD` removes from the top of the stack the position to be read, puts on the top of the stack the symbolic expression $\texttt{SLOAD}(\mathcal{S}[0])$ that represents the value read from the storage position $\mathcal{S}[0]$, and adds the same expression to the storage sequence $\mathcal{S}t$. As a result of applying τ to a sequence of bytecodes, the SMS obtained provides a specification of the target stack after executing the sequence in terms of the elements located in the stack before executing the sequence and, the target memory/storage (given as a sequence of accesses) after executing the sequence in terms of the input stack elements too. This extends function τ in [5], which only returned the stack, with the two additional elements in the SMS data structure.

$$
\begin{array}{lrcl}
(1) & \tau(\texttt{MLOAD}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) & := & <[\texttt{MLOAD}(\mathcal{S}[0]) \mid \mathcal{S}[1:n]], [\mathcal{M} \mid \texttt{MLOAD}(\mathcal{S}[0])], \mathcal{S}t> \\
(2) & \tau(\texttt{MSTORE}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) & := & <\mathcal{S}[2:n], [\mathcal{M} \mid \texttt{MSTORE}(\mathcal{S}[0], \mathcal{S}[1])], \mathcal{S}t> \\
(3) & \tau(\texttt{SLOAD}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) & := & <[\texttt{SLOAD}(\mathcal{S}[0]) \mid \mathcal{S}[1:n]], \mathcal{M}, [\mathcal{S}t \mid \texttt{SLOAD}(\mathcal{S}[0])]> \\
(4) & \tau(\texttt{SSTORE}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) & := & <\mathcal{S}[2:n], \mathcal{M}, [\mathcal{S}t \mid \texttt{SSTORE}(\mathcal{S}[0], \mathcal{S}[1])]> \\
(5) & \tau(\texttt{SWAPX}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) & := & \text{let temp} = \mathcal{S}[0] \ \ <\mathcal{S}[0/X][X/\text{temp}], \mathcal{M}, \mathcal{S}t> \\
(6) & \tau(\texttt{POP}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) & := & <\mathcal{S}[1:n], \mathcal{M}, \mathcal{S}t>
\end{array}
$$

Figure 2.1: SMS Synthesis by Symbolic execution

Example 1. *In order to illustrate our approach, we use the following bytecode that belongs to a real contract (bytecodes 0 to 47 of contract Welfare [2]). Its assembly json yield by the* ***SOLC*** *component contains 4524 bytecodes and after being partitioned by* ***BLK*** *we have 437 blocks to optimize. We illustrate the superoptimization of this block that contains in total 48 bytecodes from which 5 are the (underlined) memory/storage accesses:*

1	*PUSH1 80*	9	*DUP2*	17	*DUP4*	25	*PUSH2 3E8*	33	*PUSH2 FFFF*	41	*MUL*
2	*PUSH1 40*	10	*SLOAD*	18	*PUSH2 FFFF*	26	*PUSH1 1*	34	*MUL*	42	*OR*
3	*MSTORE*	11	*DUP2*	19	*AND*	27	*PUSH1 16*	35	*NOT*	43	*SWAP1*
4	*PUSH1 64*	12	*PUSH2 FFFF*	20	*MUL*	28	*PUSH2 100*	36	*AND*	44	*SSTORE*
5	*PUSH1 1*	13	*MUL*	21	*OR*	29	*EXP*	37	*SWAP1*	45	*POP*
6	*PUSH1 14*	14	*NOT*	22	*SWAP1*	30	*DUP2*	38	*DUP4*	46	*CALLVALUE*
7	*PUSH2 100*	15	*AND*	23	*SSTORE*	31	*SLOAD*	39	*PUSH2 FFFF*	47	*DUP1*
8	*EXP*	16	*SWAP1*	24	*POP*	32	*DUP2*	40	*AND*	48	*ISZERO*

As ***BLK*** *returns that the stack is empty when entering the block, we apply τ to the initial state $<[\,],[\,],[\,]>$ and produce the following SMS:*

$\mathcal{S} = [\texttt{ISZERO(CALLVALUE)}, \texttt{CALLVALUE}]$

$\mathcal{M} = [\texttt{MSTORE(64,128)}]$

$\mathcal{S}t = [\texttt{SLOAD}_1\texttt{(1)}, \texttt{SSTORE(1,V1)}, \texttt{SLOAD}_2\texttt{(1)}, \texttt{SSTORE(1,V2)}]$

where `V1` $=$ `OR(MUL(...)),AND(NOT(..))`, $\texttt{SLOAD}_1\texttt{(1)}$ *(omitting subexpressions) and* `V2` *another similar expression involving arithmetic, binary operations and* $\texttt{SLOAD}_2\texttt{(1)}$*. Note that we use subscripts to distinguish the* `SLOAD` *instructions by their position in $\mathcal{S}t$. The stack specification contains a term that represents the result of the opcode* `CALLVALUE` *(executed at line 46, L46 for short), and a term with the result of executing the opcode* `ISZERO` *on* `CALLVALUE`*, stored on top of the stack. The memory only contains one element that is obtained by symbolically executing the three first instructions. The* `PUSH` *instructions at L1 and L2 introduce the values* `40` *and* `80` *on the stack, and the* `MSTORE` *executed at L3 introduces in $\mathcal{M}$ the symbolic expression* `MSTORE(40,80)`*. Similarly, $\mathcal{S}t$ contains the sequence of symbolic expressions that represent the storage instructions executed in the block at L10, L23, L31 and L44 respectively. The expressions corresponding to* `V1` *and* `V2` *are also obtained by applying function τ to the corresponding state. These stack expressions can be simplified in the next step using the rules in [5].*

We note that the EVM memory is byte addressable (e.g., with instruction `MSTORE8`) and two different memory accesses may overlap. For simplicity of the presentation, we only consider the general case of word-addressable accesses, but the technique extends easily to the byte addressable case. In what follows, we use `LOAD` to abstract from the specific memory (`MLOAD`) and storage (`SLOAD`) bytecodes (and the same for `STORE`), when they are treated in the same way.

2.2 Memory/Storage Simplifications

In order to define the simplifications, and to later indicate to the SMT solver which memory instructions need to follow an order, we compute the conflicts between the different load and store instructions within each sequence.

Definition 1. *Two memory accesses A and B* conflict*, denoted as conf(A,B) if:*

(i) A is a store and B is a load and the positions they access might be the same;

(ii) A and B are both stores, the positions they modify might be the same, and they store different values.

Note that in (ii) two store instructions that might operate on the same position do not conflict if the values they store are equal, as we will reach the same memory state regardless of the order in which the stores are executed. Note that two load instructions are never in conflict as the memory state does not change if we execute them in one order or another.

Given the SMS obtained in Section 2.1, we achieve simplifications by applying the stack simplification rules of [5] and, besides, the following new memory simplification rules based on Definition 1 to the M and S components:

Definition 2 (memory simplifications)**.** *Let $< \mathcal{S}, \mathcal{M}, \mathcal{S}t >$ be an SMS, we can apply the following simplifications to any subsequence $b_1, \ldots, b_n$ in $\mathcal{M}$ or $\mathcal{S}t$:*

i) if $b_1 =$`STORE`(p, v) and $b_n =$`LOAD`(p) and $\nexists b_i =$`STORE` with $i \in \{2, \ldots, n-1\}$ and conf(b_1,b_i), we simplify it to $b_1, \ldots, b_{n-1}$ and replace b_n by v in the resulting SMS.

ii) if $b_1 =$`STORE`(p, v) and $b_n =$`STORE`(p, w) and $\nexists b_i =$`LOAD` with $i \in \{2, \ldots, n-1\}$ conf (b_1,b_i), we simplify it to $b_2, \ldots, b_n$.

iii) if $b_1 =$`LOAD`(p) and $b_n =$`STORE`$(p$,`LOAD`$(p))$ and $\nexists b_i =$`STORE` with $i \in \{2, \ldots, n-1\}$ conf(b_1,b_i), we simplify it to $b_1, \ldots, b_{n-1}$.

The simplifications can be applied in any order within $\mathcal{M}$ and $\mathcal{S}t$ until the process converges and the resulting sequence cannot be further simplified.

Intuitively, in (i), a load instruction from a position after a store instruction to the same position is simplified in the stack to the stored value provided there is no other store operation in between that might have changed the content of this position. In (ii), two subsequent store instructions to the same position are simplified to a single store if there is no load access on the same position between them. In (iii), a store instruction that stores in a position the result of the load in the same position can be removed, provided there is no other store in between that might have changed the content of this position.

Example 2. *In the SMS of Example 1, we have that conf(`SLOAD`$_1$`(1),SSTORE(1,V1)`), conf (`SLOAD`$_1$`(1),SSTORE(1,V2)`), conf(`SLOAD`$_2$`(1),SSTORE(1,V1)`), conf(`SLOAD`$_2$`(1),SSTORE(1,V2)`) and conf(`SSTORE(1,V1,SSTORE(1,V2)`) as all accesses operate on the same location. With these conflicts, we can apply rule i) to `SLOAD`$_2$`(1)`, as the previous `SSTORE` instruction has stored the value `V1` at the same location and there are no other storage instructions with conflict*

between them. Hence, we eliminate it from $\mathcal{S}t$ and replace it by `V1` *in the resulting SMS. After that, we are able to apply rule ii) on the two* `SSTORE` *instructions as they store a value at the same position without conflict loads in between. Then, we remove* `SSTORE(1,V1)` *from $\mathcal{S}t$. The resulting SMS has the same $\mathcal{S}$ and $\mathcal{M}$ and $\mathcal{S}t$ is now [*$\texttt{SLOAD}_1\texttt{(1)}, \texttt{SSTORE(1,V2')}$*] where* `V2'` *is* `V2` *replacing* $\texttt{SLOAD}_2\texttt{(1)}$ *by* `V1`*.*

2.3 Pre-Order for Memory and Uninterpreted Functions

Given the SMS and using the conflict definition above, we generate a pre-order, as defined below, that indicates to the SMT solver the order between the memory accesses that needs to be kept in order to obtain the same memory state as the original one. Clearly, having more accurate conflict tests will result in weaker pre-orders and hence a wider search space for the SMT solver. This in turn will result in potentially larger optimization. Our implementation is highly parametric on the conflict test **DEP** so that more accurate tests can be easily incorporated.

Definition 3. *Let A and B be two memory accesses in a sequence S. We say that B has to be executed after A in S, denoted as $A \sqsubset B$ if:*

i) (store-store) B is a store instruction and A is the closest store instruction predecessor of B in S such that `conf(A,B)`*.*

ii) (load-store) A is a load instruction and B is the closest store instruction successor of A in S such that `conf(B,A)`*.*

iii) (store-load) B is a load instruction and A is the closest store instruction predecessor of B in S such that `conf(A,B)`*.*

Let us observe that we do not compute the closure for the dependencies at this stage, as the SMT solver will infer them, as explained in Section 3.3.

Example 3. *From the simplified SMS of Example 2, we get the following load-store dependency,* $\texttt{SLOAD}_1\texttt{(1)} \sqsubset \texttt{SSTORE(1,V2')}$*, while the access* `MSTORE(64,128)` *has no dependencies as it is the unique memory operation.*

Importantly, the notion of pre-order between memory instructions can also be naturally extended to all other operations that occur in the specification of the target stack. These operations are handled as uninterpreted functions and have to be called in the right order to build the result that is required in the target stack. Therefore, we propose a novel implementation (both in SYRUP and GASOL) that extends the pre-order $\sqsubset$ to uninterpreted functions by adding $A \sqsubset B$ also when:

iv) (uninterpreted-functions) A and B are uninterpreted functions that occur in the target stack as $B(\ldots, A(\ldots), \ldots)$.

While in the case of uninterpreted functions the pre-order is used for improving performance, for memory operations the use of the pre-order is mandatory for soundness, since it is what ensures that the obtained block after optimization has the same final state (in the stack, memory and storage) than the original block.

2.4 Bounding the Operations Position

As we will show in the next section, a solution to our SMT encoding assigns a position in the final instruction list to each operation such that the target stack is obtained. A key element for the performance of the encoding we propose is based on extracting from the instruction pre-order $\sqsubset$, upper and lower bounds to the position the operations can take in the instruction list. The lower bound for a given function is obtained by inspecting the subterm where it occurs in the target stack and analyzing its operands to detect the earliest point in which the result of all them can be placed in the stack, taking into account that shared subcomputations can be obtained using a `DUP` opcode. On the other hand, the upper bound for a function is obtained by inspecting the position in the target stack they occur and analyzing the operations that use the term that is headed by this function, to obtain the latest point in which this term could be computed. From this analysis, we obtain both the upper $UB(\iota)$ and lower $LB(\iota)$ bounds for every uninterpreted (which includes the load) and store operation ι, which are extensively used in the encoding provided in the next section.

Chapter 3

Max-SMT Superoptimization

This chapter describes the second stage of the optimization process (named **SMT** in Figure 1.1) that consists in producing, from the SMS and the dependencies, a Max-SMT encoding such that any valid model corresponds to a bytecode equivalent to the initial one and optimized for the selected criterion.

3.1 Stack Representation in the SMT Encoding

The stack representation is like in [5]: the stack can hold non-negative integer constants in the range $\{0, \ldots, 2^{256} - 1\}$, matching the 256-bit words in the EVM; initial stack variables $s_0, \ldots, s_{k-1}$, represent the initial (unknown) elements of the stack; and fresh variables $s_k, \ldots, s_v$ abstract each different subterm (built from opcodes and the initial stack variables) that appears in the SMS. A stack variable of the form s_i is represented in the encoding as the integer constant $2^{256} + i$, so that all stack elements in the model are integer values. To represent the contents of the stack after applying a sequence of instructions, a bound on the number of operations b_o and the size of the stack b_s must be given. These numbers are statically computed by considering the size of the initial block and the maximum number of stack elements involved. Then, propositional variables $u_{i,j}$, with $i \in \{0, \ldots, b_s - 1\}$ and $j \in \{0, \ldots, b_o\}$, are used to denote whether there exists an element at position i in the stack after executing the first j operations, where the element $u_{0,j}$ refers to the topmost element of the stack. Quantified variables $x_{i,j} \in \mathbb{Z}$ are introduced to identify the word at position i after applying j operations, following the same format as $u_{i,j}$.

Once the stack representation has been established, we need to represent the impact of each concrete opcode on the variables previously introduced. We first split the set of instructions $\mathcal{I}$ in four subsets $\mathcal{I}_S \uplus \mathcal{I}_U \uplus \mathcal{I}_C \uplus \mathcal{I}_{St}$, where:

- $\mathcal{I}_S$ contains the basic stack operations: `PUSH`, `POP`, `DUP`k, and `SWAP`k, with $k \in \{1, \ldots, min(b_s, 16)\}$. `DUP`$k$ and `SWAP`k are restricted by b_s because they cannot deal with elements that go beyond the maximum stack size.
- $\mathcal{I}_U$ contains the non-commutative uninterpreted functions that appear in the SMS. Its subset $\mathcal{I}_L \subseteq \mathcal{I}_U$ denotes the set of load instructions.
- $\mathcal{I}_C$ contains the commutative uninterpreted functions in the SMS.

- $\mathcal{I}_{St}$ contains the write operations in memory structures. The subset $\mathcal{I}_{StC} \subseteq \mathcal{I}_{St}$ refers to the set of conflicting write operations.

The encoding for subsets $\mathcal{I}_S \uplus \mathcal{I}_U \uplus \mathcal{I}_C$ was already considered in [5], whereas $\mathcal{I}_{St}$ was left out. Instead, blocks were split when an opcode belonging to $\mathcal{I}_{St}$ was found. The inclusion of $\mathcal{I}_{St}$ instructions in the model leads to more savings in gas, as more optimizations can be applied in larger blocks.

A mapping θ is introduced to link every instruction in $\mathcal{I}$ to a non-negative integer in $\{0, \ldots, m_\iota\}$, where $m_\iota + 1 = |\mathcal{I}|$. This way, we can introduce the existentially quantified variables t_j, with $t_j \in \{0, \ldots, m_\iota\}$ and $j \in \{0, \ldots, b_o - 1\}$, to denote that the instruction ι is applied at step j when $t_j = \theta(\iota)$. For each $\iota \in \mathcal{I}$ and each possible position j in the sequence of instructions, we add a constraint to represent the impact of this combination on the stack. These constraints match the semantics of τ when projecting onto the stack component, so that we encode the elements of the stack after executing ι in terms of the ones before its execution. They follow the structure $t_j = \theta(\iota) \Rightarrow C_\iota(j)$, where $C_\iota(j)$ expresses the changes in the stack after applying ι. The constraints for $\mathcal{I}_S \uplus \mathcal{I}_U \uplus \mathcal{I}_C$ are detailed in [5], our extension in this section is only to include the constraints to reflect the impact of storage operations on the stack. For this purpose, we use an auxiliary predicate *Move* (already used in [5]) to denote that all elements in the stack are moved two positions to the right in the resulting stack state. Thus, we have the following constraint for each position j and each $\iota \in \mathcal{I}_{St}$, where o_0 and o_1 denote the position and value stored:

$$C_{St}(j, f) := \; t_j = \theta(\iota) \Rightarrow u_{0,j} \; \wedge \; u_{1,j} \; \wedge \; x_{0,j} = o_0 \; \wedge \; x_{1,j} = o_1 \; \wedge \\ \mathit{Move}(j, 2, b_s - 1, -2) \; \wedge \; \neg u_{b_s-1,j+1} \; \wedge \; \neg u_{b_s-2,j+1}$$

Finally, to complete the encoding, we express the contents of the stack before executing the instructions of the block (initial stack) and after having executed them (target stack) by assigning the corresponding values (whether constants or stack variables) to $u_{i,0}, x_{i,0}$ and to u_{i,b_o}, x_{i,b_o} respectively. The overall SMT encoding for the stack representation is denoted as C_{SFS}.

3.2 Encoding the Pre-order Relation

Once the stack representation has been formalized, we also need to consider the conflicts that appear among memory operations as part of our encoding as well as the dependencies between uninterpreted functions. All this is made by encoding the pre-order relation given in Section 2.3. We consider each pair of instructions f, f' s.t. $f \sqsubset f'$. This way we prevent conflicting operations from appearing in the wrong order in a model, by imposing that f cannot occur in the assignment after f'. This restriction can be encoded as follows:

$$L_{ord}(f, f') := \bigwedge_{LB(f') \leqslant j \leqslant UB(f')} (t_j = \theta(f') \Rightarrow \bigwedge_{j < i \leqslant UB(f)} (t_i \neq \theta(f))) \;\; \text{where } f \sqsubset f'$$

Additionally from the obtained upper and lower bounds for the instruction position we can add the following constraints

$$L_{bound}(f) := (\bigwedge_{0 \leqslant j < LB(f)} t_j \neq \theta(f)) \wedge (\bigwedge_{UB(f) \leqslant j < b_o} t_j \neq \theta(f))$$

3.3 Memory Representation in the SMT Encoding

Adding L_{ord} for the memory operations suffices to ensure that no such operation is misplaced, but in order to guarantee that the encoding returns a valid block, we need to enforce that all store operations are eventually performed, which could be encoded with the following constraint L_{est}:

$$L_{est} := \bigwedge_{\iota \in \mathcal{I}_{St}} \bigvee_{LB(\iota) \leqslant j \leqslant UB(\iota)} t_j = \theta(\iota)$$

Finally, the whole problem of finding a sequence of EVM instructions can be formulated combining the previous constraints:

$$C_{SMS} := C_{SFS} \ \wedge \bigwedge_{f \sqsubset f'} L_{ord}(f, f') \ \wedge \bigwedge_{f \in \mathcal{I}_C \cup \mathcal{I}_U \cup \mathcal{I}_{St}} L_{bound}(f) \ \wedge \ L_{est}$$

Any valid model for this encoding provides a block that is equivalent to the original one. However, our experiments have shown that there is an alternative encoding based on introducing new variables that behaves better in general and also allows us to get a better encoding for the optimization problem as we will see latter. In this new encoding we introduce variables $l_{\theta(\iota)}$, with $l_{\theta(\iota)} \in \{LB(\iota), \ldots, UB(\iota)\}$ for every uninterpreted function (including loads) and store operation, to represent the position in which the corresponding instruction is assigned. This correspondence is expressed by the constraint, which also includes the information given before in L_{bound}:

$$L_P := \bigwedge_{f \in \mathcal{I}_U \cup \mathcal{I}_C \cup \mathcal{I}_{St}} LB(f) \leq l_{\theta(f)} \leq UB(f) \ \wedge \bigwedge_{0 \leqslant j < b_o} (l_{\theta(f)} = j) \Leftrightarrow (t_j = \theta(f))$$

Note that this encoding already forces the store operations to appear at least once, so we can avoid introducing the constraint L_{est}. Using these variables, the pre-order encoding can be simplified as follows:

$$L_{lord}(f, f') := l_{\theta(f)} < l_{\theta(f')} \ \text{ where } f \sqsubset f'$$

which replaces the use of L_{ord} in the final SMS encoding constraints. We can finally formulate the alternative SMS encoding as:

$$C_{SMS} := \ C_{SFS} \ \wedge \bigwedge_{f, f' \in \mathcal{I}_C \cup \mathcal{I}_U \cup \mathcal{I}_{St}} L_{lord}(f, f') \ \wedge \ L_P$$

3.4 Optimization using Max-SMT

As in [5], we formulate the problem of finding an optimal block as a partial weighted Max-SMT problem. In this section we show, on the one hand, that the same encoding for gas optimization can be used in presence of memory operations and, on the other hand, that other optimization criterion, like bytes-size, can be used as well in our framework. Basically, in our Max-SMT problem, the *hard constraints* that must be satisfied by every model are those constraints for computing the SMS; and the *soft constraints* are used to find the optimal solution: a set of pairs $\{[C_1, \omega_1], \ldots, [C_n, \omega_n]\}$, where C_i denotes an SMT clause and ω_i its weight. For the gas model, the weights of the falsified soft constraints correspond to the gas spent for the sequence of instructions. As the solver minimizes the sum of the weights of the constraints that are not satisfied, an optimal solution found by the solver is also optimal in terms of gas spent. We denote this set of *soft constraints* as Ω_{SMS}. The gas optimization encoding used in [5] can be directly used without adding

any soft constraint for the store operations as they must be always performed by any valid model, and hence there is no choice and adding a cost will not change the optimal solution but adds unnecessary extra cost that can harm the search of an optimality proof.

Optimization based on the gas cost of every instruction is the only reasonable criterion for smart contract users, since they pay gas whenever they execute a function of the contract. However, this is not the case for the smart contract developers since, when a contract is deployed, 200 units of gas must be paid for each non-zero byte of the EVM binary code. An optimizer based on instruction gas optimization may fail to address this relevant cost of using smart contracts. Notoriously, the solc compiler takes into account this cost and, in some cases, it intentionally does not fully replace expressions that have a constant result by the value they represent if this constant is a large number, since the needed `PUSH` instructions will need many more non-zero bytes and hence will increment the deployment gas cost. For instance, if we want to have $2^{256}-1$ on the top of the stack we can either push a zero and perform the bitwise `NOT` operation, which has gas cost 6 and non-zero bytes length 2 or push $2^{256}-1$ directly which has gas cost 3 but non-zero bytes length 33.

When the bytes-size criterion is selected, we disable the application of the simplification rules of [5] that increase the byte-size and, besides, propose the next approach based on the *bytes-size model* for the Max-SMT encoding. This model is fairly simple except for the handling of the `PUSH` related instructions, denoted as $\mathcal{I}_P$ in what follows. All instructions that are not in $\mathcal{I}_P$ use exactly one byte. Instead `PUSHx` instructions take one byte to specify the opcode itself, and x bytes to include the pushed value. A first attempt to encode the weight of the `PUSHx` we tried was based on precisely describing the size in bytes based on the corresponding 32 options that x can take in terms of number of bytes (recall that in EVM we have 256-bit words). This encoding is precise, but did not work in practice. An alternative, much simpler encoding, is based on the observation that numerical values can only appear in a model because at least once the corresponding `PUSHx` instruction is made. Later on, this value can be repeated using `DUP`, which has a minimal cost *wrt.* size of bytes, but if the block is large, some `SWAP` operation may also be needed. To make the encoding perform well in practice we need to associate a single constant weight to all `PUSHx` operations, that is high enough to avoid models where expensive `PUSHx` operations are performed more than once instead of duplicating them. Our experiments have shown that a weight of 5 is enough to obtain optimal results for the sizes of blocks that the Max-SMT is able to handle. Then, we can assume `NOP` instructions cost 0 units, instructions in $\mathcal{I}_p$ costs 5 units and the remaining instructions cost 1 unit. Hence, three disjoint sets are introduced to match previous costs: $W_0 := \{\texttt{NOP}\}$, $W_5 := \mathcal{I}_p$ and $W_1 := \mathcal{I} \setminus (W_0 \uplus W_5)$. Ω' bytes-size model is followed directly:

$$\Omega'_{SMS} := \bigcup_{0 \leqslant j < b_o} \{[t_j = \theta(\texttt{NOP}), 1], [\bigvee_{\iota \in W_0 \uplus W_1} t_j = \theta(\iota), 4]\}$$

Example 4. *The optimized bytecode returned by GASOL 0.1 for the gas criterion is* `PUSH24* PUSH 80 PUSH 40 MSTORE PUSH 1 SLOAD PUSH32* AND PUSH21* OR PUSH32* AND OR PUSH 1 SSTORE CALLVALUE CALLVALUE ISZERO`*,we use* `*` *to skip large constants, which achieves a reduction of 5905 units* wrt. *the original version and is proven optimality. For the bytes-size criterion, GASOL 0.1 times out due to the larger size of the block when size-increasing simplification rules are disabled. This issue will be discussed in Chapter 4.*

Chapter 4

Implementation and Experiments

This chapter provides further implementation details and describes our experimental evaluation. The GASOL 0.1 tool is implemented in Python and uses as Max-SMT solver OptiMathSAT (OMS) [7] version 1.6.3 (which is the optimality framework of MathSAT). The aim of the experiments is to assess the effectiveness of our proposal by comparing it with the previous tool SYRUP. A timeout is given to the tools to specify the maximum amount of time that they can use for the analysis of each block. The timeout given to GASOL 0.1 must be larger than for SYRUP because it works on less and larger blocks in order to analyze the same contract. We have used as timeout for SYRUP 10 sec, and for GASOL 0.1, we use 10*(#store+1) sec, as this would correspond to the addition of the times in SYRUP given to the partitioned blocks. It should be noted though that the cost of the search to be performed grows exponentially with the number of additional instructions. Therefore, in spite of giving a similar timeout, GASOL 0.1 might time out in cases in which it has to deal with rather large blocks, while SYRUP does not on the corresponding smaller partitioned blocks. For this reason, we have implemented two additional versions: gasol$_{all}$ splits the blocks at all stores as SYRUP, and gasol$_{24}$ splits at store instructions only those blocks that have a size larger than 24 instructions. This is because we have observed during experimentation that the SMT search does not terminate in a reasonable time from that size on. The 24-partitioning starts from the end of the block and splits it if it finds a store. If the partitioned sub-block (from the start) still has a size larger than 24, further partitioning is done again if a new store is found from its end, and so on. Still, depending on where the stores are, the resulting blocks can have sizes larger than 24, as it happens in SYRUP as well. Further experimentation will be needed to come up with intelligent heuristics for the partitioning. The gasol versions implement all techniques described in the report, including the SMT encoding dependencies between uninterpreted functions as described in Section 2.3. We have the following versions of GASOL and SYRUP in the evaluation:

- syrup$_{cav}$: this is the original tool from [5].
- gasol$_{all}$: it splits the blocks at all stores as in syrup$_{cav}$.
- gasol$_{24}$: it performs the 24-partitioning described above.
- gasol$_{none}$: it does not perform any additional partitioning of blocks.

	$\mathbf{G}_{normal}$	$\mathbf{G}_{timeout}$	$\%\mathbf{G}_{total}$	$\mathbf{T}_{gas}$	$\mathbf{B}_{normal}$	$\mathbf{B}_{timeout}$	$\%\mathbf{B}_{total}$	$\mathbf{T}_{bytes}$
syrup$_{cav}$	35689	11129	0.62%	142,93	–	–	–	–
gasol$_{all}$	36344	11975	0.64%	120,21	3712	2213	2.64%	200,17
gasol$_{24}$	38765	12336	0.68%	327,36	4315	2238	2.92%	558,48
gasol$_{none}$	39977	0	0.53%	850,75	3871	0	1.72%	1194,38
gasol$_{best}$	41307	13197	0.72%	933,66	4676	2692	3.28%	1313,36

Table 4.1: Overall gains in gas and bytes-size and overheads

- gasol$_{best}$: it uses gasol$_{all}$, gasol$_{24}$, and gasol$_{none}$, as a portfolio of possible optimization results (running them in parallel) and keeps the best result.

We run the tools using the gas usage and the bytes-size criteria in Section 3.4. As already mentioned, SYRUP in [5] did not include the bytes-size criterion, marked as "–" in the figures. Experiments have been performed on an Intel Core i7-7700T at 4.2GHz x 8 and 64Gb of memory, running Ubuntu 16.04.

The benchmark set. We have downloaded the last 30 verified smart contracts from Etherscan that were compiled using the version 8 of solc and whose source code was available as of June 21, 2021. The reason for this selection is twofold: (1) we require version 8 in order to be able to apply the latest solc optimizer and start from a worst-case scenario in which we have the most possible optimized version and, this way, assess if there is room for further optimization and, in particular, for the two types of gains achievable by GASOL 0.1, (2) we want to make a random choice (e.g., the last 30) rather than picking up contracts favorable to us. The benchmarks in [5] require using an old version of the compiler (at most 4), hence the last solc optimizer cannot be activated. The source code of GASOL 0.1 as well as the smart contracts analyzed are available at https://github.com/costa-group/gasol-optimizer. We provide the results of analyzing the compiled smart contracts generated by the version 0.8.9 of solc with the complete optimization options. The total number of blocks, given by **BLK** for the 30 contracts is 12,378. Within them, there are 1,044 `SSTORE` instructions, 6,631 `MSTORE` and 43 `MSTORE8`. These memory instructions are used by SYRUP to split the basic blocks, while GASOL 0.1 does not split them always as explained above. This results on 15,416 blocks when considering the additional 24-partitioning, 13,030 without partitioning at stores by gasol$_{none}$, and 20,467 blocks by syrup$_{cav}$ and gasol$_{all}$. As in [5] all tools split blocks at instructions like `LOGX` or `CODECOPY` .

Efficiency gains and performance overhead. Table 4.1 shows the overall gas and size gains and the optimization time (in minutes). The total gas consumed by all contracts before running the optimizers is 7,538,907, and the bytes-size is 224,540. As it is customary, we are calculating such gas (resp. size) as the the sum of the gas (resp. size) consumed by all EVM instructions in the considered contracts.[1] For those EVM instructions that do not consume a constant and fixed amount of gas, such as `SSTORE`, `EXP` or `SHA3`, we choose the lower bound that they may consume. Column $\mathbf{G}_{timeout}$ represents the gas saved by

[1]Estimating the actual gains of executing transactions on the involved contracts is a research problem on its own which has been subject of other work, e.g., [10, 9, 12, 4].

	#B	$\mathbf{Alr}_g$	$\mathbf{Opt}_g$	$\mathbf{Bet}_g$	$\mathbf{Non}_g$	$\mathbf{Tout}_g$	$\mathbf{Alr}_b$	$\mathbf{Opt}_b$	$\mathbf{Bet}_b$	$\mathbf{Non}_b$	$\mathbf{Tout}_b$
syrup$_{cav}$	20467	70.54	27.01	0.47	0.08	1.9	–	–	–	–	–
gasol$_{all}$	20467	70.63	27.36	0.64	0.35	1.02	83.25	12.83	1.2	0.69	2.03
gasol$_{24}$	15416	62.2	33.79	1.47	0.91	1.63	75.48	16.29	3.21	1.78	3.24
gasol$_{none}$	13030	65.48	25.3	3.81	0.34	5.07	73.44	11.7	3.1	2.57	9.19

Table 4.2: Optimization report (%) for SYRUP and GASOL 0.1

the optimized blocks that reached the time out in gasol$_{none}$ with no result, $\mathbf{G}_{normal}$ the gas savings for the remaining ones, and $\mathbf{G}_{total}$ the total gains computed as the sum of the previous two, given as a percentage *wrt.* the initial gas consumption. Columns **B** have the analogous meanings for size and **T** gives the time in minutes. The first observation is that our proposal of using dependencies in gasol$_{all}$ pays off, as we achieve larger gains than syrup$_{cav}$ in less time. The second observation is that the gains in gas of GASOL 0.1 are notably larger for blocks that do not time out $\mathbf{G}_{normal}$, as a larger search space can be explored. However, those blocks that would require a larger timeout might behave worse than the syrup$_{cav}$ and gasol$_{all}$ versions working on smaller blocks, as the original bytecode is taken as the optimization result in case of timeout. This happens sometimes in the version gasol$_{24}$, and more often in gasol$_{none}$. The problem is exacerbated for the bytes-size criterion because due to the restriction to apply size-increasing simplification rules the blocks are even larger. Even in $\mathbf{B}_{normal}$ the gain is smaller for gasol$_{none}$ than for gasol$_{24}$. This is because $\mathbf{B}_{normal}$ includes timeouts for which a solution is found. Our solution to mitigate the huge computation demands required in these cases is in row gasol$_{best}$ that runs in parallel gasol$_{all}$, gasol$_{24}$ and gasol$_{none}$ and returns the best result. As it can be seen gasol$_{best}$ clearly outperforms the other systems in gas and size gains. As regards the overhead, it is also the most expensive option, as it reaches the timeout more often than the other systems and these timeouts are accumulated to the time. However, as superoptimizers are often used as offline optimization tools, which are run only prior to deployment, we argue that the gains compensate the further optimization time. Finally, it remains to be investigated the interaction between the two optimization criteria, namely how the reduction in bytes-size affects the gas consumption and viceversa.

Impact of phases 1 and 2. We would also like to estimate how much is gained in gasol$_{best}$ by applying the simplification rules and how much is gained by the SMT encoding. Regarding the simplification rules on memory, gasol$_{best}$ has applied 6 rules on storage and 11 on memory: 15 of them correspond to the rule i) (4 on storage and 11 on memory) described in Definition 2, and 2 to the rule ii) (both on storage). Rule iii) is never applied on this benchmark set, but we have applied it when optimizing other real smart contracts. As regards the percentage of the gains, 14.6% of the gas savings come from applying the memory rules, 34.4% from the stack rules and 51% is saved by the use of the Max-SMT solver. As in [5], the gains due to each phase are roughly half (i.e., 50% each). Regarding the simplification rules on stack for the gas criterion, their application has increased 11.4% in gasol$_{best}$ because it works on larger blocks and has more opportunities to apply them. However, when selecting the bytes-size criteria, there are less simplification rules applied (namely 96% less) as when the rules generate larger code in terms of size they are not applied (see Section 3.4).

Optimality results. Table 4.2 provides additional detailed information, which is also part of the *optimization report* of Figure 1.1. Column #**B** shows the total number of blocks analyzed in each case, depending on the partitioning. In the remaining columns, we show the percentages of: Column **Alr** corresponds to the percentage of blocks that are already optimal, i.e., those blocks that cannot be optimized because they already consume the minimal amount of gas; **Opt** shows the percentage of blocks that have been optimized and the SMT solver has been able to prove that the solution found is optimal, i.e., it consumes the minimum amount of gas needed to generate the provided SMS; **Bet** shows the percentage of blocks that have been optimized and therefore, consume less gas than the original ones, but the solutions have not been proved to be optimal; **Non** contains the percentage of blocks that have not been optimized and the solver has not been able to prove if they are optimal, i.e., the solution found is the original one but it may exist a better one; **Tout** corresponds to the percentage of blocks where the solver reached the timeout without finding a model. The subscripts $_b$ are the analogous for the bytes-size criterion. We can observe in the table that gasol_{none} times out in more cases due to the larger sizes of the blocks that it optimizes, but the percentages of blocks for which it finds a better and optimal solution are notably high. It should be noted also that the results of SYRUP (and gasol_{all}) and, to a less extend, of gasol_{24} *wrt.* optimality are weaker. This is because they work on strictly smaller blocks and hence they can prove optimality for the partitioned blocks, but when glued together, the optimality may be lost. This is also the reason why the results for gasol_{best} are not included, because it mixes different notions of optimality and the concepts are not well-defined. Due to this weaker optimality, the **Opt** and **Bet** results are only slightly better for GASOL 0.1 than for SYRUP. However, the truly important aspect is that the actual gas and size gains for GASOL 0.1 in Table 4.1 are notably larger.

Bibliography

[1] Compiler Input and Output JSON Description. https://docs.soliditylang.org/en/v0.8.7/using-the-compiler.html#compiler-input-and-output-json-description.

[2] Welfare contract. https://etherscan.io/address/0x3E873439949793e8c577E08629c36Ed8c184e7D9#code.

[3] The solc optimizer, 2021. https://docs.soliditylang.org/en/v0.8.7/internals/optimizer.html.

[4] 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.

[5] 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.

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

[7] 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.

[8] F. Contro, M. Crosara, M. Ceccato, and M. Dalla Preda. Ethersolve: Computing an accurate control-flow graph from ethereum bytecode. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, pages 127–137. IEEE, 2021.

[9] 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.

[10] Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *ISoLA*, volume 11247 of *LNCS*, pages 450–465. Springer, 2018.

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

[12] 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.