GASOL: GAS Optimization tooLkit

Report 1: Proof-of-concept of GASOL[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 1 of the GASOL project. The main decisions taken along this Stage of the project have been discussed and agreed with the members of the YUL team. The next items summarize the main achievements:

1. We have developed a first version of the super-optimization tool that takes as input the asm json generated by the solc compiler and that produces in the output the optimized bytecode, using the asm json format, as well;

2. We have implemented the generation of a log file to enable verification using Etherscan and evaluated its overhead;

3. We have tested the tool on the last verified contracts on Etherscan compiled with the version 0.8.5 of solc and compared experimentally the gains against the YUL optimizer and, also, when GASOL is used together with the YUL optimizer;

4. We have made the tool available in a github repository and added instructions for its usage.

Chapter 1

An Overview of GASOL

GASOL is a GAS Optimization tooLkit for Ethereum smart contracts. It applies the so-called *superoptimization* [4] technique to reduce the amount of gas consumed by each of the *basic blocks* (sequences of EVM instructions) of the smart contract being optimized. A *basic block* is a maximal sequence of straight-line consecutive code in the program with the properties that the flow of control can only enter the block through the first instruction in the block, and can only leave the block at the last instruction. Super-optimization is a general technique that tries to find the optimal sequence of instructions that produces the same result than the original one by trying out all the possible sequences of instructions. In its current version, GASOL 0.1 tries to find a sequence of EVM instructions that produces the same "stack" as the original block, but consumes a smaller amount of gas, i.e., the optimizations are mainly focused at this stage of the project on the stack operations.

Figure 1.1 overviews the architecture of GASOL. It takes as *input* the assembly EVM bytecode stored in a single json file (asm json file in what follows), which is generated by the Solidity compiler `solc`. It obtains from the bytecode the basic blocks to be optimized after parsing the json file generated by the compiler. The basic blocks of the contract may be split into smaller *sub-blocks* when we find non-stack operations so that the optimizations are applied to sequences of stack operations at the stage of the project. Namely, we split on instructions whose results are not reflected in the stack such as store (`SSTORE`, `MSTORE`) or log (`LOGX`) operations.

Example 1. *A fragment of the input asm json is shown below. Note that the EVM instructions are stored as a sequence of json elements that contain also some metadata related to the source code used to generate the bytecode. In addition, it also contais some information such as the pushed values. The asm json is parsed and the input is split into basic blocks.*

...
{"begin":601,"end":725,"name":"JUMP","source":1,"value":"[out]"},
{"begin":731,"end":1150,"name":"tag","source":1,"value":"16"},
{"begin":731,"end":1150,"name":"JUMPDEST","source":1},
{"begin":897,"end":901,"name":"PUSH","source":1,"value":"0"},
{"begin":935,"end":937,"name":"PUSH","source":1,"value":"20"},
{"begin":924,"end":933,"name":"DUP3","source":1},
{"begin":920,"end":938,"name":"ADD","source":1},
{"begin":912,"end":938,"name":"SWAP1","source":1},
{"begin":912,"end":938,"name":"POP","source":1},
{"begin":984,"end":993,"name":"DUP2","source":1},
{"begin":978,"end":982,"name":"DUP2","source":1},
{"begin":974,"end":994,"name":"SUB","source":1},
{"begin":970,"end":971,"name":"PUSH","source":1,"value":"0"},
{"begin":959,"end":968,"name":"DUP4","source":1},
{"begin":955,"end":972,"name":"ADD","source":1},
{"begin":948,"end":995,"name":"MSTORE","source":1},
{"begin":1012,"end":1143,"name":"PUSH [tag]","source":1,"value":"34"},
{"begin":1138,"end":1142,"name":"DUP2","source":1},
{"begin":1012,"end":1143,"name":"PUSH [tag]","source":1,"value":"25"},
{"begin":1012,"end":1143,"name":"JUMP","source":1,"value":"[in]"},
{"begin":1012,"end":1143,"name":"tag","source":1,"value":"34"},
{"begin":1012,"end":1143,"name":"JUMPDEST","source":1}
...

The fragment of asm json shown contains the following basic block, that goes from the first `JUMPDEST` to the `JUMP` instructions.

```
1  JUMPDEST        7  POP            13  ADD
2  PUSH1 0x00      8  DUP2           14  MSTORE
3  PUSH1 0x20      9  DUP2           15  PUSH tag 34
4  DUP4           10  SUB            16  DUP2
5  ADD            11  PUSH1 0x00     17  PUSH tag 25
6  SWAP1          12  DUP4           18  JUMP
```

Finally, the basic block shown contains a `MSTORE` at Line14 (L14 for short) whose effect is not reflected on the stack. Hence, it is split into the following subblocks, that will be analyzed by GASOL:

```
 2  PUSH1 0x00   |
 3  PUSH1 0x20   |
 4  DUP4         |
 5  ADD          |
 6  SWAP1        |
 7  POP          |   14  PUSH tag 34
 8  DUP2         |   15  DUP2
 9  DUP2         |   16  PUSH tag 25
10  SUB          |
11  PUSH1 0x00   |
12  DUP4         |
13  ADD          |
```

After that, we compute the *stack functional specification* (SFS in the figure) for each block. Basically, the SFS is our *intermediate representation* which defines the input and the target stacks of each block. The target stack is defined in terms of the elements stored in the input stack. It is computed by symbolic execution, as described in [2]. We note that, in order to compute the SFS, we need to infer the minimum stack size needed to execute each basic block. This can be done statically, as the number of stack elements that each evm instruction consumes and produces are known. We also apply *simplification rules* at this level that capture the semantics of the EVM instructions. The rules applied by GASOL are described in Appendix A.

Example 2. *In order to execute the first subblock shown above, the stack has to contain at least 2 elements. The SFS is computed by symbolically executing the block. In this case, we obtain*

$$[s_0,\ \texttt{SUB(ADD(}s_1\texttt{,32),}s_0\texttt{)},\ \texttt{ADD(}s_1\texttt{,32)},\ s_0,s_1]$$

Initially, the stack contains two elements, represented by the symbolic variables s_0 and s_1 where s_0 corresponds to the top-most element. The SFS describes the state of the stack after executing the block in terms of the initial stack (s_0 and s_1). Note that the second `ADD` instruction does not appear in the SFS. It has been simplified by a rule as it adds a 0 to the variable s_0. Hence, it has been replaced to the variable s_0 located in the top of the stack.

The second subblock shown in Example 1 needs a stack element to be executed. Similarly, we obtain the SFS [tag_{34},s_0,tag_{25}].

Once the SFS is computed for all the blocks, we generate the encoding of finding the optimal EVM bytecode as a Max-SMT problem. An off-the-shelf Max-SMT solver is used by GASOL as a black-box to obtain a *model*, i.e., a sequence of assignments that satisfies the constraints specified in the encoding. Finally, the model has to be decoded in order to synthesize the new block.

Example 3. *After decoding the model generated by the SMT solver when it takes the encoding of the first SFS shown in Example 2 as input, we obtain the following sequence of EVM instructions:*

```
DUP2 PUSH1 0x20 ADD DUP2 DUP2 SUB DUP3
```

In addition, the solver is able to prove that the new block is optimal, i.e., it consumes the minimum amount of gas needed to produce the same stack. While the original block consumes 35 units of gas, the optimized one consumes only 21 units of gas.

The block may have been optimized meaning that it consumes less gas than the original block, or it can be the original block. In addition, the solver returns if the solution found is *optimal* meaning that the block found consumes the minimum amount of gas needed to generate the same stack as the original one.

In addition, GASOL includes a *lightweight soundness check*. It verifies the correctness of the optimization as follows: we have implemented a checker that computes the SFS for the optimized block and proves the equivalence between it and the original SFS.

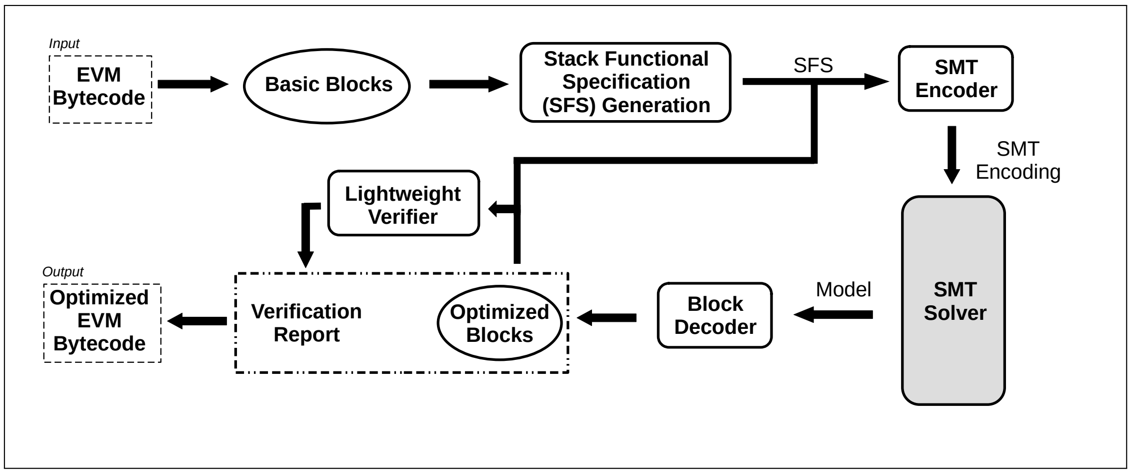

Figure 1.1: Architecture of GASOL. White boxes are components implemented by us. Gray boxes are off-the-shelf tools.

Example 4. *If we compute the SFS of the optimized block shown in Example 3 we obtain [s_0, `SUB(ADD(32,`s_1`),`s_0`)`, `ADD(32,`s_1`)`, s_0,s_1]. Hence, the lightweight verifier has to prove the equivalence between this SFS and [s_0, `SUB(ADD(`s_1`,32),`s_0`)`, `ADD(`s_1`,32)`, s_0,s_1], described in Example 2. As it can be seen, the SFSs of both blocks are equivalent as the `ADD` instruction is commutative. Therefore, GASOL proves that the optimized block generates the same stack as the original one.*

Finally, GASOL reconstructs the EVM bytecode and produces a new asm json with the optimized blocks.

Chapter 2

Experimental Results

In this section we present the main results of the experiments carried out to study the effectiveness and efficiency of GASOL. We perform a comparison with the usage of the YUL optimizer and when both GASOL and YUL are applied together. We intend to give some hints on the potential benefits of including GASOL as part of the Solidity compiler.

2.1 The Test-bed and Set-up

We have downloaded the last 30 verified 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 will provide the results of analyzing the run-time asm json files of the smart contracts generated by the version 0.8.5 of solc. The source code of the smart contracts analyzed is available at `https://github.com/costa-group/gasol-optimizer/tree/main/examples/solidity`. GASOL is implemented in Python and uses as SMT solver OptiMathSAT (OMS) [3] version 1.6.3 (which is the optimality framework of MathSAT). The experiments have been performed on an Intel Core i7-7700T at 4.2GHz x 8 and 64Gb of memory, running Ubuntu 16.04.

2.2 Comparing and Combining GASOL and YUL

We intend to analyze the gains obtained by GASOL and compare them with those obtained by the YUL optimizer, and with the combined use of GASOL and YUL. To this purpose, we have three different settings:

1. YUL+No-GASOL: It corresponds to the use of the YUL optimizer of solc compiler (−−optimize option).
2. No-YUL+GASOL: It corresponds to the use of GASOL on the EVM bytecode generated by solc compiler when the Yul optimizer is disabled (−−optimize −−no−optimize −yul options).
3. YUL+GASOL: It corresponds to the use of GASOL on the EVM bytecode generated by solc compiler when it is executed with the complete optimization options (−−optimize option).

File	YUL+No-GASOL				No-YUL+GASOL					YUL+GASOL				
	G	G’	L	L’	G	G’	L	L’	T(s)	G	G’	L	L’	T(s)
#0	500079	1803(0.36%)	5466	500(9.15%)	500079	2860.0(0.57%)	5466	511(9.35%)	402.86	500079	4732.0(0.95%)	5466	1020(18.66%)	390.14
#1	502749	1676(0.33%)	5196	466(8.97%)	502749	2819.0(0.56%)	5196	510(9.81%)	351.75	502749	4565.0(0.91%)	5196	981(18.88%)	366.0
#2	155110	1888(1.22%)	4749	533(11.22%)	155110	3466.0(2.23%)	4749	536(11.29%)	157.74	155110	5305.0(3.42%)	4749	1064(22.4%)	150.75
#3	306152	2868(0.94%)	5719	816(14.27%)	306152	3602.0(1.18%)	5719	544(9.51%)	309.3	306152	6548.0(2.14%)	5719	1368(23.92%)	266.8
#4	46636	887(1.9%)	1227	246(20.05%)	46636	444.0(0.95%)	1227	62(5.05%)	98.1	46636	1328.0(2.85%)	1227	300(24.45%)	84.3
#5	169905	1162(0.68%)	2494	323(12.95%)	169905	1242.0(0.73%)	2494	308(12.35%)	112.32	169905	2423.0(1.43%)	2494	634(25.42%)	99.56
#6	479522	878(0.18%)	6816	239(3.51%)	479522	5854.0(1.22%)	6816	605(8.88%)	361.37	479522	6811.0(1.42%)	6816	873(12.81%)	351.46
#7	1652921	1894(0.11%)	11007	529(4.81%)	1652921	6634.0(0.4%)	11007	1133(10.29%)	506.49	1652921	8613.0(0.52%)	11007	1675(15.22%)	508.93
#8	350595	1846(0.53%)	5773	511(8.85%)	350595	3787.0(1.08%)	5773	429(7.43%)	395.82	350595	5718.0(1.63%)	5773	970(16.8%)	301.42
#9	156786	612(0.39%)	1716	172(10.02%)	156786	1061.0(0.68%)	1716	155(9.03%)	119.21	156786	1749.0(1.12%)	1716	316(18.41%)	137.24
#10	213064	1223(0.57%)	4606	345(7.49%)	213064	3833.0(1.8%)	4606	529(11.48%)	157.13	213064	5071.0(2.38%)	4606	869(18.87%)	167.32
#11	162376	1475(0.91%)	3768	427(11.33%)	162376	1852.0(1.14%)	3768	336(8.92%)	190.3	162376	3310.0(2.04%)	3768	742(19.69%)	189.5
#12	32457	745(2.3%)	1543	217(14.06%)	32457	1225.0(3.77%)	1543	143(9.27%)	63.24	32457	1930.0(5.95%)	1543	348(22.55%)	46.33
#13	261702	1087(0.42%)	2966	305(10.28%)	261702	1332.0(0.51%)	2966	250(8.43%)	161.44	261702	2415.0(0.92%)	2966	541(18.24%)	187.1
#14	563227	1583(0.28%)	5394	442(8.19%)	563227	2893.0(0.51%)	5394	511(9.47%)	359.52	563227	4571.0(0.81%)	5394	967(17.93%)	360.35
#15	65070	1460(2.24%)	3128	426(13.62%)	65070	2302.0(3.54%)	3128	286(9.14%)	125.41	65070	3709.0(5.7%)	3128	692(22.12%)	106.68
#16	502619	1676(0.33%)	5154	466(9.04%)	502619	2803.0(0.56%)	5154	493(9.56%)	353.3	502619	4549.0(0.91%)	5154	988(19.17%)	368.36
#17	389179	2055(0.53%)	9112	777(8.53%)	389179	5185.0(1.33%)	9112	784(8.6%)	331.61	389179	7336.0(1.88%)	9112	1558(17.1%)	321.59
#18	90568	897(0.99%)	2280	253(11.1%)	90568	1847.0(2.04%)	2280	231(10.13%)	79.93	90568	2774.0(3.06%)	2280	475(20.83%)	71.17
#19	513882	1097(0.21%)	4398	286(6.5%)	513882	1738.0(0.34%)	4398	214(4.87%)	532.41	513882	2827.0(0.55%)	4398	551(12.53%)	471.48
#20	222146	846(0.38%)	2098	234(11.15%)	222146	1036.0(0.47%)	2098	189(9.01%)	99.14	222146	1908.0(0.86%)	2098	435(20.73%)	101.63
#21	23	0(0.0%)	9	0(0.0%)	23	0.0(0.0%)	9	0(0.0%)	0.05	23	0.0(0.0%)	9	0(0.0%)	0.05
#22	777500	1539(0.2%)	7060	426(6.03%)	777500	4320.0(0.56%)	7060	713(10.1%)	240.38	777500	5749.0(0.74%)	7060	1092(15.47%)	277.58
#23	113902	1802(1.58%)	4094	522(12.75%)	113902	2964.0(2.6%)	4094	401(9.79%)	164.7	113902	4707.0(4.13%)	4094	919(22.45%)	133.34
#24	502722	1676(0.33%)	5188	466(8.98%)	502722	2819.0(0.56%)	5188	498(9.6%)	351.2	502722	4565.0(0.91%)	5188	993(19.14%)	366.55
#25	77633	1576(2.03%)	3472	458(13.19%)	77633	2570.0(3.31%)	3472	321(9.24%)	133.7	77633	4058.0(5.23%)	3472	765(22.03%)	125.0
#26	581295	1485(0.26%)	5408	416(7.69%)	581295	3560.0(0.61%)	5408	589(10.89%)	328.29	581295	5122.0(0.88%)	5408	1021(18.88%)	340.82
#27	132180	1641(1.24%)	3269	468(14.32%)	132180	2220.0(1.68%)	3269	293(8.96%)	123.75	132180	3841.0(2.91%)	3269	756(23.13%)	122.97
#28	65070	1460(2.24%)	3128	426(13.62%)	65070	2317.0(3.56%)	3128	288(9.21%)	107.17	65070	3717.0(5.71%)	3128	694(22.19%)	96.8
#29	174556	1531(0.88%)	5968	629(10.54%)	174556	3472.0(1.99%)	5968	503(8.43%)	300.48	174556	5170.0(2.96%)	5968	1135(19.02%)	298.79

Table 2.1: Gas and size savings.

Table 2.2 shows the gains in terms of gas and size of the optimized blocks for the 30 Solidity files analyzed. In total, we have analyzed 19318 blocks for the No-YUL+GASOL setting, and 17305 basic blocks for the others. The results are split into three parts for each of the three settings. Column **G** represents the gas consumed by the instructions of the corresponding contract, i.e., it is computed as the sum of the gas consumed by all EVM instructions in the contract. 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. **G'** represents the gas saved by the optimized contracts. Similarly, **L** represents the number of instructions of the original contract analyzed and **L'** the number of instructions saved in the new optimized version. Note that GASOL can only produce solutions that are at most as large as the original as it uses the original size as a bound, and it will never produce larger solutions than the original one. **T** represents the time in seconds needed by GASOL to generate the optimized contract.

Globally, if we consider all the data obtained from the analysis of the 30 contracts, we get that the gas savings are: 0,43% in the YUL+No-GASOL setting and 0,84% in the No-YUL+GASOL setting. Therefore, the gains obtained by GASOL are higher (double) than those obtained by the YUL optimizer when they are applied to the same EVM bytecodes (YUL+No-GASOL and No-YUL+GASOL blocks in Table 2.2). However, while the YUL optimizer is executed in a few seconds, GASOL needs much more time to optimize the contract.

Interestingly, the gas savings of YUL+GASOL are 1,28%. Therefore, GASOL is able to optimize more gas when it is applied together with YUL (YUL+GASOL setting) than when it is applied alone (No-YUL+GASOL setting).

Regarding the size of the contracts analyzed, the settings YUL+No-GASOL and No-YUL+GASOL save a similar amount of instructions in most of the cases. Despite this fact, No-YUL+GASOL is able to optimize more gas than YUL+No-GASOL as reported above. Also, there a few contracts where, though No-YUL+GASOL optimizes less instructions than YUL+No-GASOL, but it saves more gas. Interestingly, when we combine the YUL optimizer with GASOL (setting YUL+GASOL) we are able to optimize even more size (and gas), as shown in column **L'** in the block YUL+GASOL in Table 2.2. They represent the size gains with respect to the original size of the bytecode generated before the execution of the YUL optimizer. Globally, YUL alone saves 9,32% and GASOL alone 10%. However, when we apply GASOL after YUL the reduction goes up to 18%.

Figure 2.2 provides more detailed information for each of the optimization options. Column **A** corresponds to the number of blocks that are already optimal, i.e., those blocks that cannot be optimized because they already consume the minimal amount of gas. **O** shows the number 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 SFS provided. **B** shows the number 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. **N** contains the 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. **T** corresponds to the number of blocks where the solver reached the timeout and hence, a model could not been found.

File	No-YUL+GASOL					YUL+GASOL				
	A	O	B	N	T	A	O	B	N	T
#0	444	237	4	1	13	362	238	4	2	18
#1	424	231	4	1	12	346	229	4	2	17
#2	464	251	3	0	0	387	245	4	0	2
#3	582	272	7	3	7	455	284	4	2	10
#4	124	42	1	0	4	85	40	2	0	5
#5	228	103	1	0	5	174	105	0	0	5
#6	629	277	9	0	9	577	298	11	0	11
#7	887	499	8	0	18	806	498	9	1	19
#8	509	244	5	0	19	430	257	5	0	15
#9	300	113	2	0	4	244	99	1	0	5
#10	420	235	2	0	5	365	233	4	0	5
#11	377	159	2	0	9	315	159	3	0	10
#12	166	96	2	0	0	130	91	1	0	0
#13	428	162	2	1	7	361	136	0	1	8
#14	461	237	8	1	12	387	238	8	2	15
#15	332	186	1	0	1	276	172	2	0	2
#16	422	227	4	0	13	344	224	5	2	17
#17	927	407	6	0	2	797	406	6	0	6
#18	248	134	0	0	2	203	126	2	0	1
#19	323	125	18	2	22	282	137	8	1	30
#20	340	104	2	0	2	265	98	0	0	4
#21	5	0	0	0	0	5	0	0	0	0
#22	618	381	1	0	9	547	377	1	1	16
#23	430	238	2	1	1	361	224	2	0	2
#24	423	231	4	1	12	345	229	4	2	17
#25	374	204	3	0	1	318	194	2	0	3
#26	420	263	4	0	13	353	263	5	2	15
#27	328	157	4	0	1	258	155	3	1	2
#28	322	187	1	0	0	271	175	2	0	1
#29	672	255	5	0	2	565	261	5	0	7

Table 2.2: Detailed Information for Blocks Analyzed by GASOL.

EVM Version	$\mathbf{T_1}$	$\mathbf{T_2}$	$\mathbf{T_3}$
Non-optimized	1842	2312	1500
No-YUL-optimized	4421	22	179
Full-optimized	4599	15	175

Table 2.3: Simplification rules identified by GASOL

2.3 Simplification Rules

GASOL applies the same simplification rules as the Solidity optimizer (those specified in [1]). We have classified these rules in three different groups:

- T_1: Evaluation of arithmetic and bit-wise operations over constant arguments (rules (1) and (2) in Figure A.1).
- T_2: Application of identity rules on arithmetic, and bit-wise operations where at least one of the arguments is symbolic (rules (3)-(7), (10)-(12), (14)-(19), (20)-(22), (24), (25), (27), (28), (39)-(45), (47)-(49) and (54)-(56) in Figure A.1).
- T_3: Application of rules related with the semantics of the EVM instructions.

Table 2.3 shows the simplification rules applied by GASOL on three versions of the EVM bytecode: (i) the row non-optimized corresponds to the rules applied by GASOL in the EVM bytecode generated by the compiler without any optimization flag, (ii) no-yul-optimized corresponds to the rules applied in the optimized EVM bytecode generated with the YUL optimizer disabled and, (iii) the row full-optimized contains the rules identified by GASOL when analyzing the fully optimized EVM bytecode of the smart contracts. Surprisingly, GASOL identifies a higher number of simplification rules that operate over constant arguments when the EVM bytecode has already been optimized by the compiler (though the number of basic blocks analyzed is smaller). We have found out that the increase of these type of rules comes from the replacement of constant values that appear in the non-optimized version by the computation process of these values. For instance, in some cases the instruction `PUSH` $2^{160} - 1$ is replaced by the sequence `PUSH 1 PUSH 160 PUSH 160 EXP SUB`. Additionally, the number of `SHL` instructions over constant arguments increases considerably. When analyzing the basic blocks of the non-optimized version of the bytecode, GASOL only identifies 91 simplification rules that involve the `SHL` instruction. However, in the basic blocks of the no-yul-optimized and full-optimized versions, GASOL identifies 2814 and 2908 simplification rules related to `SHL` respectively. On the other hand, the number of simplification rules of type 2 and 3 are reduced when the EVM bytecode is optimized.

Finally, Table 2.4 reports on the number of rules applied by GASOL on each of the analyzed Solidity files and the part of the gas savings that it is due to such rule simplification. The gas is computed as the sum of the gas saved by the unnecessary EVM opcodes that appear in the simplification rules and the gas that corresponds to the saved push instructions involved in the rules. **G'** shows the total gas optimized when the corresponding EVM code of the contract is analyzed using GASOL. $\mathbf{G}_R$ contains the amount of gas optimized due to the application of the simplification rules (and the corresponding percentage). The difference between **G'** and $\mathbf{G}_R$ is the gas saved by the use of the Max-SMT solver. Finally, $\mathbf{T}_1$, $\mathbf{T}_2$ and $\mathbf{T}_3$ represent the number of simplification rules of each type identified by GASOL for each contract. As shown in Table 2.4, the amount of gas saved in the optimized versions due to the application of the simplification rules increases. However, in the worst case, a 33.6% of the gas optimized corresponds to the application of the SMT engine. If we compute the mean of the gas optimized in each version, we obtain that if we apply GASOL on the non-optimized versions of the contracts, a 19.8% of the gas gains corresponds to the application of the simplification rules, a 52% for the no-yul-optimized versions and a 53.5% for the full-optimized versions.

If we focus on the rules T_2 and T_3 identified by GASOL in any of the optimized versions of the Solidity files analyzed, we find that in most of the cases: (i) the patterns involved in the corresponding simplification rules do not appear explicitly in the original sequence of EVM instructions, and (ii) there are several rules involved in the same pattern. Importantly, GASOL is able to identify them thanks to the SFS and its symbolic execution engine that goes beyond a syntantic application of rules. For instance, GASOL identifies the following rules:

- In the sequence of EVM bytecodes `PUSH 0 DUP2 MLOAD DUP2 LT ISZERO PUSHTAG`, the rule (33) in Figure A.1 (`ISZERO(LT(0,X))` ≡ `ISZERO(X)`) has been applied. Thanks to the SFS, GASOL is able to realize that the first argument of the `LT` instruction is 0, that has been pushed 4 instructions before.

  Similarly, in the sequence `PUSH64 MLOAD PUSH64 MLOAD DUP1 SWAP2 SUB`, GASOL identifies that both arguments of the `SUB` instruction are the same in the SFS representation.

- In the sequence `CALLER PUSH 1 PUSH 1 PUSH 160 SHL SUB AND`, GASOL computes the operations over constants (`SHL` and `SUB`) to be able to apply the rule (62) in Figure A.1 (`AND(CALLER,`2^{160}`-1)`≡`CALLER`.

  Something similar happens when the sequence `DIV PUSH 1 PUSH 1 PUSH 160 SHL SUB AND PUSH 1 PUSH 1 PUSH 160 SHL SUB AND` is analyzed by GASOL. It evaluates the constant operations, and then it applies the rules (11) and (57) shown in Figure A.1. Without the SFS representation, GASOL would not be able to apply these simplification rules.

We also found out that there are some rules that, in spite of matching some pattern are not applied by the solc optimizer, such as the following ones: `EQ ISZERO ISZERO`, `PUSH 0 DUP2 EQ`, `PUSH 0 DUP4 GT` and `DUP3 PUSH0 ADD`.

File	Non-optimized					No-YUL-optimized					Full-optimized				
	G'	G_R	T_1	T_2	T_3	G'	G_R	T_1	T_2	T_3	G'	G_R	T_1	T_2	T_3
0	10136.0	1962 (19.36%)	88	92	68	2860.0	1863 (65.14%)	191	0	10	2929.0	1977 (67.5%)	203	0	10
1	9976.0	1968 (19.73%)	91	90	66	2819.0	1830 (64.92%)	187	0	10	2889.0	1926 (66.67%)	197	0	10
2	8346.0	1496 (17.92%)	50	95	67	3466.0	1626 (46.91%)	176	0	2	3417.0	1662 (48.64%)	180	0	2
3	6932.0	1273 (18.36%)	37	100	50	3602.0	1770 (49.14%)	190	0	2	3680.0	1866 (50.71%)	200	0	2
4	923.0	114 (12.35%)	2	14	1	444.0	141 (31.76%)	14	0	1	441.0	180 (40.82%)	18	0	1
5	4154.0	1042 (25.08%)	46	62	27	1242.0	1022 (82.29%)	106	1	3	1261.0	1070 (84.85%)	112	1	3
6	12126.0	2065 (17.03%)	84	122	49	5854.0	1891 (32.3%)	193	4	8	5933.0	1942 (32.73%)	198	4	9
7	33939.0	8187 (24.12%)	488	280	163	6634.0	3945 (59.47%)	413	0	11	6719.0	4074 (60.63%)	427	0	11
8	9684.0	2211 (22.83%)	96	118	68	3787.0	1519 (40.11%)	151	6	3	3872.0	1627 (42.02%)	163	6	3
9	2273.0	400 (17.6%)	13	28	15	1061.0	563 (53.06%)	54	1	10	1137.0	591 (51.98%)	58	0	10
10	9489.0	1709 (18.01%)	62	87	86	3833.0	1728 (45.08%)	185	0	3	3848.0	1728 (44.91%)	185	0	3
11	5088.0	968 (19.03%)	39	67	18	1852.0	1164 (62.85%)	119	0	10	1835.0	1149 (62.62%)	120	0	7
12	1991.0	273 (13.71%)	4	22	19	1225.0	423 (34.53%)	46	0	1	1185.0	423 (35.7%)	46	0	1
13	2749.0	460 (16.73%)	11	33	26	1332.0	898 (67.42%)	85	2	18	1328.0	954 (71.84%)	93	0	18
14	10129.0	1907 (18.83%)	87	95	62	2893.0	1788 (61.8%)	186	0	6	2988.0	1884 (63.05%)	196	0	6
15	3842.0	546 (14.21%)	8	44	38	2302.0	846 (36.75%)	92	0	2	2249.0	846 (37.62%)	92	0	2
16	9968.0	1968 (19.74%)	91	90	66	2803.0	1821 (64.97%)	186	0	10	2873.0	1917 (66.72%)	196	0	10
17	9885.0	1882 (19.04%)	61	140	54	5185.0	3015 (58.15%)	313	2	5	5281.0	3126 (59.19%)	324	2	5
18	3819.0	614 (16.08%)	17	36	36	1847.0	729 (39.47%)	79	0	1	1877.0	765 (40.76%)	83	0	1
19	5050.0	1525 (30.2%)	64	73	57	1738.0	846 (48.68%)	75	2	10	1730.0	885 (51.16%)	81	0	9
20	1969.0	346 (17.57%)	11	24	19	1036.0	766 (73.94%)	74	2	14	1062.0	822 (77.4%)	82	0	14
21	0.0	0 (0%)	0	0	0	6.0	0 (0.0%)	0	0	0	6.0	0 (0.0%)	0	0	0
22	12047.0	2365 (19.63%)	93	109	108	4320.0	2226 (51.53%)	230	0	11	4210.0	2256 (53.59%)	235	0	8
23	5605.0	940 (16.77%)	28	67	44	2964.0	1290 (43.52%)	140	0	2	2905.0	1338 (46.06%)	144	0	4
24	9974.0	1977 (19.82%)	91	91	66	2819.0	1830 (64.92%)	187	0	10	2889.0	1926 (66.67%)	197	0	10
25	4196.0	621 (14.8%)	10	51	41	2570.0	972 (37.82%)	106	0	2	2482.0	972 (39.16%)	106	0	2
26	11646.0	2256 (19.37%)	109	86	80	3560.0	2124 (59.66%)	222	0	8	3637.0	2220 (61.04%)	232	0	8
27	4012.0	632 (15.75%)	13	53	34	2220.0	984 (44.32%)	105	0	2	2200.0	993 (45.14%)	105	0	2
28	3848.0	546 (14.19%)	8	44	38	2317.0	846 (36.51%)	92	0	2	2257.0	846 (37.48%)	92	0	2
29	6563.0	1293 (19.7%)	40	99	34	3472.0	2160 (62.21%)	224	2	2	3639.0	2262 (62.16%)	234	2	2

Table 2.4: Rules identified by GASOL per contract and gas savings due to the rules.

Chapter 3

Verification in Etherscan

In this chapter, we describe the approach agreed with the YUL team to verify that the bytecode uploaded to Etherscan has been generated using GASOL. At first, it is not possible to verify the bytecode by only providing the source code to Etherscan as the optimization process of GASOL is not guaranteed to be deterministic. This is due to the usage of SMT solvers as black-boxes to find the optimized code, combined with the fact that there may be multiple solutions with the same optimization gains.

To overcome this problem, a *log* file is introduced to record the generated solution. Given the source code, the log allows generating exactly the same optimized bytecode. From this point, the verification process would follow the same idea as Etherscan. The *log* file is associated to the source code: the optimized bytecode cannot be reconstructed using only the information from the log file. Although the optimization process is not deterministic, the Max-SMT problem always uses the same notation for variables and constraints generated from a concrete sub-block. The generation of the log file relies on this fact and stores the optimized sub-block in terms of the numerical values associated to each instruction in the Max-SMT problem. The conversion between the numerical value and the associated instruction is determined from each SFS, and therefore, from the source code.

This representation is well-suited: it is compact and it allows replicating the whole optimization process skipping only the step in which GASOL spends most time: solving the Max-SMT problem. Besides, it allows applying further verification checks. Namely, in order to verify this *log* file is indeed correct and has not been modified, we generate an SMT problem that comprises the constraints from all *sub-blocks* and also includes constraints to represent the optimized solution. This way, the SMT solver tests the optimized bytecode is indeed equivalent to the bytecode generated from the source code. This introduces some overhead (experimental figures follow next) as the SMT solvers have to deal with thousands of constraints. Another approach we might implement in the future is to generate a *hash* from the log file and store it in this file. This way, it could be checked whether it has been modified or not immediately.

3.1 Experimental Results

Figure 3.1 shows the overhead of the verification approach for the 30 Solidity files analyzed, both in terms of the size of log files and the time spent to verify the solutions contained in these files are indeed correct. The results are split into two columns according to the settings No-YUL+GASOL and YUL+GASOL described in Section 2.2. **S** represents the size in bytes of the log files generated after performing the optimization in GASOL. The blocks contained in these files correspond to columns **O** and **B** in Figure 2.2. **T** represents the time in seconds needed to perform the SMT verification check for each file. This verification step considers all optimized blocks from the log file.

It can be observed that most files take around 1 second to be verified, reaching its maximum time at less than 2 seconds. The log size associated to most files is around 20 kB, being in every case less than 50 kB. This size can be further reduced if the log file is converted into a binary file, which could be considered for later stages of the project. Nevertheless, we believe that these number show the approach is light enough to be included in the Etherscan verification process.

File	No-YUL+GASOL		YUL+GASOL	
	S	T	S	T
#0	19503	0.7	20101	0.7
#1	18726	0.66	19100	0.69
#2	20570	0.77	20650	0.72
#3	22295	0.82	22911	0.83
#4	3358	0.13	3359	0.11
#5	10240	0.3	10343	0.29
#6	25569	0.89	27126	0.95
#7	47918	1.74	48256	1.76
#8	18087	0.67	19018	0.68
#9	12263	0.39	10206	0.28
#10	18266	0.65	18086	0.64
#11	26674	1.18	15407	0.46
#12	6789	0.27	6466	0.24
#13	15783	0.55	12906	0.38
#14	20101	0.7	20392	0.7
#15	14075	0.53	13500	0.47
#16	19041	0.65	19412	0.67
#17	34449	1.1	34041	1.13
#18	10227	0.39	9879	0.32
#19	9273	0.5	10752	0.61
#20	10782	0.3	10146	0.3
#21	113	0.0	113	0.0
#22	28612	0.96	28018	0.84
#23	19638	0.68	18518	0.59
#24	19499	0.67	19869	0.68
#25	15709	0.59	14674	0.52
#26	22692	0.73	22987	0.72
#27	11037	0.4	11271	0.39
#28	12948	0.52	12256	0.46
#29	23320	0.75	23075	0.72

Table 3.1: Log size in bytes and time spent in verification

Chapter 4

Installation and Usage of GASOL

GASOL is open-source and it is available at https://github.com/costa-group/gasol-optimizer.

4.1 Installation (Ubuntu)

GASOL is implemented in Python and runs Python3. It does not need any additional library. In order to install it, run one of the following commands:

- **Download GASOL source code.**

  (a) Clone the GitHub repository in the desired directory of you machine using the command

  ```
  git clone https://github.com/costa-group/gasol-optimizer.git.
  ```

  (b) Download a zip folder with the source code that is available here and decompress it in the desired directory by executing

  ```
  unzip gasol-optimizer-main.zip
  ```

4.2 Usage

GASOL can optimize either a single sequence of assembly EVM instructions (block) or all basic blocks obtained from an assembly (in what follows asm) json that has been generated from a Solidity file by executing the following command:

```
solc --combined-json asm solidity_file.sol
```

By default, GASOL uses OptiMathSAT [3] as SMT solver with a timeout of 10s per block.

- In order to execute GASOL on a asm json file, run the following command from the root directory of the repository:

  ```
  ./gasol_asm.py asmjson_filename
  ```

where asmjson_filename is the name of the file where the asm json is stored. A set of asm json files to test the prototype is available here. GASOL will analyze all basic blocks of the provided smart contracts. Note that it may take some time to finish the execution.

As a result of the optimization, another asm json with the optimized blocks is generated. By default, this file is stored in the same folder from which the executable was invoked. Its file name corresponds to the initial asm json file name after adding the suffix *_optimized*. This output file can be specified using −o flag:

```
./gasol_asm.py asmjson_filename −o solution_filename
```

- In order to execute GASOL on a basic block, run the following command from the root directory of the repository:

  ```
  ./gasol_asm.py block_filename −bl
  ```

  where block_filename is the name of the file where the basic block is stored. Note that the basic block has to be stored as a sequence of EVM instructions separated by blanks. See the examples here for more details. In this case, the optimized output is displayed in the console.

- The generation of the log file is enabled by setting the flag −−generate−log when invoking GASOL on an asm json file:

  ```
  ./gasol−asm.py asmjson_filename −−generate−log
  ```

  After performing the optimization process, the log file /tmp/gasol/*asmjson_filename*.log is created. This log file can be used to generate exactly the same optimized bytecode from the source file by executing the following command:

  ```
  ./gasol−asm.py asmjson_filename −optimize−gasol−from−log−file /tmp/gasol/
  asmjson_filename.log
  ```

  The output follows exactly the same convention depicted in Section 4.2.

Bibliography

[1] Simplification rules from Solidity compiler, 2021. `https://github.com/ethereum/solidity/blob/develop/libevmasm/RuleList.h`.

[2] 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.

[3] 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.

[4] 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.

Appendix A

Simplification rules

This section contains the complete list of simplification rules integrated in GASOL (we have omitted the commutative cases).

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

Figure A.1: Simplification rules of GASOL