Don't Run on Fumes — Parametric Gas Bounds for Smart Contracts

Elvira Albert[a,b], Jesús Correas[b], Pablo Gordillo[b], Guillermo Román-Díez[c], Albert Rubio[a,b]

[a] *Instituto de Tecnología del Conocimiento, Spain*
[b] *Complutense University of Madrid, Spain*
[c] *Universidad Politécnica de Madrid, Spain*

Abstract

Gas is a measurement unit of the computational effort that it will take to execute every single replicated operation that takes part in the Ethereum blockchain platform. If a transaction exceeds the amount of gas allotted by the user (known as gas limit), an *out-of-gas* exception is raised and its execution is interrupted. One of the main open problems in the analysis of Ethereum smart contracts is the inference of *sound* bounds on their gas consumption.

We present, to the best of our knowledge, the first static analysis that is able to infer sound *parametric* (i.e., non-constant) gas bounds for smart contracts. The inferred bounds can be parametric on the sizes of the input parameters for the functions, but also they can be parametric on the contract state, or blockchain data. Our gas analysis is developed at EVM bytecode level, in which Ethereum gas model is defined.

Our analysis is implemented in a tool named GASTAP, Gas-Aware Smart contracT Analysis Platform, which takes as input a smart contract and automatically infers *sound* gas upper-bounds for its public functions. GASTAP has been applied over 318,093 functions fetched from the Ethereum blockchain, and succeeded to obtain gas bounds for 90.24% of them.

Keywords: Smart contracts, Resource analysis, Static analysis, Decompilation

1. Introduction

In the Ethereum consensus protocol, every operation on a replicated blockchain state, which can be performed in a transactional manner by executing a *smart contract* code, costs a certain amount of *gas* [1]. Gas has a monetary value in *Ether*, Ethereum's currency, and it is paid by a transaction-proposing party. Computations (initiated by a protocol client invoking a smart contract) that

Email addresses: elvira@sip.ucm.es (Elvira Albert), jcorreas@ucm.es (Jesús Correas), pabgordi@ucm.es (Pablo Gordillo), guillermo.roman@upm.es (Guillermo Román-Díez), alberu04@ucm.es (Albert Rubio)

require *more computational or storage resources*, cost more gas than those that require fewer resources. As regards storage, the Ethereum Virtual Machine (EVM) has three areas where it can store items: (a) the *storage* is where all *contract state* variables reside, every contract has its own storage and it is persistent between external function calls (transactions) and quite expensive to use; (b) the *memory* is used to hold temporary values, and it is erased between transactions and thus is cheaper to use; (c) the *stack* is used to carry out operations and it is free to use, but can only hold a limited amount of values.

The rationale behind the resource-aware smart contract semantics, instrumented with gas consumption, is three-fold. First, paying for gas at the moment of proposing the transaction does not allow the emitter to waste other parties' (aka *miners*) computational power by requiring them to perform a lot of worthless intensive work. Second, gas fees disincentivize users to consume too much of replicated *storage*, which is a valuable resource in a blockchain-based consensus system. Finally, such a semantics puts a cap on the number of computations that a transaction can execute, hence prevents attacks based on non-terminating executions (which could otherwise, *e.g.*, make all miners loop forever).

The gas-instrumented operational semantics of EVM has introduced novel challenges *wrt.* sound static reasoning about resource consumption, correctness, and security of replicated computations: *(i)* While the EVM specification [1] provides the precise gas consumption of the low-level operations, most of the smart contracts are written in high-level languages, such as Solidity [2] or Vyper [3]. The translation of the high-level language constructs to the low-level ones makes static estimation of runtime gas bounds challenging (as we will see throughout this paper), and is implemented in an *ad-hoc* way by state-of-the art compilers, which are only able to give constant gas bounds, or return ∞ otherwise. *(ii)* As noted in the recent study by Gretch *et al.* [4, 5], in general it is dangerous for a smart contract to make its gas consumption dependent on the size of the data it stores (*i.e.*, the *contract state*), as well as on the size of its functions inputs, or of the current state of the blockchain. However, according to our experiments, almost 10% of the contract functions we have analyzed feature such dependencies. Note that there are a number of situations that, without having iterative code in the Solidity code, the Solidity compiler produces a loop at EVM bytecode level. These "hidden" loops might come from functions that receive as parameters arrays or strings, the size of the message data, or the length of data structures in storage. Let us show an example:

```
1  contract HiddenLoops {
2    uint [ ] data;
3    function f (uint [ ] memory values) public {   // loop1
4      delete data;                                 // loop2
5      for (uint i = 0; i < values.length; i++) { // loop3
6        data.push(values[i]);
7      }
8    }
9  }
```

The gas consumed by function f does not only depend on the length of values as it can be expected by looking at the source code. The EVM program obtained from this Solidity program includes two *hidden* loops: one that copies the contents of values in memory (loop_1); and another one that traverses all elements stored in data for setting them to 0 (loop_2). The use of resource analysis techniques at the level of the EVM allows us to infer the following accurate and sound upper-bound expression that accumulates the costs of these two hidden loops: $5768 + \underbrace{3 * \mathsf{values}}_{loop_1} + \underbrace{5057 * \mathsf{data}}_{loop_2} + \underbrace{40451 * \mathsf{values}}_{loop_3}$. The inability to soundly estimate the cost, and the lack of analysis tools, might lead to design mistakes, which make a contract unsafe to run or prone to exploits. For instance, a contract whose state size exceeds a certain limit, can be made forever *stuck*, not being able to progress. Those vulnerabilities have been recognized before, but only discovered by means of unsound, pattern-based analysis of control-flow graphs [4].

In this article, we address these challenges in a principled way by developing a new static analysis that is able to infer *parametric* gas bounds for smart contracts. The upper-bounds we infer are given in terms of the sizes of the input parameters of the functions, the contract state, and/or on the blockchain data that the gas consumption depends upon (*e.g.*, on the *Ether* value effective at the moment when the corresponding transaction takes place). The inference of gas requires complex transformation and analysis processes on the code that include: (1) the construction of the EVM control-flow graphs and the decompilation of low-level EVM bytecode to a higher-level rule-based representation of the program; (2) the special treatment of EVM data types (e.g., strings and bytes are challenging to deal with) and its storage and memory model; (3) the definition of a static cost-model that captures the gas consumed by the program and that can be plugged within state-of-the-art cost analysis tools; (4) the implementation effort to take advantage of off-the-shelf analysis tools that are able to compute parametric bounds for the input program.

A challenging aspect in the definition of the gas bounds analysis has been the approximation of the EVM gas model (which is formally specified in [1]). This is because the EVM gas model is highly complex and unconventional. The gas consumption of each instruction has two parts: (i) the *memory gas cost*, if the instruction accesses a location in memory which is beyond the previously accessed locations (known as *active* memory [1]), it pays a gas proportional to the distance of the accessed location. (ii) The second part, the *opcode gas cost*, is related to the bytecode instruction itself. This is quite challenging to infer as, somewhat counter-intuitively, it is not always a constant, but might depend in some cases on the current state of a contract and the blockchain. Our analysis is able to soundly approximate both components: the former is estimated by means of an instance of a *peak resource analysis* [6, 7] and the latter using a *parametric* cost model within standard resource analysis [8].

The analysis is implemented in a tool named GASTAP, a *Gas-Aware Smart contracT Analysis Platform*, which is, to the best of our knowledge, the first automatic and sound gas analyzer for smart contracts. GASTAP takes as input

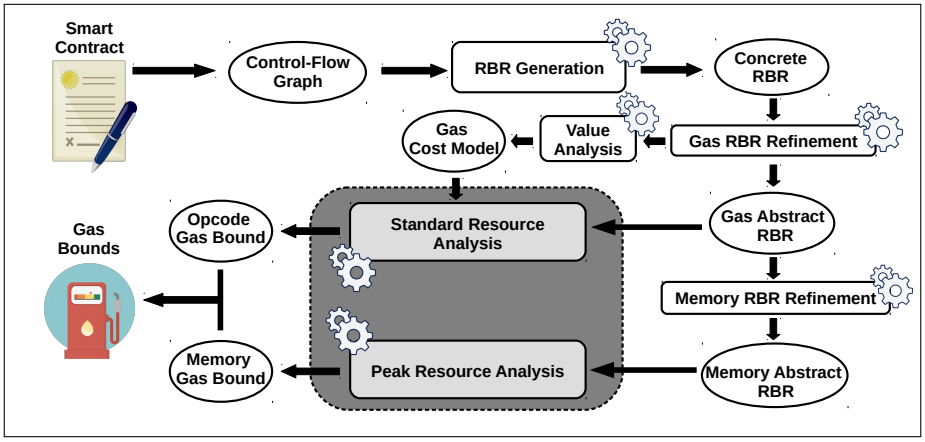


Figure 1: Architecture of GASTAP. White boxes are components implemented by us. Gray boxes are off-the-shelf tools.

a smart contract provided in Solidity source code [2], or in low-level (possibly decompiled [9]) EVM code, and automatically infers upper-bounds on the gas consumption for each of its public functions. Figure 1 provides an overview of the architecture of our analyzer and its different components will be introduced throughout the paper. GASTAP has a wide range of applications for contract developers, attackers and owners, including the detection of vulnerabilities, debugging and verification/certification of gas usage. For contract developers and owners, a precise resource analyzer makes it possible to answer the following query about a specific smart contract: "what is the amount of gas necessary to *safely* (*i.e.*, without an out-of-gas exception) reach a certain execution point in the contract code, or to execute a function?" This can be used for debugging, verifying/certifying a safe amount of gas for running, as well as ensuring progress conditions. Besides, GASTAP allows calculating the safe amount of gas that one must provide to an external data source (*e.g.*, contracts using the popular Oraclize service [10]) in order to enable a *successful asynchronous callback* in a forthcoming separate response transaction. On the other hand, a rational attacker, can use a resource analysis in order to estimate how much *Ether* (in gas), they have to pour into a contract in order to execute the Denial-of-Service attack. We note that such an attack may, however, be economically impractical [4].

1.1. Summary of Contributions

In summary, this article makes the following main contributions:

1. *Rule-based representation of EVM code.* The starting point for our analysis is EVM bytecode from which a control flow graph (CFG) is obtained by our tool. This has required the implementation of an *address analysis* that

figures out all possible jumping addresses. From the CFG, the box *RBR generation* of Figure 1 provides the decompilation from EVM bytecode to a high-level Rule-Based Representation (RBR) that enables subsequent static analysis on the EVM code. Our decompiled programs represent explicitly the local and state variables, the operand stack, and blockchain data, by means of rule parameters.

2. *Handling EVM data and storage/memory model.* In Section 4, we will describe the box *GAS RBR refinement* of Figure 1 that by means of a number of program transformations in the RBR allows us to handle the specific features of EVM programs. In particular, we are able to handle strings and bytes —what is fundamental to achieve a precise gas analysis of EVM programs. The transformation in this case consists in retrieving the sizes of these data types and making them visible within the variables of the RBR. Another transformation is performed in order to represent the storage and memory accesses within the RBR program so that we can infer information on them in the subsequent analysis.

3. *Opcode gas bounds.* In order to be able to apply a cost analysis framework to infer gas upper-bounds, we need to provide the definition of the *gas operations cost model* which (over-)approximates the gas usage of every EVM bytecode instruction. Over-approximation is needed because the gas model of EVM [1] is declared for concrete executions in which the state is fully instantiated (e.g., the gas cost of `EXP` varies logarithmically depending on the value of operand on which it is applied, the gas cost of `SSTORE` varies depending on whether the value to be stored is equals to zero or not). However, in order to statically infer gas bounds, we need to provide a static, parametric, gas model that soundly defines the gas cost for any concrete state that might arise during the execution.

4. *Memory gas bounds.* The memory gas costs are computed separately in our framework by using a non-standard cost analysis. As we will explain, the inference of memory gas bounds boils down to inferring the address of the highest slot of memory accessed and, then, instantiating this value within the formula for the memory gas cost. We propose to infer this highest slot by means of a peak resource analysis [6]. As we will describe in Section 5.2, this can be done by means of a transformation within the RBR which annotates –using *acquire* declarations– the memory address accessed. The peak analysis then infers the peak value of all the *acquire* annotations, hence obtaining the highest slot accessed that we need.

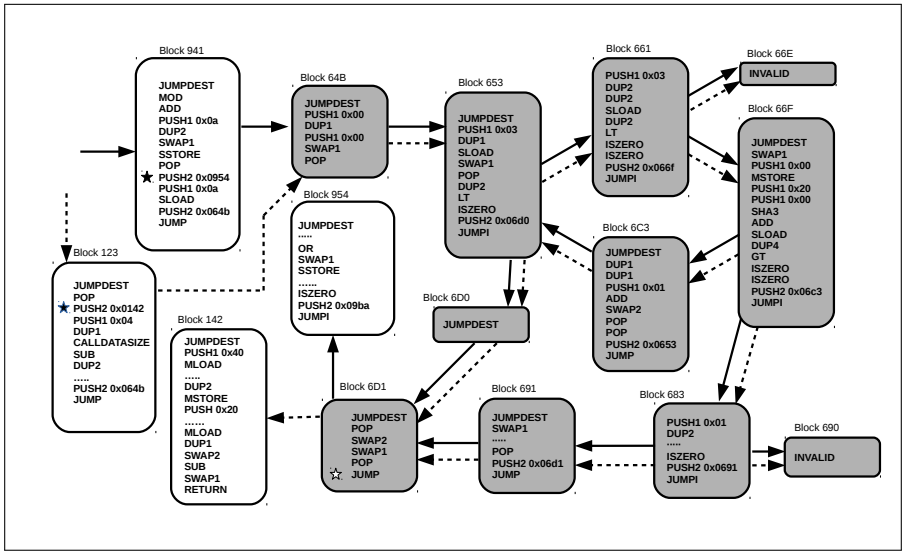
5. *Implementation and experimental evaluation.* Finally, Section 6 describes our experimental evaluation: we have analyzed more than 34,000 real smart contracts, and have succeeded to analyze 318,093 public functions and inferred gas bounds for 90.24% of them in 407.5 hours. GASTAP can be used from a web interface at `https://costa.fdi.ucm.es/gastap`.

Basically, the first contribution (item 1 above) enables using a standard resource analyzer but it requires solving the unique problems presented by smart contracts that are the contributions listed in item 2 (i.e., handling its storage/memory model and data), item 3 (providing a static definition of the gas operations cost model), and item 4 (developing new techniques to infer the memory gas costs).

A preliminary version of the decompilation phase in Section 3 appeared as a tool demonstration paper in the proceedings of ATVA'18 [11] and a first description of the basic components of the gas analyzer appeared in the proceedings of VECoS'19 [12]. This article provides the formal basis for these tools and formally describes all the steps carried out in the analysis: namely we formalize a rule-based representation and a new gas-aware semantics that can be used for multiple purposes beyond gas analysis; as another novelty w.r.t. the previous publications, we present the extensions in item 2 above to handle the particularities of EVM code within static analysis; and we give a formal definition of the gas cost model (item 3) and the computation of the memory gas bounds by relying on existing cost analysis techniques. Beyond the formal contributions, we have also improved the experimental evaluation of [11] and [12] significantly, since the former versions of our tool were relying on [13] to build the CFGs (that could be incomplete and was failing in many examples) and we have now fully implemented the CFG generation. Besides, we have applied our tool to more contracts (namely to the top-valued 300 contracts whose Solidity code was available) and assessed the accuracy of the upper bounds obtained for the analyzed functions by comparing them with the actual gas cost of real transactions.

2. Preliminaries: *Stack-Sensitive* Control Flow Graphs

The EVM language is a simple stack-based language with words of 256 bits with a local volatile memory that behaves as a simple word-addressed array of bytes, and a persistent storage that is part of the blockchain state. A more detailed description of the language and the complete set of operation codes can be found in [1]. In this section, we only describe the (Stack-Sensitive) CFGs from which our contribution starts. As usual, the computation of the CFG is based on the notion of *block*. In general [14], given a program P, a *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. In the case of EVM programs, blocks are split by using jumping instructions, jump destinations and ending instructions like `RETURN`, `INVALID` or `REVERT`. One significant difference between the EVM and other virtual machine languages (like Java Bytecode or CLI for .Net programs) is the use of the stack for saving the jump addresses instead of having them explicit in the code of the jumping instructions. In EVM, instructions `JUMP` and `JUMPI` will jump, unconditionally and conditionally respectively, to the program counter stored in the top of the execution stack. The difficulty is that the address might have been stored in a different block, as we will show in the example below. This happens for




```
1  contract EthereumPot {
2    address[] public addresses;
3    address public winnerAddress;
4    uint[] public slots;
5    function findWinner (uint random) constant returns (address winner) {
6      for (uint i = 0; i < slots.length; i++) {
7        if (random <= slots[i]) {
8          return addresses[i];
9        }
10     }
11   }
12   // Other functions
13 }
```

Figure 2: Excerpt of Solidity code for `EthereumPot` contract and its *S-CFG*

instance when a public function is invoked privately from other methods of the same contract, the returning program counter is introduced by the invokers at different program points and it will be used in a unique `JUMP` instruction when the invoked method finishes in order to *return* to the particular caller that invoked that function. This feature of the EVM requires, in order to obtain the control flow graph of the program, to keep track of the information stored in the stack and compute the different stack sizes with which the blocks can be reached.

Following [15, 4, 16], in order to precisely infer the possible addresses at jumping program points, we need a *context-sensitive* static analysis that analyzes separately all blocks for each possible stack that can reach them (only considering the addresses stored in the stack). This context-sensitive analysis

information is used to compute a *stack-sensitive* control flow graph (*S-CFG*) of the form $S\text{-}CFG = \langle V, E \rangle$ in that, for producing the set of vertices V, we replicate each block for each different stack state that could be used for invoking it (e.g. gray nodes in Figure 2 are cloned in the *S-CFG*) and the corresponding edges E are replicated according to the replicated blocks. Each block in V has the form $block_{i:id}$, where i is the program counter of the first instruction of the block and id is a numeric identifier for the clone. For those blocks which are not cloned, we use 0 as identifier and omit it when it is clear from the context. We also use function $getSize(pc, id)$, which returns the size of the stack at program point pc for the clone identified with id. A precise definition of this phase and the assumed function $getSize$ is available in an online technical report [17]. There are some cases in which this analysis is not able to generate an *S-CFG*, mostly due to the use of recursion and higher-order programming in Solidity. Importantly, the analysis is incomplete but sound: if an *S-CFG* is generated for a given EVM program P, then for each execution trace of P there exists a corresponding path in the *S-CFG* that correctly represents it (proofs of soundness the *S-CFG*-generation can be found at the above cited technical report [17]).

Example 1. *In order to describe our techniques, we use as running example method* `findWinner` *extracted from the* `EthereumPot` *contract [18] that implements a lottery system. Figure 2 shows the blocks (nodes) obtained for* `findWinner` *and their corresponding jump invocations. Solid and dashed edges represent the two possible execution paths depending on the entry block: solid edges represent the path that starts from block* `941` *and dashed edges the path that starts from* `123`*. Interestingly, these two paths end in block* `6D1`*, where we have a* `JUMP` *(marked with ☆) that takes the return address from function* `findWinner`*. If* `findWinner` *is publicly invoked, it jumps to address* `142` *(pushed at block* `123` *at ⋆) and if it is invoked from function* `__callback` *it jumps to* `954` *(pushed at block* `941` *at ⋆). The S-CFG includes non-replicated nodes for those blocks that only receive one possible stack state (white nodes in Figure 2). However, the nodes that could be reached by two different stack states (gray nodes in Figure 2) are cloned in our S-CFG. For example, the S-CFG includes* $block_{653:1}$*,* $block_{653:2}$*,* $block_{661:1}$ *and* $block_{661:2}$*. The edges also consider the different replicas of the nodes and we have* $block_{653:1} \rightarrow block_{661:1}$ *and* $block_{653:2} \dashrightarrow block_{661:2}$*.*

■

3. From the S-CFG to a High-Level Rule-Based Representation

This section presents the process of obtaining a high-level *rule-based representation* (RBR) from the *S-CFG* described in Section 2. This part of the analysis, has been implemented as a standalone tool, EthIR [11], to facilitate its integration into other tools and corresponds to the *RBR generation* box in Figure 1. Given a *S-CFG*, it provides a set of rules that contain a high-level representation of all bytecode instructions in the block (*e.g.*, a `PUSH` operation is represented as an assignment to a stack variable). The rules have as parameters an explicit representation of the stack as well as of the memory, storage and

blockchain data. Conditional branching in the CFG is represented by means of guarded rules that contain mutually exclusive boolean guards. The grammar of the RBR language into which the EVM is translated is as follows:

$$
\begin{array}{lcl}
RBR & \rightarrow & Rule \;\; RBR \mid \epsilon \\
Rule & \rightarrow & rulename\ (\bar{s}, \texttt{MEM}, \texttt{STO}, \texttt{BLC}) \Rightarrow Guard\ \text{"|"}\ Instr\ \ (Call \mid \epsilon) \\
Instr & \rightarrow & S, Instr \mid \epsilon \\
S & \rightarrow & Var = Exp \mid \texttt{nop}(EVMInstr) \\
Exp & \rightarrow & num \mid Var \mid \texttt{BLC}(Blcvar) \mid s_i + s_j \mid s_i - s_j \mid s_i * s_j \mid s_i / s_j \mid s_i \,\%\, s_j \mid \\
 & & s_i^{s_j} \mid eq(s_i, s_j) \mid lt(s_i, s_j) \mid gt(s_i, s_j) \mid \\
 & & and(s_i, s_j) \mid or(s_i, s_j) \mid xor(s_i, s_j) \mid not(s_i) \\
Call & \rightarrow & call(rulename\ (\bar{s}, \texttt{MEM}, \texttt{STO}, \texttt{BLC})) \\
Guard & \rightarrow & s_i = 0 \mid s_i \neq 0 \\
Var & \rightarrow & s_i \mid \texttt{MEM}[s_i] \mid \texttt{STO}[s_i] \\
Blcvar & \rightarrow & \texttt{GAS} \mid \texttt{BALANCE} \mid \texttt{NUMBER} \mid \ldots
\end{array}
$$

We assume that all instructions $Instr$ in the RBR rules are of the form pp:b where pp is a unique program point identifier in the RBR program and b is either an assignment or a `nop` instruction. In what follows, we will ignore the program point unless it is required. Note that, the program point of the RBR does not correspond to the program counter of the original EVM program because some blocks have been replicated. The left hand side of RBR rules has the form $rulename\ (\bar{v})$, where $\bar{v}$ are parameters that can be of the following types:

1. *Stack variables* ($\bar{s}$): A relevant ingredient of the transformation is that the stack is flattened into explicit variables, *i.e.*, the different stack elements used by the instructions are modeled by using different variables $s_0, s_1, s_2, \ldots, s_{n-1}$, where s_{n-1} represents the top of the stack. The stack is explicitly received by the rules as parameters $s_0, s_1, \ldots, s_{n-1}$ and denoted as $\bar{s}$ and its size for any block $B_{i:id}$ is obtained from function $getSize(i, id)$. We use ST to refer to the set of all stack variables used in the program.

2. *Memory variables* (`MEM`): We use the array `MEM` (that has a size of 2^{256} elements, like the maximum memory size) to model the content of the local memory of the transaction. The memory position to be accessed is pushed in the stack, hence accesses to the array will use as index a stack variable s_i, i.e., $\texttt{MEM}[s_i]$ accesses the memory location stored in s_i.

3. *State variables* (`STO`): We also model the contract state variables by means of an array `STO` (again with 2^{256} elements, as the maximum storage size). As for memory, it is indexed using the stack variables as follows $\texttt{STO}[s_i]$, where stack variable s_i contains the address involved in the operation.

4. *Blockchain data* (`BLC`): We model blockchain data using a mapping $\texttt{BLC}(var)$, where var is the name of a blockchain variable that represents the environmental or blockchain data, as *e.g.* $\texttt{BLC}(\texttt{GAS})$ or $\texttt{BLC}(\texttt{NUMBER})$. All these data are accessed through dedicated opcodes, which may consume some offsets

of the stack and normally place the result on top of the stack (some of them, like `CALLDATACOPY`, can besides store information in the local memory). We will extend the notation to refer to additional parameters for some environmental information, as *e.g.* $\texttt{BLC}(\texttt{CALLDATALOAD}, s_{n-1})$ to refer to the access to the word of the message call input data pointed by s_{n-1}. We use BLV to refer to the set of names of blockchain variables.

3.1. RBR generation

The starting point of the RBR generation is the CFG for the bytecode as described in Section 2. Intuitively, for each block in the CFG, we produce an RBR rule and the edges in the CFG induce the invocations between the different rules. Besides, each bytecode is transformed into a corresponding high-level instruction and an associated `nop` storing the original EVM operation. EVM instructions are kept in the RBR so as to precisely compute the gas cost of executing the original EVM instructions. The following definition formalizes the transformation of each EVM bytecode instruction into RBR instructions:

Definition 1 (RBR instructions)**.** *Given an EVM instruction of the form* $b_{pc} \in B_{i:id}$ *where* b *is the EVM bytecode and* pc *is the program counter in the EVM program, we define function* $\phi(b_{pc}, id)$ *as follows:*

$$\phi(b_{pc}, id) = \phi'(b_{pc}, n), \texttt{nop}(b_{pc})$$

where $n = getSize(pc, id)$ *and* $\phi'(b, n)$ *is defined for some representative EVM instructions as follows:*

b	$\phi'(b, n)$
JUMP|*JUMPI*|*JUMPDEST*	ϵ
PUSHx v	$s_n = v$
DUPx	$s_n = s_{n-x}$
SWAPx	$s_n = s_{n-1}$,
	$s_{n-1} = s_{n-1-x}$,
	$s_{n-1-x} = s_n$
ADD|*SUB*|*MUL*|*DIV*	$s_{n-2} = s_{n-1} + \mid - \mid * \mid \ / \ s_{n-2}$
GT	$s_{n-2} = gt(s_{n-1}, s_{n-2})$
LT	$s_{n-2} = lt(s_{n-1}, s_{n-2})$
EQ	$s_{n-2} = eq(s_{n-1}, s_{n-2})$
ISZERO	$s_{n-1} = eq(s_{n-1}, 0)$
MLOAD	$s_{n-1} = \mathit{MEM}[s_{n-1}]$
MSTORE	$\mathit{MEM}[s_{n-1}] = s_{n-2}$
SLOAD	$s_{n-1} = \mathit{STO}[s_{n-1}]$
SSTORE	$\mathit{STO}[s_{n-1}] = s_{n-2}$
CALLDATASIZE	$s_n = \mathit{BLC}(\mathit{CALLDATASIZE})$
ADDRESS	$s_n = \mathit{BLC}(\mathit{ADDRESS})$
...	

■

It can be seen that function ϕ translates each bytecode into two different components: (1) it applies function ϕ' to produce the corresponding high-level assignments between the RBR variables; and (2) it adds to the RBR the original bytecode instruction by simply wrapping it within a `nop` functor. Observe that function ϕ' transforms EVM instructions into assignments between explicit stack variables (see `PUSH`, `DUP` or `SWAP`), converts arithmetic operations into the same operations between elements in the stack, boolean operations are transformed into functions, or accesses to `MEM`, `STO` arrays or to map `BLC`. We now formalize the generation of the RBR rules using ϕ:

Definition 2 (RBR rules)**.** *Given the stack-sensitive control flow graph S-CFG* $= \langle V, E\rangle$ *of an EVM program* P*, and a block* $B_{i:id} \in V$ *with instructions* $B_{i:id} \equiv b_i, \ldots, b_j$ *in* P*, the generated rules are:*

$$
\begin{array}{ll}
(1) & \textit{if } b_j \equiv \texttt{JUMPI} \wedge (B_{i:id} \to B_{i_2:id_2}) \in E \wedge (B_{i:id} \to B_{i_3:id_3}) \in E \\
& \quad block\_i\_id(s_0, \ldots, s_{n-1}, \texttt{MEM}, \texttt{STO}, \texttt{BLC}) \quad \Rightarrow \\
& \qquad true \mid \\
& \qquad \phi(b_i, id), \ldots, \phi(b_j, id), \\
& \qquad call(jump\_i\_id(s_0, \ldots, s_{m+1}, \texttt{MEM}, \texttt{STO}, \texttt{BLC})) \\
& \quad jump\_i\_id(s_0, \ldots, s_{m+1}, \texttt{MEM}, \texttt{STO}, \texttt{BLC}) \quad \Rightarrow \\
& \qquad s_m \neq 0 \mid \\
& \qquad call(block\_i_2\_id_2(s_0, \ldots, s_{m-1}, \texttt{MEM}, \texttt{STO}, \texttt{BLC})) \\
& \quad jump\_i\_id(s_0, \ldots, s_{m+1}, \texttt{MEM}, \texttt{STO}, \texttt{BLC}) \quad \Rightarrow \\
& \qquad s_m = 0 \mid \\
& \qquad call(block\_i_3\_id_3(s_0, \ldots, s_{m-1}, \texttt{MEM}, \texttt{STO}, \texttt{BLC})) \\
(2) & \textit{if } b_j \not\equiv \texttt{JUMPI} \wedge (B_{i:id} \to B_{i_2:id_2}) \in E \\
& \quad block\_i\_id(s_0, \ldots, s_{n-1}, \texttt{MEM}, \texttt{STO}, \texttt{BLC}) \quad \Rightarrow \\
& \qquad true \mid \\
& \qquad \phi(b_i, id), \ldots, \phi(b_j, id), \\
& \qquad call(block\_i_2\_id_2(s_0, \ldots, s_{m-1}, \texttt{MEM}, \texttt{STO}, \texttt{BLC})) \\
(3) & \textit{if } b_j \not\equiv \texttt{JUMPI} \wedge \nexists\, (B_{i:id} \to \_) \in E \\
& \quad block\_i\_id(s_0, \ldots, s_{n-1}, \texttt{MEM}, \texttt{STO}, \texttt{BLC}) \quad \Rightarrow \\
& \qquad true \mid \\
& \qquad \phi(b_i, id), \ldots, \phi(b_j, id)
\end{array}
$$

where $n = getSize(i, id)$*,* $m = getSize(i_2, id_2)$*.*

■

In the RBR generation we distinguish three cases: (1) if the last bytecode in the block is a conditional jump (`JUMPI`) we produce two additional *guarded* rules (with a guard different from *true*) which represent the continuation when the condition holds, and when it does not; (2) if it is not a conditional jump and the *S-CFG* includes an outgoing edge from the corresponding block, we generate a single rule with a call to the continuation block; and (3) if the block ends in an instruction which terminates the execution, e.g. `REVERT` or `INVALID`, we just produce the rule without any call. Note that the number of stack variables n (m for the calls) is determined by the size of the stack at the first instruction of the rule by means of the function $getSize$ of Section 2.

Example 2. *Using Definition 2 to translate blocks* `64B` *and* `653` *in the CFG of Figure 2 we obtain the following rules:*

$block\_653\_1(s_0, \ldots, s_8, \texttt{MEM,STO,BLC}) \Rightarrow$
$true \mid$
$\texttt{nop(JUMPDEST)}$,
$s_9 = 3, \texttt{nop(PUSH1)}$,
$s_{10} = s_9, \texttt{nop(DUP1)}$,
$s_{10} = \texttt{STO}[s_{10}], \texttt{nop(SLOAD)}$,
$s_{11} = s_9, s_9 = s_{10}, s_{10} = s_{11}, \texttt{nop(SWAP1)}$,
$\texttt{nop(POP)}$,
$s_{10} = s_8, \texttt{nop(DUP2)}$,
$s_9 = lt(s_{10}, s_9), \texttt{nop(LT)}$,
$s_9 = eq(s_9, 0), \texttt{nop(ISZERO)}$,
$s_{10} = 0x6D0, \texttt{nop(PUSH2)}$,
$\texttt{nop(JUMPI)}$,
$call(jump\_653\_1(s_0, \ldots, s_{10}, \texttt{MEM,STO,BLC}))$

$jump\_653\_1(s_0, \ldots, s_{10}, \texttt{MEM,STO,BLC}) \Rightarrow$
$s_9 \neq 0 \mid$
$call(block\_6D0\_1(s_0, \ldots, s_8, \texttt{MEM,STO,BLC}))$

$jump\_653\_1(s_0, \ldots, s_{10}, \texttt{MEM,STO,BLC}) \Rightarrow$
$s_9 = 0 \mid$
$call(block\_661\_1(s_0, \ldots, s_8, \texttt{MEM,STO,BLC}))$

We can see that some operations produce assignments between the explicit stack variables (e.g. `PUSH1`, `DUP1`, *...), and that bytecode instructions that operate on storage or memory are transformed into assignments on the involved variables, e.g,* `SLOAD` *in* block_653_1 *is translated into* $s_{10} = \texttt{STO}[s_{10}]$. *For arithmetic operations, boolean operations, bit-wise operations, etc., the variables they operate on are also made explicit, e.g.,* `LT` *operates on* s_{10} *and* s_9 *and it is transformed into* $lt(s_{10}, s_9)$. *The guards of the rules also make explicit the corresponding condition and the stack variable involved on it (e.g.* $s_9 = 0$*). In this example we have included true guards, however, in the rest of the paper for brevity we will omit them. Observe that conditional jumps are implemented by means of two rules (e.g.* jump_653_1*) with the same name and mutually exclusive guards. As explained in Section 2, conditional jumps (*`JUMPI` *opcodes) take the two top-most elements from the stack, the jump address and a value (that comes from checking some boolean condition). If the value is 1, then the execution jumps to the opcode located in the address read from the stack and if it is 0 it continues with the next opcode.*

■

3.2. A Gas-Aware Semantics

Rules in Figure 3 define a *gas-aware operational semantics* for the (EVM) RBR. We use $a \cdot as$ to denote a non-empty sequence of elements, where as can be the empty sequence, denoted as ϵ. An *activation record* has the form $\langle r, bs, st \rangle$, where r is a rule name, bs is a sequence of instructions in r, if it is not empty is denoted as $b \cdot bs$ where b is the next instruction to be executed, and st is a mapping of stack variables to their values $st : \text{ST} \mapsto \mathcal{Z}$, where

Figure 3: RBR Semantics

$\mathcal{Z}$ is the set of all possible values that can be stored in an EVM word, i.e., $\mathcal{Z} = \{z \in \mathbb{N} \cup \{0\} \mid z < 2^{256}\}$.

A *program state* $\mathbb{S}$ has the form $\langle A \cdot As, mem, sto, O, M\rangle$ where $A \cdot As$ is a sequence of activation records, *mem* is a mapping of local memory addresses to their values[1] $mem : \mathcal{Z} \mapsto \mathcal{Z}$, *sto* is a mapping of storage addresses to their values $sto : \mathcal{Z} \mapsto \mathcal{Z}$, *blc* is a mapping of blockchain variables $blc : \text{BLV} \mapsto \mathcal{Z}$, and $O, M \in \mathbb{N} \cup \{0\}$ correspond to the gas consumption accumulated up to the current state of the execution for instructions execution and memory allocation, respectively. Rule (1) handles the assignment of an expression to a stack variable s_i. According to the grammar seen before, this expression can be a constant, the contents of a memory or storage address, the contents of a blockchain variable, an arithmetic operation between stack variables, or some other operations represented by a functor as for example $xor(s_i, s_j)$. Function $eval(expr, st, mem, sto)$ evaluates the expression using 256-bit arithmetics with respect to the mappings of stack, memory and storage variables. Rule (2) models the assignment of a

[1] Although the local memory is byte addressable with instruction `MSTORE8`, for the sake of keeping the semantics simple we will only consider the general case of word-addressable `MSTORE`.

blockchain data to a stack variable. Rules (3) and (4) correspond to assignments of stack variables to memory or storage elements. Rule (5) corresponds to the evaluation of a `nop` annotation of any EVM instruction except external calls, *i.e.* the ones in the set $Call = \{\texttt{CALL}, \texttt{CALLCODE}, \texttt{DELEGATECALL}, \texttt{STATICCALL}\}$. At this point, the gas consumption O, M in the program state must be updated with the cost of the corresponding EVM instruction. For this purpose, functions $op\_gas(instr, st, sto)$ and $mem\_gas(instr, st)$ provide the gas consumed by $instr$, as defined in Appendix H of [1], and will be defined on the RBR in Sections 3.3.1 and 3.3.2 for some representative cases. `GAS` is the blockchain variable that contains the amount of available gas for the current transaction. If the execution exceeds the gas limit there is no matching rule in the semantics and hence the execution does not progress further (in the blockchain execution it throws an out-of-gas exception). The instructions that correspond to external calls, i.e., those in set $Call$, are handled separately by Rule (6). In this case, besides the gas consumed by the instruction itself, the gas limit must also consider the gas consumed by the external contract execution. In addition, according to [1], an external call might change the data stored in memory and storage (for instance, external contracts might perform callback calls to the current contract, possibly modifying storage variables). This is expressed in the semantics by a call to the function $external\_call(instr, st, mem, sto)$, that returns a tuple $\langle mem_e, sto_e, gas_e\rangle$ that contains the state of the local memory and the storage after executing the external call, as well as the gas consumed by the executed contract, respectively. Rule (7) evaluates a call to a rule in the RBR. A fresh activation record is created containing the code of the called rule, and the argument values are passed to it. Finally, Rule (8) corresponds to the termination of an activation record.

Program execution. The execution of a function f of a smart contract starts from an initial state $\mathbb{S}_0$ of the form $\langle\langle \texttt{block}0, b \cdot bs, st_\emptyset\rangle, mem_\emptyset, sto_0, 0, 0\rangle$, where `block`0 is the RBR rule for the first block, $b \cdot bs$ is the block code, $st_\emptyset$ and $mem_\emptyset$ represent the empty mappings for stack variables and local variables, respectively, and sto_0 contains the current mapping of state variables to their values in the blockchain. Additionally, *blc* contains the current state of blockchain variables as well as message data such as the hash code bound to f. A trace is of the form $t \equiv \mathbb{S}_0 \rightsquigarrow \cdots \rightsquigarrow \mathbb{S}_i \rightsquigarrow \cdots \mathbb{S}_n$. As we have already mentioned, there are no infinite traces: Rules (5) and (6) guarantee that any execution of a smart contract terminates, as the gas consumed is limited to the amount of gas set in the blockchain variable `GAS` at the beginning of the execution. Therefore, an execution trace terminates either because it runs out of gas, or the last state of the trace is of the form $\langle\epsilon, \_, \_, \_, \_\rangle$, where ϵ is the empty sequence of activation records.

Definition 3 (RBR gas cost)**.** *Given a function f of a smart contract with initial storage state sto_0 and blockchain information blc, the execution trace of f on blc is of the form $t \equiv \mathbb{S}_0 \rightsquigarrow \cdots \rightsquigarrow \mathbb{S}_i \rightsquigarrow \cdots \mathbb{S}_n$ where $\mathbb{S}_n = \langle\epsilon, mem_n, sto_n, O_n, M_n\rangle$ and its gas consumption is defined as $\mathcal{G}(f, blc) = O_n + M_n$.* ∎

3.3. The Gas Cost Model

The EVM gas cost model is complex and unconventional. Its computation can be separated into two different components: (1) one part depends on the EVM instruction executed, represented by the function *op_gas* (described in Section 3.3.1); and (2) another part of the gas consumed is related to the memory consumption represented by function *mem_gas* (described in Section 3.3.2).

3.3.1. Opcode gas model

In this section we focus on the computation of the gas attributed to the EVM instructions. In order to compute the gas fee bound to each opcode, we define the function *op_gas* that is used in the RBR semantics.

Definition 4 (opcode cost function)**.** *Given an EVM opcode b, a stack mapping st, and a storage mapping sto, op_gas is a function defined for the different EVM opcodes in the following table:*

	b	$op\_gas(b, st, sto)$
Fixed	$b \in FixedCost$	$k_{[1]}$
Cond. Constant	`SSTORE`	$\begin{cases} 20000 & \text{if} \quad st(s_{n-2}) \neq 0 \wedge st(s_{n-1}) = i \wedge \ sto(i) = 0 \\ 5000 & \text{otherwise} \end{cases}$

	`CALL`
	`DELEGATECALL`
	`CALLCODE`
	`STATICCALL`
		$700 + C_{XFER} + C_{NEW}$
		$C_{XFER} \equiv \begin{cases} 9000 & \text{if} \quad st(s_{n-3}) \neq 0 \\ 0 & \text{otherwise} \end{cases}$
		$C_{NEW} \equiv \begin{cases} 25000 & \text{if} \quad \texttt{DEAD}(st(s_{n-2})) \wedge \ st(s_{n-3}) \neq 0 \\ 0 & \text{otherwise} \end{cases}$

	`SELFDESTRUCT`	$5000 + \begin{cases} 25000 & \text{if} \quad \texttt{DEAD}(st(s_{n-1})) \wedge \ \texttt{BALANCE} \neq 0 \\ 0 & \text{otherwise} \end{cases}$
Parametric	`EXP`	$\begin{cases} 10 & \text{if} \quad st(s_{n-2}) = 0 \\ 10{+}50{\cdot}(1{+}\lfloor \log_{256}(st(s_{n-2}) \rfloor) & \text{if} \quad st(s_{n-2}) > 0 \end{cases}$

	`CALLDATACOPY`
	`CODECOPY`
	`RETURNDATACOPY`
		$3 + 3 \cdot \lceil st(s_{n-3}) \div 32 \rceil$

	`EXTCODECOPY`	$700 + 3 \cdot \lceil st(s_{n-4}) \div 32 \rceil$
	`LOGX`	$375{+}8{\cdot}st(s_{n-2}){+}X \cdot 375$ *where* $X \in \{0, 1, 2, 3, 4\}$
	`SHA3`	$30 + 6 \cdot \lceil st(s_{n-2}) \div 32 \rceil$

■

Observe that *op_gas* takes three elements, an EVM opcode, the state of the stack and the mapping of state variables when reaching the opcode, and returns the gas consumed by the corresponding instruction. It can be seen that Definition 4 distinguishes between three types of instructions:

1. Most bytecode instructions have a *fixed* constant gas consumption. We define the set

$$\begin{aligned} FixedCost \quad = \quad & W_{zero} \cup W_{base} \cup W_{verylow} \cup \\ & W_{low} \cup W_{mid} \cup W_{high} \cup W_{extcode} \cup \\ & \{\texttt{SLOAD}, \texttt{EXTCODE}, \texttt{BALANCE}, \texttt{CREATE}, \texttt{JUMPDEST}, \texttt{BLOCKHASH}\} \end{aligned}$$

using the sets defined in [1] for all these instructions. For example, the gas consumption of `JUMPDEST` is 1, and the gas consumption of `SLOAD` is 200. We use a generic k in Definition 4 to refer to these fixed costs.

2. Bytecode instructions that have different *constant* gas consumption gs_1 or gs_2 depending on some given condition. This is the case of `SSTORE` that costs $gs_1 = 20000$ if the storage value is set from zero to non-zero (first assignment), and $gs_2 = 5000$ otherwise. It is also the case for `CALL` and `SELFDESTRUCT`. To refer to all these instructions, we define the set

$$\begin{aligned} ConstantCost \quad = \quad & \{\texttt{SSTORE}, \texttt{CALL}, \texttt{DELEGATECALL}, \texttt{CALLCODE}, \\ & \texttt{STATICCALL}, \texttt{SELFDESTRUCT}\} \end{aligned}$$

   The gas consumption for these EVM opcodes, except `SSTORE`, in Definition 4 varies with the actual value of blockchain variables such as `BALANCE`, `GAS`, and `DEAD` (the latter checks if an account is non-existent or it has no code, no activity and zero balance). Observe that the execution cost of the external code invoked by call instructions is not included in this gas model, as it is already captured by means of the function *external_call* in Rule (6) in the semantics.

3. Bytecode instructions with a non-constant (*parametric*) gas consumption that depends on the value of some stack location. We define the set

$$\begin{aligned} ParamCost \quad = \quad & \{\texttt{EXP}, \texttt{CALLDATACOPY}, \texttt{CODECOPY}, \texttt{RETURNDATACOPY}, \\ & \texttt{EXTCODECOPY}, \texttt{LOGX}, \texttt{SHA3}\} \end{aligned}$$

   to include them. A complex case is `EXP`, whose gas consumption is defined as $10 + 50 \cdot (1 + \lfloor log_{256}(st(s_{n-2})) \rfloor)$ if $s_{n-2} \neq 0$ (and 10 otherwise), i.e, the gas consumption depends on the value of the exponent, that is stored in s_{n-2}, accounting for the larger computational effort.

3.3.2. Memory gas model

The second component that adds gas to a transaction is the amount of memory accessed. The memory gas of an EVM instruction b is defined in [1] as the difference $C_{mem}(\mu') - C_{mem}(\mu)$, where μ and μ' denote the *highest memory slot* accessed in the local memory before and after the execution of b, respectively, and $C_{mem}(a)$ is a function that, for a given memory slot a, is defined as

$$C_{mem}(a) \;=\; 3 \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor .$$

The cost computed for all the EVM instructions is accumulated to obtain the final memory cost. Column μ' in Definition 5 shows how μ' is computed in [1], given an EVM instruction b, the local state of the stack st, and the actual highest memory slot accessed μ. It uses an auxiliary function M, defined as follows:

$$M(h, f, l) = \begin{cases} h & \text{if} \quad l = 0 \\ max(h, \lceil (f + l)/32 \rceil) & otherwise \end{cases}$$

It can be seen that, besides `MLOAD` or `MSTORE`, instructions like `SHA3` or `CALL`, among others, use the local memory, and hence can increase the memory gas cost.

In our semantics, the maximum cost is computed by Rules (5) and (6) of Figure 3. Hence, the function *mem_gas* has to infer only the memory gas of the slot being accessed by each operation executed in the program.

Definition 5 (memory cost function)**.** *Given an EVM opcode b, and a stack mapping st, mem_gas is defined as*

$$mem\_gas(b, st) = C_{mem}(highest(b, st))$$

where $highest(b, st)$ is defined as follows:

b	μ'	$highest(b, st)$
MLOAD	$max(\mu, \lceil (st(s_{n-1}) + 32)/32 \rceil)$	$\lceil (st(s_{n-1}) + 32)/32 \rceil$
MSTORE	$max(\mu, \lceil (st(s_{n-1}) + 32)/32 \rceil)$	$\lceil (st(s_{n-1}) + 32)/32 \rceil$
MSTORE8	$max(\mu, \lceil (st(s_{n-1}) + 1)/32 \rceil)$	$\lceil (st(s_{n-1}) + 1)/32 \rceil$
CALLDATACOPY *CODECOPY* *RETURNDATACOPY*	$M(\mu, st(s_{n-1}), st(s_{n-3}))$	$M_{RBR}(st(s_{n-1}), st(s_{n-3}))$
EXTCODECOPY	$M(\mu, st(s_{n-2}), st(s_{n-4}))$	$M_{RBR}(st(s_{n-2}), st(s_{n-4}))$
LOGX	$M(\mu, st(s_{n-1}), st(s_{n-2}))$	$M_{RBR}(st(s_{n-1}), st(s_{n-2}))$
CALL *CALLCODE*	$M(M(\mu, st(s_{n-4}), st(s_{n-5})),$ $st(s_{n-6}), st(s_{n-7}))$	$max(M_{RBR}(st(s_{n-4}), st(s_{n-5})),$ $M_{RBR}(st(s_{n-6}), st(s_{n-7})))$
DELEGATECALL *STATICCALL*	$M(M(\mu, st(s_{n-3}), st(s_{n-4})),$ $st(s_{n-5}), st(s_{n-6}))$	$max(M_{RBR}(st(s_{n-3}), st(s_{n-4})),$ $M_{RBR}(st(s_{n-5}), st(s_{n-6})))$
RETURN	$M(\mu, st(s_{n-1}), st(s_{n-2}))$	$M_{RBR}(st(s_{n-1}), st(s_{n-2}))$
REVERT	$M(\mu, st(s_{n-1}), st(s_{n-2}))$	$M_{RBR}(st(s_{n-1}), st(s_{n-2}))$
SHA3	$M(\mu, st(s_{n-1}), st(s_{n-2}))$	$M_{RBR}(st(s_{n-1}), st(s_{n-2}))$
CREATE	$M(\mu, st(s_{n-2}), st(s_{n-3}))$	$M_{RBR}(st(s_{n-2}), st(s_{n-3}))$

and function M_{RBR} is defined as

$$M_{RBR}(f, l) = \begin{cases} 0 & \text{if} \quad l = 0 \\ \lceil (f + l)/32 \rceil & \text{otherwise} \end{cases}$$

■

The last column of the table shown in Definition 5, $highest(b, st)$, shows the computation of the highest memory slot accessed by the EVM instruction b in the RBR semantics. Note that M_{RBR} is replacing the original M above with the only difference that the computation of the maximum is extracted from it and computed by Rules (5) and (6). Hence, M_{RBR} returns 0 if $l = 0$ instead of the highest slot of memory accessed up to this point as M does, and the maximum cost is kept thanks to the *max* function in Rules (5) and (6).

4. GAS-RBR: RBR Transformations for Gas Analysis

An RBR program, together with its instrumented semantics, allows *dynamically* computing the gas consumption of concrete EVM transactions. However,

in order to enable the use of existing analyzers for *statically* inferring gas bounds that are sound for *any* transaction, we need to make some transformations into the RBR. Section 4.1 details the adaptations in the RBR to model the local memory and the storage accesses by means of local variables. This allows that a resource analysis that only handles local variables can be used to infer gas on EVM programs, without requiring specific extensions for the EVM memory and storage models. Besides, as the contents in the storage and in the memory can be modified when a call to another contract is executed, in Section 4.2 we describe some additional transformations. Finally, in Section 4.3 we will see how RBR programs are transformed to abstract strings and byte arrays into their sizes, enabling a resource gas analyzer to deal with them.

4.1. Flattening the Storage and Memory Locations

It is well known that shared mutable data structures, such as those stored in the heap, are the bane of formal reasoning and static analysis and most tools are not able to keep track of object fields nor of array contents. As we have seen in Section 3.2, the RBR semantics models the local memory and storage with arrays, imitating their treatment in EVM programs. Inspired by the ideas of [19], we aim at flattening memory/storage allocated data by means of variables that contain the contents of particular addresses of the memory/storage. This avoids the need of extending resource analyzers with a complex treatment for reasoning on array contents. Our proposal consists in analyzing the addresses used in the load and store instructions, *i.e.*, `MLOAD`, `MSTORE`, `SLOAD` and `SSTORE`, to identify those addresses accessed that are trackable. Basically, if the load-/store instruction always accesses the same address, we use a explicit variable for modeling this location and otherwise we do not track it. Existing *heap* and *pointer* [20, 21, 22] analyses could be applied to gain accuracy in modelling these accesses. As this is not the goal of our analysis, we just assume the following: (1) V_{mem} (V_{sto}) is a set of the form $\{l_0, l_1, ..., l_k\}$ ($\{g_0, g_1, ..., g_h\}$, respectively) containing all memory (storage) variables identified by the analysis; (2) given a program point pp with instruction `MLOAD` or `MSTORE`, function $getMemV(pp)$ returns a variable of the form l_i when the address accessed at pp can be uniquely identified by the analysis or $\perp$ when it cannot be identified; and, (3) $getStV(pp)$ returns g_i or $\perp$ analogously.

By means of these functions, the GAS-RBR program can be refined to, instead on including `MEM`, `STO` accesses as we have in the RBR, include the concrete variables identified by the analysis, modifying the corresponding RBR instructions to use them. The following definition illustrates the refinement of the RBR to deal with flattened memory and storage accesses. Besides, we use V_{blc} to represent all blockchain variables that can be extracted.

Definition 6 (GAS-RBR rule refinement #1)**.** *Given an RBR rule of the form*

$$rule(\bar{s}, \texttt{MEM,STO,BLC}) \Rightarrow g \mid b_i, \ldots b_j, call(rule2(\bar{y}, \texttt{MEM,STO,BLC}))$$

the refined GAS-RBR rule is computed as follows:

$$rule(\bar{s}, V_{mem}, V_{sto}, V_{blc}) \Rightarrow g \mid \nu(b_i), \ldots \nu(b_j), call(rule2(\bar{y}, V_{mem}, V_{sto}, V_{blc}))$$

where function $\nu(b_{pp})$ is defined as follows:

EVM	b_{pp}	$\nu(b_{pp})$	
MLOAD	$s_x = \mathit{MEM}[s_x]$	$s_x = getMemV(pp)$	*if* $getMemV(pp) \neq \bot$
		$s_x = \mathit{fresh()}$	*if* $getMemV(pp) = \bot$
MSTORE	$\mathit{MEM}[s_x] = s_y$	$getMemV(pp) = s_y$	*if* $getMemV(pp) \neq \bot$
		$l_i = \mathit{fresh()} \quad \forall l_i \in V_{mem}$	*if* $getMemV(pp) = \bot$
SLOAD	$s_x = \mathit{STO}[s_x]$	$s_x = getStV(pp)$	*if* $getStV(pp) \neq \bot$
		$s_x = \mathit{fresh()}$	*if* $getStV(pp) = \bot$
SSTORE	$\mathit{STO}[s_x] = s_y$	$getStV(pp) = s_y$	*if* $getStV(pp) \neq \bot$
		$g_i = \mathit{fresh()} \quad \forall g_i \in V_{sto}$	*if* $getStV(pp) = \bot$
CALLDATASIZE	$s_x = \mathit{BLC}(\mathit{CALLDATASIZE})$	$s_x = \mathit{CALLDATASIZE}$	
ADDRESS	$s_x = \mathit{BLC}(\mathit{ADDRESS})$	$s_x = \mathit{ADDRESS}$	
CALLVALUE	$s_x = \mathit{BLC}(\mathit{CALLVALUE})$	$s_x = \mathit{CALLVALUE}$	
...			

■

Observe that `MLOAD` and `SLOAD` RBR instructions are replaced by an assignment of the concrete memory/storage variable to its destination stack variable when the concrete variable can be tracked. However, when the address is not known, we assign function `fresh` to model the lack of information about its content. We use `fresh`() to denote a generator of fresh variables to safely represent the unknown value of the corresponding addresses. Analogously, `MSTORE` and `SSTORE` are replaced by assignments to the particular memory or storage variable. The treatment when the address accessed by an `MSTORE` or `SSTORE` is unknown extends to all memory/storage variables, respectively. As the destination of the assignment is not known, we have to "*forget*" all memory contents (or storage, respectively) from that point on, since the writing may affect any memory location (or storage location, respectively) and it is not sound any more to rely on the previously collected information. Our implementation though is able to detect the positions in which the length of the arrays is stored and it does not forget it (this is safe as this length cannot be modified when the bytecode has been obtained by compilation from a Solidity code). Keeping this length is crucial for the accuracy of the gas analysis.

Example 3. *Function $getStV(pp)$ applied to program points* 948 *and* 94F *in block* 941 *in Figure 2 returns g_{10} as the EVM instructions `SSTORE` and `SLOAD` located in these program points always operate over the same state variable. In these cases, the storage address 10 (`0x0a` in hexadecimal) is located in the stack with the EVM instructions `PUSH1 0x0a`. After that, it is used by `SSTORE` and `SLOAD` to access to the 11th slot of storage (represented by g_{10}). Something similar happens when function $getMemV()$ is applied to program point* 673 *in block* 66F*, that returns l_0. However if we apply $getStV(pp)$ to the `SLOAD` located at program point* 67A*, we obtain $\bot$ as it is loading a different storage location at each iteration of the loop (it accesses the storage address generated by the previous `SHA3` instruction as it is traversing the array `slots` of Figure 2).*

If we consider only the blocks in Figure 2, function $V_{sto} = \{g_2, g_3, g_{10}\}$ and $V_{mem} = \{l_0\}$. These variables are added to the head of the rules. For instance,

the rule for block 653 in Example 2 now is:

$$\begin{aligned} &block\_653\_1(s_0, \ldots, s_8, l_0, g_2, g_3, g_{10}) \Rightarrow \\ &\quad \ldots, \\ &\quad s_{10} = g_{10}, \texttt{nop(SLOAD)}, \\ &\quad \ldots, \\ &\quad call(jump\_653\_1(s_0, \ldots, s_{10}, l_0, g_2, g_3, g_{10})) \end{aligned}$$

hence a resource analyzer that works on integer variables can reason on the contents of memory/storage/blockchain data without requiring specific extensions.

■

4.2. External calls

External contracts are invoked by the EVM instructions `CALL`, `CALLCODE`, `DELEGATECALL`, `STATICCALL` and handled by Rule (6) in the semantics. In these cases, the address of the invoked contract is passed as an element on the stack, and the actual code might only be known at run-time. There are three aspects to be taken into account when analyzing a contract with external calls: the gas consumed by the invoked contract, the existence of *callbacks*, and the return value. We focus on the gas consumption of the execution of a single contract and thus we do not add the gas consumed by external contracts which, on the other hand, might only be known at run-time. Regarding the second aspect, since an external call might make in turn a call to the invoking contract to execute some of its functions (this is known as a *callback*), we do have to consider that the external code, whose code is unknown at analysis time, might modify the storage of the invoking contract. In order to be sound, the GAS-RBR program must therefore "forget" any previous value of the storage, as we have done when a `SSTORE` accesses an unknown storage address. Finally, the results returned by external calls are stored in a sequence of positions of the local memory of variable length known at run-time. Therefore, we also have to assign to fresh any previous value of the local memory. The following refinement on the RBR is done to soundly approximate possible modifications performed in the memory and the storage due to an external call. Note that we apply this refinement on an RBR already refined.

Definition 7 (GAS-RBR rule refinement #2)**.** *Given an RBR rule of the form*

$$rule(\bar{s}, V_{mem}, V_{sto}, V_{blc}) \Rightarrow g \mid b_i, \ldots b_j, call(rule2(\bar{y}, V_{mem}, V_{sto}, V_{blc}))$$

the refined GAS-RBR rule is computed as follows:

$$rule(\bar{s}, V_{mem}, V_{sto}, V_{blc}) \Rightarrow g \mid \eta(b_i), b_i, \ldots \eta(b_j), b_j, call(rule2(\bar{y}, V_{mem}, V_{sto}, V_{blc}))$$

where function $\eta(b_{pp})$ *is defined as follows*

b_{pp}	$\eta(b_{pp})$

`nop(CALL)`
`nop(CALLCODE)`
`nop(DELEGATECALL)`
`nop(STATICCALL)`
	$l_i = \mathit{fresh}() \quad \forall l_i \in V_{mem}$
	$g_i = \mathit{fresh}() \quad \forall g_i \in V_{sto}$

■

In this case, the refinement described in Definition 7 has to maintain the original `nop` functors represented by the instructions b_i and b_j as they are needed to compute the gas consumed.

4.3. String and Byte arrays

Solidity storage variables of types `string` and `bytes` are particular cases of Solidity dynamic arrays that are challenging for resource analysis because the way they are stored by the EVM depends on their actual sizes. In particular, if they are shorter than 31 bytes, their data is stored packed in the word of the storage slot corresponding to that variable together with its length. Otherwise, the slot word just contains the length and the data contents are stored at the address obtained like for standard arrays.

Typically loops that traverse data structures have a cost that is proportional to the size of the data structure being traversed. Hence, resource analyzers use a size analysis to (over-)approximate the size of the data structures of the program and then bound the number of iterations of loops and infer upper-bounds for them. In our language, inferring the size of the arrays and strings is crucial for obtaining precise results in the gas analysis. Unfortunately, the instructions used to access the length of these packed arrays include bit-wise operations, among others, that state-of-the-art size analysis cannot handle. Standard size analysis works for the case of unpacked string and bytes. Developing a specific size analysis for this purpose would be challenging and, besides, it is unnecessary as the issue can be solved by means of a simple transformation as follows. We match in the rules of the RBR the particular sequence of instructions generated by the compiler (which are always the same) that start by pushing the contents of the string or bytes variable at the top of the stack, and then obtaining its length, leaving it stored at the top of the stack (at the same position). Every time we find this pattern of instructions applied to a variable of type `string` or `bytes`, we remove them from the RBR as we are only interested in their size (keeping their nops to account for their gas). Leaving them would make a size analyzer that is unable to deal with bit-wise operations assume an unknown value and therefore also lose the information that is stored at the top of the stack. Importantly, since the top of the stack indeed contains the size, by removing the remaining instructions, the analyzer is able to handle that stack variable and reason on the size of the `string` and `bytes`. In particular, assuming that we have placed the contents of the string or bytes variable at the top of

the stack, which is s_i, the transformation applied is the following:

$$\boxed{\begin{array}{l} s_{i+1} = 1, \texttt{nop(PUSH1)}, s_{i+2} = s_i, \texttt{nop(DUP2)}, s_{i+3} = 1, \texttt{nop(PUSH1)}, \\ s_{i+2} = \mathit{and}(s_{i+3}, s_{i+2}), \texttt{nop(AND)}, s_{i+2} = \mathit{eq}(s_{i+2}, 0), \texttt{nop(ISZERO)}, \\ s_{i+3} = 256, \texttt{nop(PUSH2)}, s_{i+2} = s_{i+3} * s_{i+2}, \texttt{nop(MUL)}, s_{i+1} = s_{i+2} - s_{i+1}, \\ \texttt{nop(SUB)}, s_i = \mathit{and}(s_{i+1}, s_i), \texttt{nop(AND)}, s_{i+1} = 2, \texttt{nop(PUSH1)}, \\ s_{i+2} = s_i, s_i = s_{i+1}, s_{i+1} = s_{i+2}, \texttt{nop(SWAP1)}, \\ s_i = s_{i+1}/s_i, \texttt{nop(DIV)} \end{array}}$$

$$\Downarrow \textit{(refinement \#3)}$$

$$\boxed{\begin{array}{l} \texttt{nop(PUSH1)}, \texttt{nop(DUP2)}, \texttt{nop(PUSH1)}, \texttt{nop(AND)}, \texttt{nop(ISZERO)}, \texttt{nop(PUSH2)}, \\ \texttt{nop(MUL)}, \texttt{nop(SUB)}, \texttt{nop(AND)}, \texttt{nop(PUSH1)}, \texttt{nop(SWAP1)}, \texttt{nop(DIV)} \end{array}}$$

This transformation is applied whenever possible and, *e.g.*, it is needed to infer bounds for the functions `getPlayers` and `getSlots`, studied in the experiments reported in Figure 4. Without this transformation, the tool fails to find gas bounds.

5. Gas Analysis of EVM Transactions

As we have seen in Section 3.3, the computation of the gas fee can be split into two different parts, one part related to the EVM opcodes executed, and another part that depends on the memory addresses accessed. Likewise, our analysis splits the computation of the gas upper bound $\widehat{\mathcal{U}}_g$ of a transaction in two different upper bounds:

$$\widehat{\mathcal{U}}_g = \widehat{\mathcal{U}}_{op} + \widehat{\mathcal{U}}_{mem}$$

where $\widehat{\mathcal{U}}_{op}$ corresponds to the *opcode gas upper-bound* that can be computed using a standard resource analyzer [23]; and $\widehat{\mathcal{U}}_{mem}$ which is the *memory gas upper-bound* whose computation is performed by means of a *peak* resource analyzer [6].

5.1. Opcode Gas Upper-Bound

Resource analysis, a.k.a. cost analysis, is a rather complex type of analysis that we handle in this article as a black-box since existing techniques can be used. Resource analyzers, in order to quantify the cost associated to an instruction, use the generic notion of *cost model*. A cost model is a function θ that assigns a measure of the cost of executing to each instruction in the program. We represent the resource analysis as a function $\mathit{cost}(P, \theta)$ that receives an RBR program P and a cost model θ and returns an upper-bound $\widehat{\mathcal{U}}(f)$ on the resource modeled by θ for each function f in P. In order to use this function *cost* for our purpose, we need to provide now the definition of a static *opcode gas model* θ_{op}. Note that the gas model in Section 3.3.1 is dynamic, i.e., it describes the gas usage for concrete executions, while in order to define θ_{op} we need a static gas model that provides a gas cost that is sound for any concrete execution. As explained in Section 3.3.1, we can distinguish three kinds of instructions depending on their gas consumption (see Definition 4 for details):

1. instructions that have a *fixed* gas consumption whose cost can be directly applied in the cost model,

2. instructions that could have different *constant* gas consumption depending on a concrete value stored in the stack,

3. instructions with a non-constant (*parametric*) gas cost whose concrete consumption depends on a value stored in the stack.

We can see that the gas cost of instructions included in points (2) and (3) is dynamic as it depends on concrete values stored in the stack, which in general are unknown at static analysis time. One possible but imprecise solution for defining the cost in the static cost model is to take always the worst case cost, e.g. always consider cost 20000 for instruction `SSTORE`. We can improve the precision of the static cost model by using a value analysis which finds out whether a stack variable contains a constant value to be stored on storage and hence we can determine the concrete cost.

Given an *RBR* instruction $inst \equiv pp{:}b$ located at program point pp, we assume that we have a function $getVal(pp, s_i)$ that returns the inferred value stored in s_i, if it is constant at program point pp or $\perp$ if the value is not known or not constant at pp. We do not detail the implementation of this function, as it can be done with a simple syntactic analysis or a more precise semantic constancy or value analysis. Additionally, we use function $getSize(pp)$ (see Section 2) to get the size of the stack at RBR program point pp.

Definition 8. *Given an EVM instruction b at program point pp and $n = getSize(pp)$, we define the static* EVM op cost model*,*

$$\theta_{op}(pp{:}b) = \begin{cases} 0 & \text{if} \quad b \neq \texttt{nop}(\_) \\ op\_cost(pp{:}b) & \text{if} \quad b = \texttt{nop}(b) \end{cases}$$

where $op\_cost(pp{:}b)$ is defined in the following table

	b	$op\_cost(pp{:}b)$
Fixed	$b \in FixedCost$	$k_{[1]}$
Constant	`SSTORE`	$\begin{cases} 5000 & if\ getVal(pp, s_{n-2}) = 0 \\ 20000 & otherwise \end{cases}$
	`CALL` `DELEGATECALL` `CALLCODE` `STATICCALL`	$700 + C_{XFER} + C_{NEW} + C(CALL)$ $C_{XFER} \equiv \begin{cases} 9000 & if\ \ getVal(pp, s_{n-3}) \neq 0 \\ 0 & otherwise \end{cases}$ $C_{NEW} \equiv 25000$
	`SELFDESTRUCT`	30000
Parametric	`EXP`	$\begin{cases} 10 & if\ \ getVal(pp, s_{n-2}) = 0 \\ 10{+}50{\cdot}(1{+}\lfloor \log_{256}(s_{n-2}) \rfloor) & if\ \ getVal(pp, s_{n-2}) > 0 \end{cases}$
	`CALLDATACOPY` `CODECOPY` `RETURNDATACOPY`	$3 + 3 \cdot \lceil s_{n-3} \div 32 \rceil$
	`EXTCODECOPY`	$700 + 3 \cdot \lceil s_{n-4} \div 32 \rceil$
	`LOGX`	$375{+}8{\cdot}s_{n-2}{+}\mathbf{X} \cdot 375$ *where* $\mathbf{X} \in \{0, 1, 2, 3, 4\}$
	`SHA3`	$30 + 6 \cdot \lceil s_{n-2} \div 32 \rceil$

■

Note that in the definition of θ_{op} we only consider the cost associated to the `nop` instructions, that is, instructions that directly correspond to EVM instructions, the remaining RBR instructions do not add any cost. In the cost model definition, we can see the following considerations: (1) the cost produced by fixed cost instructions is like in the dynamic gas model; (2) instructions whose cost is constant but it depends on a concrete value stored in the stack rely on $getVal$ to retrieve the value stored in the corresponding stack element or, if it is unknown, we consider the worst case cost; (3) as we do not have information about the state of the blockchain, for those expressions that depend on the results returned by $DEAD$ we directly consider the worst case cost, e.g. in `SELFDESTRUCT`; (4) as we analyze single contracts, the execution cost of external calls is added as a symbolic value represented by C(CALL); and (5) instructions with parametric cost directly include the stack variables in the cost expression.

The GAS-RBR we are producing adheres to the rule based representations of [23, 24], hence we can directly feed the resource analyzers with our GAS-RBR. We have needed only to implement the cost model θ_{op}, and the analyzer returns *closed-form gas upper-bounds*, i.e., a cost expression that is parametric on the input arguments of the function. Note that the input arguments of the function are, not only the function parameters, but also the list of storage variables inferred and other blockchain values associated to the transaction, e.g. `CALLDATASIZE` (see Section 4.1 for details). The resource analyzer is able, without requiring any modification and in a fully automated way, to find a cost expression, which over-approximates the amount of gas related to the operations executed in EVM.

Example 4. *The gas bound computed by* GASTAP *for function* `findWinner` *(see Figure 2) is* $1555 + 779 \cdot \mathsf{nat}(g3)$*, which is is parametric on the state variable* g_3 *that corresponds to the size of the array slots. Function* `nat` *is defined as*

$\mathsf{nat}(x) = x$ *if* $x > 0$ *and* $\mathsf{nat}(x) = 0$ *otherwise. Other gas upper-bounds can be found in Figure 4 in Section 6. Note that these upper-bounds are parametric on different state variables, input and blockchain data.*

■

5.2. Memory Gas Upper-Bound

As we have already mentioned, the problem of inferring the memory gas bound boils down to (over)-approximating the highest memory address accessed by the transaction. Our approach to solve this problem is to view it as an instance of the peak resource analysis problem [6, 7]. This analysis, rather than accumulating all costs as in standard resource analysis, computes the peak (*i.e.*, the supremum) of the resource consumption of the whole execution. The cost model in this case is atypical as the resource we need to measure is the value of the memory location. Thus, for each instruction that accesses a memory location l we count as it allocates l resources. The work at [6] uses the notation `acquire(e)` to allocate `e` amount of resources, where `e` is a cost expression (that can include variables). The peak analysis allows inferring the maximal value of all `acquire(e)` annotations. Analogously to the case of standard cost analysis, we represent the peak analysis as a function $peak(P)$ that returns the peak (*i.e.*, the supremum) of the resource consumption of the `acquire(e)` annotations within the RBR program P. Thus, we just need to perform a transformation into the GAS-RBR program (see Section 4) that converts the memory accesses like `MLOAD addr` into allocations of `addr` resources. The slots of memory used by each EVM instruction according to [1] is given in Definition 5. Note that in some cases, it depends on a given condition that checks, by comparing the corresponding stack variable with 0, if the memory address is accessed. We include this information within `acquire` statements before the `nop` functors that contain EVM instructions that access to memory as shown in the following table, where $\texttt{div}(x, y)$ stands for $\lceil x/y \rceil$ and s_{n-1} for the top-most stack variable. Then, the peak resource analysis computes the maximum of all the expressions within `acquire`, what corresponds to the highest memory slot addressed in the whole execution.

b	*Transformed code*	*Condition*
MLOAD MSTORE	acquire(div$((s_{n-1}+32),32)$)	
MSTORE8	acquire(div$((s_{n-1}+1),32)$)	
CALLDATACOPY CODECOPY RETURNDATACOPY	acquire(div$((s_{n-1}+s_{n-3}),32)$)	if $getVal(pp, s_{n-3}) \neq 0$
EXTCODECOPY	acquire(div$((s_{n-2}+s_{n-4}),32)$)	if $getVal(pp, s_{n-4}) \neq 0$
LOGX	acquire(div$((s_{n-1}+s_{n-2}),32)$)	if $getVal(pp, s_{n-2}) \neq 0$
CALL CALLCODE	acquire(div$((s_{n-4}+s_{n-5}),32)$) acquire(div$((s_{n-6}+s_{n-7}),32)$)	if $getVal(pp, s_{n-5}) \neq 0$ if $getVal(pp, s_{n-7}) \neq 0$
DELEGATECALL STATICCALL	acquire(div$((s_{n-3}+s_{n-4}),32)$) acquire(div$((s_{n-5}+s_{n-6}),32)$)	if $getVal(pp, s_{n-4}) \neq 0$ if $getVal(pp, s_{n-6}) \neq 0$
RETURN REVERT SHA3	acquire(div$((s_{n-1}+s_{n-2}),32)$)	if $getVal(pp, s_{n-2}) \neq 0$
CREATE	acquire(div$((s_{n-2}+s_{n-3}),32)$)	if $getVal(pp, s_{n-3}) \neq 0$

An alternative way to infer this information is by means of a size analysis [25] that computes upper-bounds on sizes of all expressions used to access the memory. After this, in a second step, a maximization of all these obtained upper-bound sizes would be required. In contrast to this approach, we directly obtain the maximum of all memory accesses. Finally, we apply the function C_{mem} defined in Section 3.3.2 on the result of the analysis to obtain the memory gas upper-bound $\widehat{\mathcal{U}}_{mem}(f)$ for each public function f in the contract. Let us give an example that illustrates the analysis:

Example 5. *The function* `findWinner` *(see Figure 2) executes* 8 *EVM instructions that access to memory: two* `MLOAD`*,* `MSTORE` *and* `RETURN` *in the rule* block_142*; and* `MSTORE` *and* `SHA3` *in rules* block_66F *and* block_691*. If we analyze the EVM instructions of* block_142*, we infer that* `MLOAD` *instructions load the value stored in the memory address* 64 *(contained in top-most stack variable* s_2*),* `MSTORE` *stores a value in the memory address* 128 *(in the top-most stack variable* s_5*) and* `RETURN` *operates on the values* 32 *(stored in* s_1*) and* 128 *(stored in the top-most stack variable* s_2*). Similarly, the* `MSTORE` *instructions of rules* block_66F *and* block_691*, stored a value in the memory address* 0 *(stored in the top-most stack variable* s_7*) and the* `SHA3` *instructions operates on the values* 0 *and* 32 *(stored in the top-most stack variables* s_7*, and* s_6 *respectively). If we compute the expressions that appear in the table above, we obtain that* 3*,* 5*,* 5*,* 1*, and* 1 *resources are acquired respectively. Hence the analysis returns* 5 *as the highest slot of memory accessed during the execution of the function* `findWinner`*. Finally, if we apply the function* C_{mem} *we obtain* 15*, the memory gas upper-bound for this function. More examples can be found in Figure 4 of Section 6. Finally, as the gas upper-bound for the function* `findWinner` *(Figure 2) is* $1555 + 779 \cdot nat(g3)$ *and the memory gas upper-bound is* 15*, we can conclude that its gas upper-bound is* $1570 + 779 \cdot nat(g3)$*.* ∎

Theorem 1. *Given a public function f of a GAS-RBR program P with blockchain information blc, the following holds:*

$$\mathcal{G}(f, blc) \leq \widehat{\mathcal{U}}_g(f)$$

where $\widehat{\mathcal{U}}_g(f) = \widehat{\mathcal{U}}_{op}(f) + \widehat{\mathcal{U}}_{mem}(f)$.

Sketch of proof. Soundness follows from the following facts: Program transformations of GAS-RBR programs in Sections 4 and 5 preserve the `nop` instructions of the original program. Regarding the opcode cost, it is straightforward to check that θ_{op} is an over-approximation of $op\_gas(b, st, sto)$ for every EVM instruction b, and soundness of the resource analysis $cost(P, \theta_{op})$ [23] ensures that we infer an upper bound of the opcode cost. Finally, the soundness of the peak analysis $peak(P)$ [6] guarantees as well that the memory gas cost component is a sound upper bound of the memory cost. □

6. Implementation and Evaluation

The analysis presented in this article has been implemented in a tool named GASTAP, *Gas-Aware Smart contracT Analysis Platform*, that takes as input a smart contract and automatically infers *sound* gas upper-bounds for its public functions. This section provides implementation details and the results of our evaluation of GASTAP. In Section 6.1, we provide some implementation details of GASTAP. In Section 6.2, we evaluate the accuracy of the gas bounds inferred by GASTAP on the EthereumPot by comparing them with the bounds computed by the Solidity compiler. In Section 6.3, we evaluate the efficiency and effectiveness of our tool by analyzing more than 34,000 Ethereum smart contracts obtained from the Ethereum blockchain using the popular Etherscan service [26]. The whole dataset used in Section 6.3 can be found at `https://github.com/costa-group/EthIR/tree/master/examples/gastap`. Note that the results presented in this section do not add the so called *intrinsic gas* cost of the execution as Solidity compiler does. However, GASTAP has a flag to incorporate the transaction fee of 2,300 gas. Finally, in Section 6.4, we assess the accuracy of the upper-bounds inferred by GASTAP. To do so, we compare the upper-bounds obtained on the 300 top-valued accounts by Ether (the most valuable ones) obtained using Etherscan service with the gas cost of more than 4000 real transactions.

GASTAP tool can be used from an online web interface at `https://costa.fdi.ucm.es/gastap` where we have also made available a subset of the smart contracts used for our experimental evaluation. To run the tool, the user has to either write her Solidity contract in the text area or to select an available one in the file manager area on the left side. In both cases, then the `Refresh Outline` allows selecting by means of a check button the function(s) whose gas cost is going to be inferred. Finally, by clicking on `Apply` the analysis starts and the gas bound is obtained. In the `Settings` menu it can be specified if the input is a Solidity program or an EVM code.

6.1. Implementation

Figure 1 shows in white boxes the components developed by us –that are all open-source– and in gray those that we use out-of-the-box. The SACO analyzer we are using is not open-source but it is available through an executable. We classify the components implemented by us in two groups:

- *Python implementation.* The components related to the generation of the stack sensitive control flow graph introduced in Section 2, the generation of the rule-based representation presented in Section 3 and the abstractions described in Section 4 have been implemented in Python. These components have been implemented in more than 18,000 lines of code (LOC).

- *Prolog implementation.* As the SACO analyzer is implemented in Prolog, we have implemented the components that are directly integrated in it in Prolog as well. In particular, the "Value Analysis" and "Gas Cost Model" components in Figure 1 are implemented in 613 LOC in Prolog. The first component implements a classical data-flow analysis and corresponds to the function $getVal$ used in Section 5.1 to retrieve the value stored in a given stack element. This analysis is used in the second component that contains the models described in Section 5.1. The fixed cost instructions are modeled with facts in Prolog specifying its concrete cost. The instructions whose cost is constant but it depends on a concrete value or those instructions with parametric cost are modeled with rules in Prolog that use the result of the value analysis to compute the cost.

The output of the different components is stored in text files, e.g., the *S-CFG* and the RBR are written in separated files that are read by the next components.

6.2. Gas Bounds for EthereumPot Case Study

Figure 4 shows in column **solc** the gas bound provided by the Solidity compiler **solc** [2], and in the next two columns the bounds produced by GASTAP for opcode gas and memory gas, respectively, for all public functions in the contract. If we add the gas and memory bounds, it can be observed that, for those functions with constant gas consumption, we are as accurate as **solc**. Hence, we do not lose precision due to the use of static analysis.

For those 6 functions that **solc** fails to infer constant gas consumption, it returns ∞. For opcode gas, we are able to infer precise *parametric* bounds for five of them, `rewardWinner` is linear on the size of the first and third state variables ($g1$ and $g3$ represent resp. the sizes of the arrays `addresses` and `slots` in Figure 2), `getSlots` and `findWinner` on the third, `getPlayers` on the first, and `__callback` besides depends on the value of `result` (second function parameter) and `proof` (last parameter). It is important to note that, although the Solidity source code of some functions (*e.g.*, of `getSlots` and `getPlayers`) does not contain loops, they are generated by the compiler and are only visible at the EVM level. This also happens, for example, when a function takes a *string* or

Function	solc	opcode bound GASTAP	memory bound GASTAP
`totalBet`	790	775	15
`locked`	706	691	15
`getEndTime`	534	519	15
`slots`	837	822	15
`rewardWinner`	∞	80422+ 5057·**nat**(g3)+5057·**nat**(g1)	18
`Kill`	30883	30874	9
`amountWon`	438	423	15
`getPlayers`	∞	1373+292·**nat**(g1-1/32) +75·**nat**(g1+31/32)	6·**nat**(g1)+24+ $\left\lfloor \frac{(6\cdot\mathbf{nat}(g1)+24)^2}{512} \right\rfloor$
`getSlots`	∞	1507+250·**nat**(g3-1/32) +75·**nat**(g3+31/32)	6·**nat**(g3)+24+ $\left\lfloor \frac{(6\cdot\mathbf{nat}(g3)+24)^2}{512} \right\rfloor$
`winnerAddress`	750	735	15
`__callback`	∞	229380+3·(**nat**(proof)/32) +103·**nat**(result/32) +50·**nat**((32-**nat**(result))) +5836·**nat**(g3)+5057·**nat**(g1) +c(**CALL**1)+c(**CALL**2)+c(**CALL**3)	max_error
`owner`	662	647	15
`endTime`	460	445	15
`potTime`	746	731	15
`potSize`	570	555	15
`joinPot`	∞	no_rf	9
`addresses`	1116	1101	15
`findWinner`	∞	1555+779·**nat**(g3)	15
`random_number`	548	533	15

Figure 4: Gas bounds for `EthereumPot` [18]. Function `nat` defined as `nat(l) = max(0,l).`

bytes variable as argument. This shows another reason for the need of developing the gas analyzer at the
EVM level.

For `joinPot` we cannot ensure that the gas consumption is finite without embedding information about the blockchain in the analyzer. This is because `joinPot` has a loop:

```
for (uint i = msg.value; i >= minBetSize; i−= minBetSize) { tickets++; }
```

where `minBetSize` is a state variable that is initialized in the definition line as `uint minBetSize = 0.01ether`, and `ether` is the value of the *Ether* at the time of executing the instruction. This code has indeed several problems. The first one is that the initialization of the state variable `minBetSize` to the value `0.01ether` does not appear in the
EVM code available in the blockchain. This is because this instruction is executed only once when the contract is created. So our analyzer cannot find this instruction and the value of `minBetSize` is unknown (and hence no bound can be found). Besides, the analyzer cannot infer that the loop terminates if `minBetSize` is not guaranteed to be strictly greater than zero. If we add the

initialization instruction, then we are able to infer a bound for `joinPot`.

For `__callback` we guarantee that the memory gas is *finite* but we cannot obtain an upper-bound for it, GASTAP yields a *maximization error*. Maximization errors may occur when the analyzer needs to compose the cost of the different fragments of the code because it needs to find the maximal value of the cost of inner components in their calling contexts (see [27] for details). If the maximization process involves memory locations that are "unknown", i.e., those translated into a fresh variable in Definition 2 from Section 3, the upper-bound cannot be inferred. Still, if there is no ranking function error, we know that all loops terminate, thus the memory gas consumption is finite.

Finally, this transaction is called always with a constant gas limit of 400,000. This contrasts with the non-constant gas bound obtained using GASTAP. Note that if the gas spent (without including the *refunds*) goes beyond the gas limit the transaction ends with an out-of-gas exception. Since the size of $g3$ and $g1$ is the same as the number of players, from our bound, we can conclude that from 16 players on the contract is in risk of running out-of-gas and get stuck as the 400,000 gas limit cannot be changed. So using GASTAP we can prevent an out-of-gas vulnerability: the contract should not allow more than 15 players, or the gas limit must be increased from that number on.

6.3. Statistics for Analyzed Contracts

Our experimental setup consists on 34,460 contracts taken from the blockchain as follows. We pulled all Ethereum contracts from the blockchain of January 2018 whose Solidity source code was available. This fact reduces the number of smart contracts to be analyzed significantly as it is estimated that less than 1% of the smart contracts deployed on Ethereum blockchain have the source code available. Then, we removed duplicates instances of the same contracts, and after that, we removed those smart contracts that led to a compiler error due to a lower version of the compiler (it introduces changes in the syntax of Solidity that are not supported by the newer versions of the compiler). This ended up in 9,760 files. Each Solidity file often contains several contracts. When the Solidity compiler generates the EVM bytecode of one of these files, it produces one different EVM bytecode file for each contract defined in the Solidity taken as input and GASTAP analyzes each of these EVM bytecode files separately. However, we have excluded the files where the decompilation phase fails in any of the contracts it includes, since in that case we do not get any information on the whole file. This failure happens in 83 files, which represents a 0.85% of the total. The failures of ETHIR are mainly due to recursive and high-order functions, which are not handled by the tool. This number is smaller than the failure rate of other tools like Vandal [15] (5% of failure rate), Oyente [13] (10% of failure rate) and Rattle [28] (30% of failure rate) or the previous prototype of GASTAP [12] (7% of failure rate). Finally, ETHIR has a timeout set to 60s that is reached by 33 files.

After removing these files, our experimental evaluation has been carried out on the remaining 9,644 files, containing the mentioned 34,460 contracts. In total we have analyzed 318,093 public functions (and all auxiliary functions

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

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

Figure 5: (Top) Statistics of gas usage on the analyzed 34,460 smart contracts from Ethereum blockchain. (Bottom) Timing breakdown for GASTAP on the analyzed 34,460 smart contracts.

that are used from them). The Solidity compiler can generate two different EVM bytecode versions: (i) the binary version used when the contract is deployed on the blockchain that includes the code of the constructor of the contracts, and (ii) the runtime version used when the contract has already been deployed, i.e., the code that is actually placed on the blockchain that only includes the bytecode of the functions of the contract excluding the constructor. GASTAP analyzes the runtime version by default so it does not include the code related to the constructors of the contracts.

Experiments have been performed on an Intel Core i7-7700T at 2.9GHz x 8 and 7.7GB of Memory, running Ubuntu 16.04. GASTAP accepts smart contracts written in versions of Solidity up to 0.7.1 or bytecode for the Ethereum Virtual Machine v1.9.20[2]. The statistics that we have obtained in number of functions, and the time taken by the analyzer are summarized in Figure 5. The results for the opcode and memory gas consumption are presented separately.

Let us first discuss the results in Figure 5 which aim at showing the effectiveness of GASTAP. Columns **#opc** and **#mem** contain the number of analyzed functions for opcode and memory gas, resp., and columns preceded by **%** the percentage they represent. For the analyzed contracts, we can see that a large number of functions, 83.75% (resp. 86.44%), have a constant opcode (resp. memory) gas consumption. This is as expected because of the nature of smart contracts, as well as because of the Ethereum safety recommendations

[2]Latest versions released up to September 2020

mentioned in Section 1. Still, there is a relevant number of functions 6.49% (resp. 5.51%) for which we obtain an opcode (resp. memory) gas bound that is not constant (and hence are potentially vulnerable). Additionally, 6.27% of the analyzed functions for opcodes and 5.69% for memory reach the timeout (set to 30 seconds) due to the further complexity of solving the equations. Thanks to the information provided by the SACO analyzer used by GASTAP, we are able to classify the types of errors that have led to a "*don't-know*" answer and which in turn explain the sources of incompleteness by our analysis:

- *Maximization error*: In many cases, a *maximization error* is a consequence of loss of information by the size analysis or by the decompilation when the values of memory locations are lost. As mentioned, even if we do not produce the gas formula, we know that the gas consumption is *finite* (otherwise the system flags a ranking function error described below).

- *Ranking function error:* The solver needs to find ranking functions to bound the maximum number of iterations of all loops the analyzed code might perform. If GASTAP fails at this step, it outputs a *ranking function error*. Section 6 has described a scenario where we have stumbled across this kind of error. We note that number of these failures for **mem** is lower than for **opcode** because when the cost accumulated in a loop is 0, SACO does not look for a ranking function.

- *Cover point error:* The equations are transformed into direct recursive form to be solved [27]. If the transformation is not feasible, a *cover point error* is thrown. This might happen when we have mutually recursive functions, but it also happens for nested loops as in non-structured languages. This is because they contain jump instructions from the inner loop to the outer, and vice versa, and become mutually recursive. A loop extraction transformation would solve this problem, and we leave its implementation for the future work.

Let us discuss in further detail the type of parametric bounds we have obtained. 99% of the parametric opcode gas bounds shown in Figure 5 are linear bounds. In case of the memory gas bounds, the linear bounds are the 97%. Importantly, a state-of-the-art cost solver is needed to automatically infer such linear costs, as this is a very complex problem to automate that, among other things, requires the inference of linear ranking functions [29] that bound the number of loop iterations. In general, such linear ranking functions may involve multiple program variables and some of them can increase through the loop execution, others decrease, or both, etc. Finding out automatically the loop bounds requires the use of static analysis techniques.

Note that GASTAP considers all EVM bytecode instructions within the public function being analyzed when computing the upper bounds, there might be bytecode instructions related to failures (e.g., `REVERT` or `INVALID`) but they are not the reason for having parametricity, since we just accumulate their gas consumption. There are cases in which the EVM bytecode of a contract has

a loop, though it is not visible in its Solidity source code. For instance, these "hidden" loops might come from functions that return arrays or strings, receive them as parameters, the size of the message data, or the length of data structures in storage. As GASTAP analyzes EVM bytecode, it is able to detect these loops and it infers a parametric upper-bound for these functions.

As regards the sources of the parametric bounds, a good percentage correspond to getters of *public* state variables which are strings (such as symbol, name or version). Their getters are almost the 17% of the parametric functions and the 2,32% of the 318,093 public functions analyzed. The getter functions are introduced automatically by the Solidity compiler when the state variables are public. They are standard functions that can be called by any user and their content may be changed by any transaction. In the case of *constant* variables in Solidity (state variables with modifier `constant`), the value has to be a constant at compile time and it has to be assigned when the variable is declared. Then, the value is translated directly when the EVM bytecode is generated, and the upper-bounds inferred are constant. However, GASTAP is not able to detect the cases in which the results inferred rely on state variables that are immutable (those that are not modified by any method of the contract). This kind of constancy is not detected by our analysis, as it would require another (orthogonal) analysis that detects that the state variable is set to a constant in the constructor and ensures that it is never changed again by any other function. This requires an inter-procedural constancy analysis that is complementary to our work. If such constant value is found out, its value can be replaced within the formulas inferred by our tool.

As regards the efficiency of GASTAP, the total analysis time for all functions is 1,467,027.17 sec (407.5 hours). Columns **T** and **%** show, resp., the time in seconds for each phase and the percentage of the total for each type of gas bound. The first three rows are common for the inference of the opcode and memory bounds, while equation generation and solving is separated for opcode and memory. Most of the time is spent in solving the GE (68.43%), which includes some timeouts. The time taken by ETHIR is negligible, as it is a syntactic transformation process, while all other parts require semantic reasoning. All in all, we argue that the statistics from our experimental evaluation show the accuracy, effectiveness and efficiency of our tool. Also, the sources of incompleteness point out directions for further improvements of the tool.

Finally, note that GASTAP works at EVM bytecode level. Thus, the analysis is independent of the version of the compiler used. The same Solidity code compiled with different versions of the compiler might produce different upper-bounds and the optimizations performed by the compiler are orthogonal to our analyzer. In the case of the contracts published in the Ethereum platform, this point is specially interesting because the contracts are published with a particular version of the compiler and cannot be changed.

6.4. Accuracy of GASTAP *when compared to real transactions*

In this section, we assess the accuracy of our tool by comparing the upper-bounds inferred by GASTAP with the gas that real transactions consume on a

Type of result	#opc	%opc	#mem	%mem
Constant gas bound	220	63.22%	255	73.28%
Parametric gas bound	61	17.53%	31	8.91%
Time out	33	9.49%	39	11.21%
Finite gas bound (maximization error)	21	6.03%	23	6.61%
Termination unknown (ranking function error)	1	0.28%	0	0%
Complex control flow (cover point error)	12	3.45%	0	0%
Total number of functions	348	100%	348	100%

Figure 6: Statistics of gas usage on the analyzed 300 most valuable smart contracts whose source code is available from Ethereum blockchain.

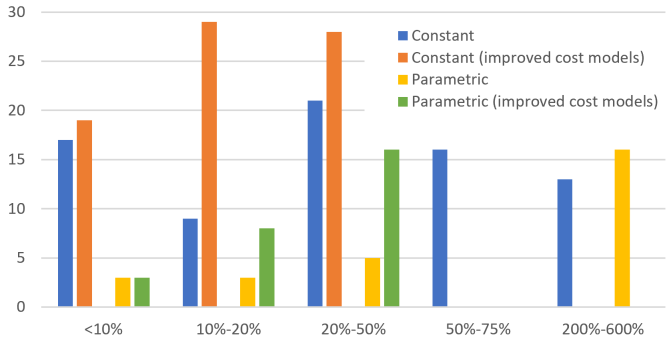


Figure 7: Accuracy of upper bounds inferred by GASTAP compared to real transactions

smaller, but more prominent, set of benchmarks. To do so, we have downloaded the 300 top-valued accounts (those that are most valuable in Ether) whose source code was available using the Etherscan service [26] by 21 September of 2020. This has resulted in 16 Solidity files that contain 24 smart contracts. We have analyzed the 348 public functions contained in these contracts and obtained the results in Figure 6 (the rows have the same meaning explained in Section 6.3). Note that the number of parametric gas upper-bounds obtained for this subset is even higher (17.53%) than for the benchmarks in Section 6.3. For the 125 called, 76 correspond to constant bounds, 27 to parametric bounds, 11 of the functions called return a maximization error, the termination of 1 function could not be proved as the analyzer did not find a ranking function, 3 functions raise a cover point error and 7 functions reach the time out (set to 60s). The number of cover point errors is also higher as one of the contracts analyzed has various public functions that have three nested loops with a conditional statement inside that requires the loop extraction transformation mentioned in Section 6.3 that is not yet implemented in the SACO analyzer.

Now, in order to assess the accuracy, we have to restrict ourselves to those functions that have been invoked by existing transactions. This information can be retrieved using the Etherscan service [26] and Bloxy service [30]. From

the 348 public functions analyzed, 125 (almost a 37%) were called in 4054 real executions. In order to compare the bounds inferred by GASTAP and the gas consumed by real executions, we have to add the transaction fee (21,000) to the GASTAP bounds as it is not added by default. We compute the average of the gas consumed by each transaction to each function and compare our upper bound to such average. Figure 7 summarizes our results. We have grouped the analyzed functions in five groups: those whose upper bound has an overhead <10% (leftmost group), those whose upper bound has an overhead between 10-20%,.., until the rightmost group with an overhead between 200-600%. Within each group, we separate the results of those functions that have a constant upper bounds from those that have a parametric one. We use also two cost models: the cost model described in the paper that is sound in all possible scenarios, and one (denoted as "improved cost model" in the figure) that captures better the state of the blockchain when these transactions were executed by accounting a less pessimistic gas consumption for certain bytecodes, namely these ones:

b	$op\_cost(pp{:}b)$
`SSTORE`	5000

`CALL`
`DELEGATECALL`
`CALLCODE`
`STATICCALL`
	$700 + C_{\text{XFER}}$
	$C_{\text{XFER}} \equiv \begin{cases} 9000 & \text{if} \quad getVal(pp, s_{n-3}) \neq 0 \\ 0 & \text{otherwise} \end{cases}$

`SELFDESTRUCT`	5000

In this improved cost model, instead of assuming that the account involved in `SELFDESTRUCT` operation does not exist or is "dead"(worst case, see [1]), considers that it is active. Thus, it would only consume 5,000 units of gas instead of 30,000. In the case of `SSTORE` bytecode, we assume that the storage location involved in the operation has been previously used and it does not contain a 0. Thus, it would consume 5,000 units of gas instead of 20,000. Finally, for the opcodes related to external calls (`CALL`, `DELEGATECALL`, `CALLCODE` and `STATICCALL`), we assume that the contract called has been previously created. Thus, it saves 25,000 units of gas compared with the sound model.

As an example, the leftmost orange bar in Figure 7 is displaying the number of functions (namely 19) that have a constant cost using the improved cost model above such that the overhead of the upper bound obtained for them is less than 10% w.r.t. the average of the gas consumed by the actual transactions invoking this functions.

We can observe in Figure 7 that using the sound cost model, the upper-bounds inferred for more than half of the functions is are at least 50% higher than the real amount of gas consumed by the transactions. However, the results are improved notably when we assume the improved cost model, indeed, all constant gas bounds disappear from the last two groups in which the overhead is larger. We have still 16 functions in the rightmost group, that have a parametric bound, for which the overhead of the upper bound is large. This is not surprising as the actual transactions run on concrete data while our upper bounds are covering all

possible input data values. An important result of these experiments is that 132 out of the 4054 analyzed (3.64%) raise an *out-of-gas* exception. In these cases, GASTAP infers upper-bounds that are higher than the gas limit specified in the corresponding transactions. Thus, our bounds would have helped to prevent such exceptions. In addition, there are 7 functions that raise an *out-of-gas* exception in the transactions in which they are involved.

Finally, we have made a manual inspection of the functions in order to explain the sources of the parametricity of the functions shown in Figure 6. We can classify them as follows:

- 11.48% of the parametric bounds depend on input data, i.e., on a value that is passed to the function as a parameter.
- 16.39% of the parametric bounds depend on the length of the message data. In this case, the bound is expressed in terms of `CALLDATASIZE`.
- 67.21% come from traversing state variables that are arrays. In this case, the loop is explicit in the code in a 79% of the times. However, for the rest 21% it appears when the source code is translated into EVM code (e.g. when using the primitive `delete`).
- 4.91% correspond to getters of *public* state variables that are strings. The bounds are expressed in terms of the length of the string returned.

7. Related Work

Analysis of Ethereum smart contracts for possible safety and security violations and vulnerabilities is a popular topic that has received a lot of attention recently, with numerous tools developed, leveraging techniques based on symbolic execution [31, 32, 33, 34, 35, 36], SMT solving [37, 38, 39], and certified programming [40, 41, 42]. Those approaches target vulnerabilities induced by contract-specific phenomena such as *reentrancy* [32], mishandled control flow [36], as well as trace-based properties [33, 38]. However, most of the state-of-the art contract analysis tools ignore resource usage, focusing on non-gas-related safety, security, and temporal properties. As GASTAP is not meant to be an *all-in-one* smart contract analyzer and focuses exclusively on gas consumption, in this survey we only relate to the tools and approaches that are concerned with gas usage and the corresponding *out-of-gas* vulnerabilities.

The GASPER tool identifies gas-costly programming patterns [43], which can be optimized to consume less. For doing so, it relies on matching specific control-flow patterns, SMT solvers and symbolic computation, which makes their analysis neither sound, nor complete for determining gas usage bounds. They classify the patterns in two groups: useless-code related patterns and loop-related patterns. These techniques focus on detecting code patterns to improve the development of the contract rather than inferring gas consumption bounds.

The previous work is extended in [44] and [45]. In [44] they present GASREDUCER, a tool that also works at EVM bytecode level. GASREDUCER takes the

bytecode of a smart contract and outputs an optimized version of the original one that consumes less gas. In this case the patterns are defined as a sequence of EVM instructions that can be replaced by another one that consumes less gas but has the same semantics as the original sequence rather than using high-level structures such as dead code or opaque code. To identify the patterns, they inspect several instances from the execution traces of deployed smart contracts. In [45] the tool GasChecker is presented. It follows a new approach to identify gas-inefficient code based on symbolic execution parallelized using a MapReduce model and cloud computing. They increase the patterns described in [43]. They also improve the performance of the tool making it fully scalable as it is shown in the experiments.

In a similar vein, the work by Grech *et al.* [4] identifies a number of classes of gas-focused vulnerabilities, and provides MadMax, a static analysis, also working on a decompiled EVM bytecode, by combining techniques from flow analysis together with control-flow analysis (CFA), context-sensitive analysis and modeling of memory layout. In a first step, MadMax infers loop and data-flow information. From loops, it infers information related to the exit condition of the loop or induction variables, i.e., those that are incremented by a concrete value inside the loop. The data-flow analysis provides information such as aliasing or dependencies between variables. Using the basic loop and data-flow analysis, MadMax is able to infer high-level concepts such as array iterators or if the storage is increased on public functions. In its techniques, MadMax differs from Gastap, as it focuses on identifying control- and data-flow patterns inherent for the gas-related vulnerabilities, thus, working as a bug-finder, rather than complexity analyzer.

Since deriving accurate worst-case complexity boundaries is not a goal of any of GASPER, GasReducer, GasChecker and MadMax, they are unsuitable for tackling Challenge (1) posed in the introduction. Other tools based on proof assistants [46, 47, 48, 41] may be used to detect out-of-gas exceptions. They can model how gas is updated along the execution of the trace and encode it as constraint formulas. However these tools are not able to infer loop invariants. They have to be specified manually. Thus, they are not able no automatically infer gas bounds for programs that involve loops as we do.

A preliminary prototype of Gastap has been used in [49] to develop Gasol, a gas optimizer of smart contracts. Gasol uses the gas analysis Gastap to detect under-optimized storage patterns and generate an automatic optimization that avoids storage accesses if they can be replaced by memory accesses. For this purpose, it includes a simplified version of our gas cost model that only accounts for the gas cost of storage operations.

Marescotti *et al.* [37] propose a methodology, based on the notion of the so-called *gas consumption paths* (GCPs) to estimate the worst-case gas consumption using techniques from symbolic bounded model checking [50]. Their approach is based on symbolically enumerating all execution paths and unwinding loops to a limit. In contrast, Gastap infers the maximal number of iterations for loops and generates accurate gas bounds which are sound for any possible execution of the function and not only for the unwound paths. While

their approach unwinds loops to a given limit, using a resource analysis approach, we are able to infer the maximal number of iterations that loops may execute. As we have seen, using a resource analysis approach, in addition to inferring precise cost expressions for constant gas consumption as in [37], we can go beyond that, and generate parametric gas bounds. Besides, to the best of our knowledge, the approach by Marescotti *et al.* has not been implemented in the context of EVM and has not been evaluated on real-world smart contracts as ours. Therefore, we have not been able to compare the results generated by our tool with the approach proposed in [37]

In [51], Visualgas is presented, a tool to visualize gas costs. Visualgas records the deployment and initialization transactions of a given contract and uses them to generate transactions to run all its public functions applying the techniques described in [52, 53]. This approach differs from our analysis in that it is dynamic and hence it is focused on computing the gas cost of concrete transactions that have been generated previously. On the other hand, our technique is static and infers the gas cost consumed in the worst case for any arbitrary input of the public function under analysis.

8. Conclusions and Future Work

Automated sound static reasoning about resource consumption is critical for developing safe and secure blockchain-based replicated computations, managing billions of dollars worth of virtual currency. In this work, we have adapted and extended state-of-the art techniques in resource analysis, showing that such reasoning is feasible for Ethereum, where it can be used at scale not only for preventing vulnerabilities, but also for verification/certification of existing smart contracts. Note that improvements in all the auxiliary analyses used in GASTAP will have an impact in the precision and the performance of the tool. As future work, we want to improve the accuracy of the tool in several directions. First, we are studying a more precise abstraction for the memory-allocated data (*i.e.*, for the abstraction explained in Section 3). Also, we aim at handling bit-wise operations by including in our tool an abstraction for them.

Acknowledgements

This work was funded partially by the Spanish the Spanish MCIU, AEI and FEDER (EU) project RTI2018-094403-B-C31 and RTI2018-094403-B-C33, by the CM projects P2018/TCS-4314 and S2018/TCS-4339 co-funded by EIE Funds of the European Union and by the UCM CT27/16-CT28/16 grant.

References

[1] G. Wood, Ethereum: A secure decentralised generalised transaction ledger (2014).

[2] Ethereum, Solidity, https://solidity.readthedocs.io (2018).

[3] Ethereum, Vyper, https://vyper.readthedocs.io (2018).

[4] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, Y. Smaragdakis, Madmax: surviving out-of-gas conditions in ethereum smart contracts, PACMPL 2 (OOPSLA) (2018) 116:1–116:27.

[5] E. Foundation, Safety - Ethereum Wiki, https://github.com/ethereum/wiki/wiki/Safety, last accessed on 14 November 14 2018 (2018).

[6] E. Albert, J. Correas, G. Román-Díez, Non-Cumulative Resource Analysis, in: Proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015, Vol. 9035 of Lecture Notes in Computer Science, Springer, 2015, pp. 85–100.

[7] J. Hoffmann, K. Aehlig, M. Hofmann, Multivariate amortized resource analysis, in: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January, 2011, 2011, pp. 357–370.

[8] B. Wegbreit, Mechanical program analysis, Commun. ACM 18 (9) (1975) 528–539.

[9] M. Suiche, Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode (2017).

[10] T. Bernani, Oraclize, http://www.oraclize.it (2016).

[11] E. Albert, P. Gordillo, B. Livshits, A. Rubio, I. Sergey, EthIR: A Framework for High-Level Analysis of Ethereum Bytecode, in: S. Lahiri, C. Wang (Eds.), 16th International Symposium on Automated Technology for Verification and Analysis, ATVA 2018. Proceedings, Vol. 11138 of Lecture Notes in Computer Science, Springer, 2018, pp. 513–520.

[12] E. Albert, P. Gordillo, A. Rubio, I. Sergey, Running on Fumes: Preventing Out-Of-Gas Vulnerabilities in Ethereum Smart Contracts using Static Resource Analysis, in: P. Ganty, M. Kaâniche (Eds.), 13th International Conference on Verification and Evaluation of Computer and Communication Systems, VECoS 2019. Proceedings, Vol. 11847 of Lecture Notes in Computer Science, 2019, pp. 63–78.

[13] Oyente: An Analysis Tool for Smart Contracts, https://github.com/melonproject/oyente (2018).

[14] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[15] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, B. Scholz, Vandal: A Scalable Security Analysis Framework for Smart Contracts, arXiv:1809.03981 (2018).

[16] N. Grech, L. Brent, B. Scholz, Y. Smaragdakis, Gigahorse: thorough, declarative decompilation of smart contracts, in: J. M. Atlee, T. Bultan, J. Whittle (Eds.), Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE / ACM, 2019, pp. 1176–1186.

[17] E. Albert, J. Correas, P. Gordillo, A. Hernández-Cerezo, G. Román-Díez, A. Rubio, Analyzing Smart Contracts: From EVM to a Sound Control-Flow Graph, Tech. rep. (2020).

[18] The EthereumPot contract, https://etherscan.io/address/0x5a13caa82851342e14cd2ad0257707cddb8a31b7 (2017).

[19] E. Albert, P. Arenas, S. Genaim, G. Puebla, G. Román-Díez, Conditional Termination of Loops over Heap-allocated Data, Science of Computer Programming 92 (2014) 2 – 24.

[20] L. O. Andersen, Program analysis and specialization for the c programming language, Ph.D. thesis, University of Cophenhagen (1994).

[21] B. Steensgaard, Points-to analysis in almost linear time, in: H. Boehm, G. L. S. Jr. (Eds.), Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Florida, USA, January, 1996, ACM Press, 1996, pp. 32–41.

[22] S. H. Yong, S. Horwitz, Pointer-range analysis, in: R. Giacobazzi (Ed.), Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings, Vol. 3148 of Lecture Notes in Computer Science, Springer, 2004, pp. 133–148.

[23] E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, G. Román-Díez, Object-Sensitive Cost Analysis for Concurrent Objects, STVR 25 (3) (2015) 218–271.

[24] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost Analysis of Object-Oriented Bytecode Programs, Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages) 413 (1) (2012) 142–159.

[25] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, J. Giesl, Analyzing runtime and size complexity of integer programs, ACM Trans. Program. Lang. Syst. 38 (4) (2016) 13:1–13:50.

[26] Etherscan, https://etherscan.io (2018).

[27] E. Albert, P. Arenas, S. Genaim, G. Puebla, Automatic inference of upper bounds for recurrence relations in cost analysis, in: M. Alpuente, G. Vidal (Eds.), Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings, Vol. 5079 of Lecture Notes in Computer Science, Springer, 2008, pp. 221–237.

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

[29] E. Albert, P. Arenas, S. Genaim, G. Puebla, Closed-Form Upper Bounds in Static Cost Analysis, Journal of Automated Reasoning 46 (2) (2011) 161–203.

[30] Bloxy, https://bloxy.info/ (2018).

[31] L. Luu, D. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October, 2016, ACM, 2016, pp. 254–269.

[32] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, Y. Zohar, Online detection of effectively callback free objects with applications to smart contracts, PACMPL 2 (POPL) (2018) 48:1–48:28.

[33] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, A. Hobor, Finding the greedy, prodigal, and suicidal contracts at scale, in: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December, 2018, ACM, 2018, pp. 653–663.

[34] J. Krupp, C. Rossow, teether: Gnawing at ethereum to automatically exploit smart contracts, in: USENIX Security Symposium, USENIX Association, 2018, pp. 1317–1333.

[35] S. Kalra, S. Goel, M. Dhawan, S. Sharma, ZEUS: analyzing safety of smart contracts, in: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February, 2018, The Internet Society, 2018.

[36] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, M. T. Vechev, Securify: Practical security analysis of smart contracts, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October, 2018, ACM, 2018, pp. 67–82.

[37] M. Marescotti, M. Blicha, A. E. J. Hyvärinen, S. Asadi, N. Sharygina, Computing Exact Worst-Case Gas Consumption for Smart Contracts, in: Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November, 2018, Proceedings, Part IV, Vol. 11247 of Lecture Notes in Computer Science, Springer, 2018, pp. 450–465.

[38] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, P. Saxena, Exploiting The Laws of Order in Smart Contracts, CoRR abs/1810.11605. arXiv:1810.11605.

[39] Á. Hajdu, D. Jovanovic, Smt-friendly formalization of the solidity memory model, in: P. Müller (Ed.), Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Dublin, Ireland, Proceedings, Vol. 12075 of Lecture Notes in Computer Science, Springer, 2020, pp. 224–250.

[40] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, S. Zanella-Béguelin, Formal verification of smart contracts: Short paper, in: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 2016, ACM, 2016, pp. 91–96.

[41] I. Grishchenko, M. Maffei, C. Schneidewind, A Semantic Framework for the Security Analysis of Ethereum Smart Contracts, in: Principles of Security and Trust - 7th International Conference, POST 2018, Thessaloniki, Greece. Proceedings, Vol. 10804 of Lecture Notes in Computer Science, Springer, 2018, pp. 243–269.

[42] S. Amani, M. Bégel, M. Bortin, M. Staples, Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL, in: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, 2018, ACM, 2018, pp. 66–77.

[43] T. Chen, X. Li, X. Luo, X. Zhang, Under-optimized smart contracts devour your money, in: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE Computer Society, 2017, pp. 442–446.

[44] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, X. Zhang, Towards saving money in using smart contracts, in: A. Zisman, S. Apel (Eds.), Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, 2018, ACM, 2018, pp. 81–84.

[45] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, X. Zhang, Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts, IEEE Transactions on Emerging Topics in Computing PP(99) (2020) 1–14.

[46] Y. Hirai, Defining the Ethereum Virtual Machine for Interactive Theorem Provers, 2017.

[47] A. Mavridou, A. Laszka, Designing secure ethereum smart contracts: A finite state machine based approach, CoRR abs/1711.09327.

[48] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, D. Park, Y. Zhang, B. Moore, G. Rosu, KEVM: A Complete Semantics of

the Ethereum Virtual Machine, in: 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, 2018, IEEE Computer Society, 2018, pp. 204–217.

[49] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, A. Rubio, GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts, in: A. Biere, D. Parker (Eds.), Proceedings of 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2020, Vol. 12079 of Lecture Notes in Computer Science, 2020, pp. 118–125.

[50] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu, Symbolic model checking without bdds, in: Proceedings of 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 1999, Vol. 1579 of Lecture Notes in Computer Science, Springer, 1999, pp. 193–207.

[51] C. Signer, Gas Cost Analysis for Ethereum Smart Contracts, Master's thesis, Swiss Federal Institute of Technology Zurich, Switzerland (2018).

[52] N. Ambroladze, Fast and scalable analysis of smart contracts, Master's thesis, Swiss Federal Institute of Technology Zurich, Switzerland (2018).

[53] J. He, M. Balunovic, N. Ambroladze, P. Tsankov, M. T. Vechev, Learning to fuzz from symbolic execution with application to smart contracts, in: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, 2019, ACM, 2019, pp. 531–548. doi:10.1145/3319535.3363230.