Running on Fumes*

Preventing Out-of-Gas Vulnerabilities in Ethereum Smart Contracts using Static Resource Analysis

Elvira Albert[1], Pablo Gordillo[1], Albert Rubio[1], and Ilya Sergey[2]

[1] Complutense University of Madrid, Spain
[2] Yale-NUS College and School of Computing, NUS, Singapore

Abstract. Gas is a measurement unit of the computational effort that it will take to execute every single operation that takes part in the Ethereum blockchain platform. Each instruction executed by the Ethereum Virtual Machine (EVM) has an associated gas consumption specified by Ethereum. If a transaction exceeds the amount of gas allotted by the user (known as gas limit), an *out-of-gas* exception is raised. There is a wide family of contract vulnerabilities due to *out-of-gas* behaviors. We report on the design and implementation of GASTAP, a Gas-Aware Smart contracT Analysis Platform, which takes as input a smart contract (either in EVM, disassembled EVM, or in Solidity source code) and automatically infers gas upper bounds for all its public functions. Our bounds ensure that if the gas limit paid by the user is higher than our inferred gas bounds, the contract is free of out-of-gas vulnerabilities.

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* [29], a monetary value in *Ether*, Ethereum's currency, paid by a transaction-proposing party. Computations (performed by invoking smart contracts) that require *more computational or storage resources*, cost more gas than those that require fewer resources. As regards storage, the EVM has three areas where it can store items: 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; the *memory* is used to hold temporary values, and it is erased between transactions and is cheaper to use; 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'

* This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R and MINECO/FEDER, UE project TIN2015-69175-C4-3-R, by Spanish MICINN/FEDER, UE projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, by the CM project S2018/TCS-4314 and by the UCM CT27/16-CT28/16 grant.

(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).

In general, the gas-aware operational semantics of EVM has introduced novel challenges *wrt.* sound static reasoning about resource consumption, correctness, and security of replicated computations: (1) While the EVM specification [29] provides the precise gas consumption of the low-level operations, most of the smart contracts are written in high-level languages, such as Solidity [13] or Vyper [14]. 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. (2) As noted in [17], it is discouraged in the Ethereum safety recommendations [16] that the gas consumption of smart contracts depends 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 functions we have analyzed do. The inability to estimate those dependencies, and the lack of analysis tools, leads 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 perform any operation within a reasonable gas bound. Those vulnerabilities have been recognized before, but only discovered by means of unsound, pattern-based analysis [17].

In this paper, we address these challenges in a principled way by developing GASTAP, a *Gas-Aware Smart contracT Analysis Platform*, which is, to the best of our knowledge, the first automatic gas analyzer for smart contracts. GASTAP takes as input a smart contract provided in Solidity source code [13], or in low-level (possibly decompiled [26]) EVM code, and automatically infers an upper bound on the gas consumption for each of its public functions. The upper bounds that GASTAP infers 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).

The inference of gas requires complex transformation and analysis processes on the code that include: (1) construction of the control-flow graphs (CFGs), (2) decompilation from low-level code to a higher-level representation, (3) inference of size relations, (4) generation of gas equations, and (5) solving the equations into closed-form gas bounds. Therefore, building an automatic gas analyzer from EVM code requires a daunting implementation effort that has been possible thanks to the availability of a number of existing open-source tools that we have succeeded to extend and put together in the GASTAP system. In particular, an extension of the tool OYENTE [3] is used for (1), an improved representation of

EthIR [6] is used for (2), an adaptation of the size analyzer of Saco [4] is used to infer the size relations, and the Pubs [5] solver for (5).

The most challenging aspect in the design of Gastap has been the approximation of the EVM gas model (which is formally specified in [29]) that is required to produce the gas equations in step (4). 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 [29]), 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 component is also complex to infer because it is not always a constant amount, it might depend in some cases on the current global and local state.

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. As contract developers and owners, having a precise resource analyzer allows answering 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 us to calculate the safe amount of gas that one should provide to an external data source (e.g., contracts using Oraclize[8]) in order to enable a successful callback. As an attacker, one might estimate, how much *Ether* (in gas), an adversary has to pour into a contract in order to execute the DoS attack. We note that such an attack may, however, be economically impractical.

Finally, we argue that our experimental evaluation shows that Gastap is an effective and efficient tool: we have analyzed more than 29,000 real smart contracts pulled from etherscan.io [2], that in total contain 258,541 public functions, and inferred gas bounds for 91.85% of them in 342.54 hours. Gastap can be used from a web interface at `https://costa.fdi.ucm.es/gastap`.

2 Description of Gastap Components

Figure 1 depicts the architecture of Gastap. In order to describe all components of our tool, we use as running example a simplified version (without calls to the external service Oraclize and the authenticity proof verifier) of the `EthereumPot` contract [1] that implements a simple lottery. During a game, players call a method `joinPot` to buy lottery tickets; each player's address is appended to an array `addresses` of current players, and the number of tickets is appended to an array `slots`, both having variable length. After some time has elapsed, anyone can call `rewardWinner` which calls the `Oraclize` service to obtain a random number for the winning ticket. If all goes according to plan, the `Oraclize` service then responds by calling the `__callback` method with this random number and the authenticity proof as arguments. A new instance of the game is then started, and the winner is allowed to withdraw her balance using a `withdraw` method. In

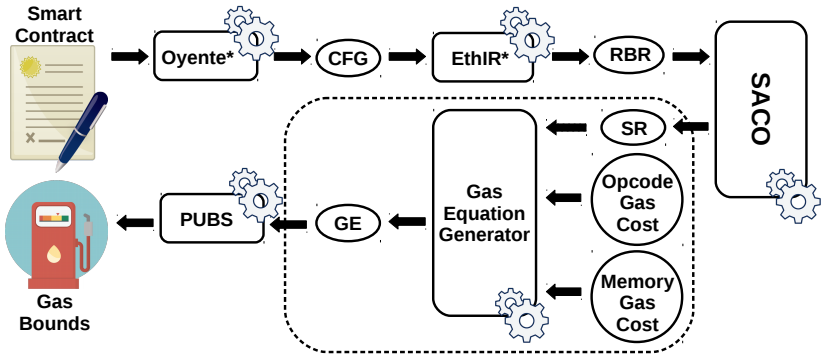


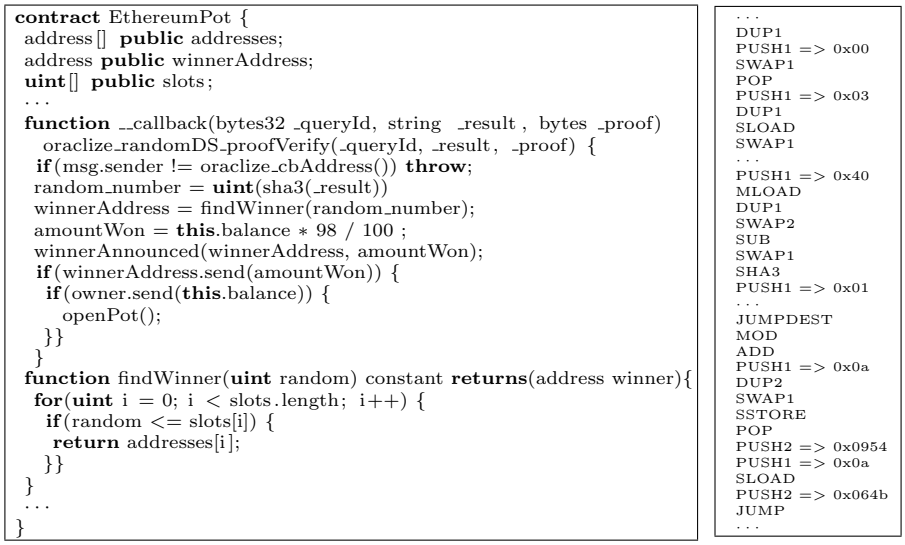
Fig. 1. Architecture of GASTAP (CFG: control flow graph; RBR: rule-based representation; SR: size-relations; GE: gas equations)

Fig. 2, an excerpt of the Solidity code (including the public function `findWinner`) and a fragment of the EVM code produced by the compiler, are displayed. The Solidity source code is showed for readability, as GASTAP analyzes directly the EVM code (if it receives the source, it first compiles it to obtain the EVM code).

2.1 Oyente*: from EVM to a complete CFG

The first component of our tool, OYENTE*, is an extension of the open-source tool OYENTE [3], a symbolic execution tool developed to analyze Ethereum smart contracts and find potential security bugs. As OYENTE's aim is on symbolic execution rather than on generating a complete CFG, some extensions are needed to this end. The EthIR framework [6] had already extended OYENTE for two purposes: (1) to recover the list of addresses for unconditional blocks with more than one possible jump address (as OYENTE originally only kept the last processed one), and (2) to add more explicit information to the CFG: jump operations are decorated with the jumping address, discovered by OYENTE, and, other operations like store or load are also decorated with the address they operate: the number of state variable for operations on storage; and the memory location for operations on memory if OYENTE is able to discover it (or with "?" otherwise).

However EthIR's extension still produced incomplete CFGs. OYENTE* further extends it to handle a more subtle source of incompleteness in the generated CFG that comes directly from the fact that OYENTE is a symbolic execution engine. For symbolic execution, a bound on the number of times a loop is iterated is given. Hence it may easily happen that some (feasible) paths are not reached in the exploration within this bound and they are lost. To solve this problem, we have modified OYENTE to remove the execution bound (as well as other checks that were only used for their particular applications), and have added information to the path under analysis. Namely, every time a new jump is found, we check if the jumping point is already present in the path. In such case, an edge to that point is added and the exploration of the trace is stopped. As a side effect, we not only produce a complete CFG, but also avoid much useless exploration for our purposes which results in important efficiency gain.



```
contract EthereumPot {
 address[] public addresses;
 address public winnerAddress;
 uint[] public slots;
 ...
 function __callback(bytes32 _queryId, string _result, bytes _proof)
   oraclize_randomDS_proofVerify(_queryId, _result, _proof) {
  if(msg.sender != oraclize_cbAddress()) throw;
  random_number = uint(sha3(_result))
  winnerAddress = findWinner(random_number);
  amountWon = this.balance * 98 / 100 ;
  winnerAnnounced(winnerAddress, amountWon);
  if(winnerAddress.send(amountWon)) {
   if(owner.send(this.balance)) {
     openPot();
  }}
 }
 function findWinner(uint random) constant returns(address winner){
  for(uint i = 0; i < slots.length; i++) {
   if(random <= slots[i]) {
    return addresses[i];
  }}
 }
 ...
}
```

```
...
DUP1
PUSH1 => 0x00
SWAP1
POP
PUSH1 => 0x03
DUP1
SLOAD
SWAP1
...
PUSH1 => 0x40
MLOAD
DUP1
SWAP2
SUB
SWAP1
SHA3
PUSH1 => 0x01
...
JUMPDEST
MOD
ADD
PUSH1 => 0x0a
DUP2
SWAP1
SSTORE
POP
PUSH2 => 0x0954
PUSH1 => 0x0a
SLOAD
PUSH2 => 0x064b
JUMP
...
```

Fig. 2. Excerpt of Solidity code for `EthereumPot` contract (left), and fragment of EVM code for function `__callback` (right)

When applying Oyente*, our extended/modified version of Oyente, we obtain a *complete* CFG, with the additional annotations already provided by [6].

2.2 EthIR*: from CFG to an annotated rule-based representation

EthIR*, an extension of EthIR [6], is the next component of our analyzer. EthIR provides a rule-based representation (RBR) for the CFG obtained from Oyente*. Intuitively, for each block in the CFG it generates a corresponding rule that contains a high-level representation of all bytecode instructions in the block (e.g., load and store operations are represented as assignments) and that has as parameters an explicit representation of the stack, local, state, and blockchain variables (details of the transformation are in [6]). Conditional branching in the CFG is represented by means of guards in the rules. EthIR* provides three extensions to the original version of EthIR [6]: (1) The first extension is related to the way function calls are handled in the EVM, where instead of an explicit `CALL` opcode, as we have seen before, a call to an internal function is transformed into a `PUSH` of the return address in the stack followed by a `JUMP` to the address where the code of the function starts. If the same function is called from different points of the program, the resulting CFG shares for all these calls the same subgraph (the one representing the code of the function) which ends with different jumping addresses at the end. As described in [17], there is a need to clone parts of the CFG to explicitly link the `PUSH` of the return address with the final `JUMP` to this address. This cloning in our implementation is done at the level of the RBR as follows: Since the jumping addresses are known thanks to the symbolic execution applied by Oyente, we can find the

connection between the `PUSH` and the `JUMP` and clone the involved part of the RBR (between the rule of the `PUSH` and of the `JUMP`) using different rule names for each cloning. (2) The second extension is a flow analysis intended to reduce the number of parameters of the rules of the RBR. This is crucial for efficiency as the number of involved parameters is a bottleneck for the successive analysis steps that we are applying. Basically, before starting the translation phase, we compute the inverse connected component for each block of the CFG, i.e, the set of its predecessor blocks. During the generation of each rule, we identify the local, state or blockchain variables that are used in the body of the rule. Then, these variables have to be passed as arguments only to those rules built from the blocks of its inverse connected component. (3) When we find a store on an unknown memory location "?", we have to "forget" all the memory from that point on, since the writing may affect any memory location, and it is not sound anymore to assume the previous information. In the RBR, we achieve this deletion by assigning fresh variables (thus unknown values) to the memory locations at this point.

Optionally, EthIR provides in the RBR the original bytecode instructions (from which the higher-level ones are obtained) by simply wrapping them within a nop functor (see Fig. 3). Although nop annotations will be ignored by the size analysis, they are needed later to assign a precise gas consumption to every rule.

$$
\begin{array}{l}
block1647(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow \\
\quad nop(JUMPDEST), s_{11} = s_9, s_9 = s_{10}, s_{10} = s_{11}, nop(SWAP), s_{11} = 0, nop(PUSH), \\
\quad \underline{l_2 = s_{10}}, nop(MSTORE), s_{10} = 32, nop(PUSH), s_{11} = 0, nop(PUSH), \underline{s_{10} = sha3(s_{11}, s_{10})}, \\
\quad \underline{nop(SHA3)}, s_9 = s_{10} + s_9, \underline{nop(ADD)}, \underline{gl} = s_9, s_9 = fresh_0, \underline{nop(SLOAD)}, s_{10} = s_6, \\
\quad nop(DUP4), call(jump1647(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc})), nop(GT), nop(ISZERO), nop(ISZERO), \\
\quad nop(PUSH), nop(JUMPI)
\end{array}
$$

Fig. 3. Selected rule including nop functions needed for gas analysis

Example 1. Figure 3 shows the RBR for *block1647*. Bytecode instructions that load or store information are transformed into assignments on the involved variables. For arithmetic operations, operations on bits, sha, etc., the variables they operate on are made explicit. Since stack variables are always consecutive we denote by $\overline{s_n}$ the decreasing sequence of all s_i form n down to 0. $\overline{lv}$ includes l_2 and l_0, which is the subset of the local variables that are needed in this rule or in further calls (second extension of EthIR*). The unknown location "?" has become a fresh variable $fresh_0$ in *block1647*. For state variables, $\overline{sv}$ includes the needed ones $g_{11}, g_8, g_7, g_6, g_5, g_3, g_2, g_1, g_0$ (g_i is the i-th state variable). Finally, $\overline{bc}$ includes the needed blockchain state variables `address`, `balance` and `timestamp`.

2.3 SACO: size relations for EVM smart contracts

In the next step, we generate *size relations* (SR) from the RBR using the Saco tool [4]. SR are equations and inequations that state how the sizes of data change in the rule [12]. This information is obtained by analyzing how each instruction of the rules modifies the sizes of the data it uses, and propagating this information as usual in dataflow analysis. SR are needed to build the gas equations and then generate gas bounds in the last step of the process. The size analysis of Saco

has been slightly modified to ignore the *nop* instructions. Besides, before sending the rules to SACO, we replace the instructions that cannot be handled (e.g., bitwise operations, hashes) by assignments with fresh variables (to represent an unknown value). Apart from this, we are able to adjust our representation to make use of the approach followed by SACO, which is based on abstracting data (structures) to their sizes. For integer variables, the size abstraction corresponds to their value and thus it works directly. However, a language specific aspect of this step is the handling of data structures like array, string or bytes (an array of byte). In the case of array variables, SACO's size analysis works directly as in EVM the slot assigned to the variable contains indeed its length (and the address where the array content starts is obtained with the hash of the slot address).

Example 2. Consider the following SR (those in brackets) generated for rule *jump1649* and *block1731*:
$jump1619(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) = block1633(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc})\{s_{10} < s_9\}$
$block1731(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) = 41 + block1619(s'_8, \overline{s_7}, \overline{sv}, \overline{lv}, \overline{bc})\{s'_8 = 1 + s_8\}$
The size relations for the *jump1619* function involve the `slots` array length (g_3 stored in s_9) and the local variable `i` (in s_8 and copied to s_{10}). It corresponds to the guard of the `for` loop in function `findWinner` that compares `i` and `slots.length` and either exits the loop or iterates (and hence consume different amount of gas). The size relation on s_8 for *block1731* corresponds to the size increase in the loop counter.

However, for bytes and string it is more challenging, as the way they are stored depends on their actual sizes. Roughly, if they are short (at most 31 bytes long) their data is stored in the same slot together with its length. Otherwise, the slot contains the length (and the address where the string or bytes content starts is obtained like for arrays). Our approach to handle this issue is as follows. In the presence of bytes or string, we can find in the rules of the RBR a particular sequence of instructions (which are always the same) that start pushing the contents of the string or bytes variable in the top of the stack, obtain its length, and leave it stored in the top of the stack (at the same position). Therefore, to avoid losing information, since SACO is abstracting the data structures to their sizes, every time we find this pattern of instructions applied to a string or bytes variable, we just remove them from the RBR (keeping the nops to account for their gas). Importantly, since the top of the stack has indeed the size, under SACO's abstraction it is equal to the string or bytes variable. Being precise, assuming that we have placed the contents of the string or bytes variable in the top of the stack, which is s_i, the transformation applied is the following:

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

$$\Downarrow$$

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

Since the involved instructions include bit-wise operations among others and, as said, the value of the stack variable becomes unknown, without this transformation the relation between the stack variable and the length of the string or bytes would be lost and, as a result, the tool may fail to provide a bound on the gas consumption. This transformation is applied when possible and, e.g., is needed to infer bounds for the functions `getPlayers` and `getSlots` (see Table 3.2).

2.4 Generation of equations

In order to generate gas equations (GE), we need to define the EVM gas model, which is obtained by encoding the specification of the gas consumption for each EVM instruction as provided in [29]. The EVM gas model is complex and unconventional, it has two components, one which is related to the memory consumption, and another one that depends on the bytecode executed. The first component is computed separately as will be explained below. In this section we focus on computing the gas attributed to the opcodes. For this purpose, we provide a function $C_{opcode} : s \mapsto g$ which, for an EVM opcode, takes a stack s and returns a gas g associated to it. We distinguish three types of instructions: (1) Most bytecode instructions have a *fixed* constant gas consumption that we encode precisely in the cost model C_{opcode}, i.e., g is a constant. (2) Bytecode instructions that have different *constant* gas consumption g_1 or g_2 depending on some given condition. This is the case of `SSTORE` that costs $g_1 = 20000$ if the storage value is set from zero to non-zero (first assignment), and $g_2 = 5000$ otherwise. But it is also the case for `CALL` and `SELFDESTRUCT`. In these cases we use $g = max(g_1, g_2)$ in C_{opcode}. (3) Bytecode instructions with a non-constant (*parametric*) gas consumption that depends on the value of some stack location. For instance, the gas consumption of `EXP` is defined as $10 + 10 \cdot (1 + \lfloor log_{256}(\mu_s[1]) \rfloor)$ if $\mu_s[1] \neq 0$ where $\mu_s[0]$ is the top of the stack. Therefore, we have to define g in C_{opcode} as a parametric function that uses the involved location. Other bytecode instructions with parametric cost are `CALLDATACOPY`, `CODECOPY`, `RETURNDATACOPY`, `CALL`, `SHA3`, `LOG*`, and `EXTCODECOPY`.

Given the RBR annotated with the nop information, the size relations, and the cost model C_{opcode}, we can generate GE that define the gas consumption of the corresponding code applying the classical approach to cost analysis [28] which consists of the following basic steps: (i) Each rule is transformed into a corresponding cost equation that defines its cost. Example 2 also displays the GE obtained for the rules *jump1619* and *block1731*. (ii) The nop instructions determine the gas that the rule consumes according to the gas cost model C_{opcode} explained above. (iii) Calls to other rules are replaced by calls to the corresponding cost equations. See for instance the call to *block1619* from rule *block1731* that is transformed into a call to the cost function *block1619* in Ex. 2. (iv) Size relations are attached to rules to define their applicability conditions and how the sizes of data change when the equation is applied. See for instance the size relations attached to *jump1619* that have been explained in Ex. 2.

As said before, the gas model includes a cost that comes from the memory consumption which is as follows. Let $C_{mem}(a)$ be the memory cost function for

a given memory slot a and defined as $G_{memory} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$ where $G_{memory} = 3$. Given an EVM instruction, μ_i' and μ_i denote resp. the *highest memory slot* accessed in the local memory, resp., after and before the execution of such instruction. The memory gas cost of every instruction is the difference $C_{mem}(\mu_i') - C_{mem}(\mu_i)$. Besides `MLOAD` or `MSTORE`, instructions like `SHA3` or `CALL`, among others, make use of the local memory, and hence can increase the memory gas cost.

In order to estimate this cost associated to all EVM instructions in the code of the function, we first make the following observations: (1) Computing the sum of all the memory gas cost amounts to computing the memory cost function for the highest memory slot accessed by the instructions of the function under analysis. This is because, as seen, μ_i and μ_i' refer to this position in each operation and hence we pay for all the memory up to this point. (2) This is not a standard memory consumption analysis in which one obtains the total amount of memory allocated by the function. Instead, in this case, we infer the actual value of the highest slot accessed by any operation executed in the function.

Example 3. Let us show how we obtain the memory gas cost for *block1647*. In this case, the two instructions in this block that cost memory are underlined in Fig. 3 and correspond to a `MSTORE` and `SHA3` bytecodes. In this block, both bytecodes operate on slot 0 of the memory, and they cost 3 units of gas because they only activate up to slot 1 of the memory.

2.5 PUBS solver: from equations to closed-form bounds

The last step of the gas bounds inference is the generation of a *closed-form gas upper bound*, i.e., a solution for the GE as a non-recursive expression. As the GE we have generated have the standard form of cost relations systems, they can be solved using off-the-shelf solvers, such as PUBS [5] or COFLOCO [15], without requiring any modification. These systems are able to find polynomial, logarithmic and exponential solutions for cost relations in a fully automatic way. The gas bounds computed for all public functions of EthereumPot using PUBS can be found in Table 3.1, note that they are parametric on different state variables, input and blockchain data.

3 Experimental Evaluation

This section presents the results of our evaluation of GASTAP. In Sec. 3.1, 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 Sec. 3.2, we evaluate the efficiency and effectiveness of our tool by analyzing more than 29,000 Ethereum smart contracts. To obtain these contracts, we pulled from etherscan.io [2] all Ethereum contracts whose source code was available on January 2018. GASTAP is available at `https://costa.fdi.ucm.es/gastap`.

3.1 Gas Bounds for EthereumPot Case Study

Table 3.1 shows in column **solc** the gas bound provided by the Solidity compiler **solc** [13], 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 Fig. 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 *bytes* variable as argument. This shows 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 loop indeed does not terminate if `minBetSize` in not strictly greater than zero (which could indeed happen if `ether` would take zero or a negative value). If we add the initialization instruction, and embed in the analyzer the invariant that `ether`> 0 (hence `minBetSize` becomes > 0), 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* which is a consequence of the information loss due to the soundness requirement described in extension 3 of Section 2.2. Intuitively, maximization errors may occur when the analyzer needs to compose the cost of the different fragments of the code. For the composition, it needs to maximize (*i.e.*, find the maximal value) the cost of inner components in their calling contexts (see [5] for details). If the maximization process involves memory locations that have been "forgotten" by ETHIR* (variables "?"), 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.

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	∞	80391+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+$\left\lfloor \frac{(6 \cdot nat(g1)+24)^2}{512} \right\rfloor$
getSlots	∞	1507+250·nat(g3-1/32) +75·nat(g3+31/32)	6·nat(g3)+24+$\left\lfloor \frac{(6 \cdot nat(g3)+24)^2}{512} \right\rfloor$
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)	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

Table 3.1. Gas bounds for `EthereumPot`. Function `nat` defined as `nat(l)=max(0,l)`.

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.

3.2 Statistics for Analyzed Contracts

Our experimental setup consists on 29,061 contracts taken from the blockchain as follows. We pulled all Ethereum contracts from the blockchain as of January 2018, and removed duplicates. This ended up in 10,796 files (each file often contains several contracts). 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 is due to OYENTE in 1,230 files, which represents a 11.39% of the total and to ETHIR in 829 files, which represents a 7.67% of the total. The failures of ETHIR are mainly due to the cloning mechanism in involved CFGs for which we fail to find the relation between the jump instruction and the return address.

After removing these files, our experimental evaluation has been carried out on the remaining 8,737 files, containing 29,061 contracts. In total we have analyzed 258,541 public functions (and all auxiliary functions that are used from them).

Type of result	#opc	%opc	#mem	%mem
Constant gas bound	223,294	86.37%	225,860	87.36%
Parametric gas bound	14,167	5.48%	13,312	5.15%
Time out	13,140	5.08%	13,539	5.24%
Finite gas bound (maximization error)	7,095	2.74%	5,830	2.25%
Termination unknown (ranking function error)	716	0.28%	0	0%
Complex control flow (cover point error)	129	0.05%	0	0%
Total number of functions	258,541	100%	258,541	100%

Table 3.2. Statistics of gas usage on the analyzed 29,061 smart contracts from Ethereum blockchain

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 0.4.25 or bytecode for the Ethereum Virtual Machine v1.8.18. The statistics that we have obtained in number of functions are summarized in Table 3.2, and the time taken by the analyzer in Table 3.3. The results for the opcode and memory gas consumption are presented separately.

Let us first discuss the results in Table 3.2 which aim at showing the effectiveness of GASTAP. Columns **#opc** and **#mem** contain 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, 86.37% (resp. 87.36%), 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 mentioned in Section 1. Still, there is a relevant number of functions 5.48% (resp. 5.15%) for which we obtain an opcode (resp. memory) gas bound that is not constant (and hence are potentially vulnerable). Additionally, 5.08% of the analyzed functions for opcodes and 5.24% for memory reach the timeout (set to 1 minute) due to the further complexity of solving the equations.

As the number of analyzed contracts is very large, a manual inspection of all of them is not possible. Having inspected many of them and, thanks to the information provided by the PUBS solver 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: (i) *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). (ii) *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 3 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, PUBS does not look for a ranking function. (iii) *Cover point error:* The equations are transformed into direct recursive form to be solved [5]. If the transformation is not feasible, a

Phase	$\mathbf{T}_{opcode}$ (s)	$\mathbf{T}_{mem}$ (s)	$\mathbf{T}_{total}$ (s)	**%opc**	**%mem**	**%total**
CFG generation (OYENTE*)	—	—	17,075.55	—	—	1.384%
RBR generation (ETHIR*)	—	—	81.37	—	—	0.006%
Size analysis (SACO)	—	—	105,732	—	—	8.57%
Generation of gas equations	141,576	125,760	267,336	11.48%	10.2%	21.68%
Solving gas equation (PUBS)	395,429	447,502	842,931	32.06%	36.3%	68.36%
Total time GASTAP			1,233,155.92			100%

Table 3.3. Timing breakdown for GASTAP on the analyzed 29,061 smart contracts

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.

As regards the efficiency of GASTAP, the total analysis time for all functions is 1,233,155.92 sec (342.54 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 (68.36%), 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.

4 Related Work and Conclusions

Analysis of Ethereum smart contracts for possible safety violations and security and vulnerabilities is a popular topic that has received a lot of attention recently, with numerous tools developed, leveraging techniques based on symbolic execution [23,19,25,22,20,27], SMT solving [24,21], and certified programming [9,18,7], with only a small fraction of them focusing on analyzing gas consumption.

The GASPER tool identifies gas-costly programming patterns [11], 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. In a similar vein, the recent work by Grech *et al.* [17] identifies a number of classes of gas-focused vulnerabilities, and provides MADMAX, a static analysis, also working on a decompiled EVM bytecode, data-combining techniques from flow analysis together with CFA context-sensitive analysis and modeling of memory layout. 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 both GASPER and MADMAX, they are unsuitable for tackling the challenge 1, which we have posed in the introduction.

In a concurrent work, Marescotti *et al.* identified three cases in which computing gas consumption can help in making Ethereum more efficient: (a) prevent errors causing contracts get stuck with *out-of-gas* exception, (b) place the right price on the gas unit, and (c) recognize semantically-equivalent smart contracts [24]. They 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 [10]. Their approach is based on symbolically enumerating all execution paths and unwinding loops to a limit. Instead, using resource analysis, GASTAP infers the maximal number of iterations for loops and generates accurate gas bounds which are valid for any possible execution of the function and not only for the unwound paths. Besides, 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.

Conclusions. Automated 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 employed 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 detecting vulnerabilities, but also for verification/certification of existing smart contracts.

References

1. The EthereumPot contract, 2017. https://etherscan.io/address/0x5a13caa82851342e14cd2ad0257707cddb8a31b7.
2. Etherscan. https://etherscan.io, 2018.
3. Oyente: An Analysis Tool for Smart Contracts, 2018. https://github.com/melonproject/oyente.
4. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *TACAS'14*, LNCS 8413, pages 562–567. Springer.
5. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *SAS'08*, LNCS 5079, pages 221–237. Springer.
6. E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *ATVA'18*, LNCS 11138, pages 513–520. Springer.
7. S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *CPP'18*, pages 66–77. ACM.
8. T. Bernani. Oraclize, 2016. http://www.oraclize.it.
9. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *PLAS'16*, pages 91–96. ACM.
10. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *TACAS 1999*, LNCS 1579, pages 193–207. Springer.

11. T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *SANER'17*, pages 442–446. IEEE Computer Society.
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL 1978*, pages 84–96.
13. Ethereum. Solidity, 2018. https://solidity.readthedocs.io.
14. Ethereum. Vyper, 2018. https://vyper.readthedocs.io.
15. A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *APLAS'14*, LNCS 8858, pages 275–295. Springer.
16. Ethereum Foundation. Safety - Ethereum Wiki, 2018. https://github.com/ethereum/wiki/wiki/Safety, last accessed on 14 November 14 2018.
17. N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *PACMPL*, 2(OOPSLA):116:1–116:27, 2018.
18. I. Grishchenko, M. Maffei, and C. Schneidewind. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *POST'18*, volume 10804 of *LNCS 2018*, pages 243–269. Springer.
19. S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL*, 2(POPL):48:1–48:28, 2018.
20. S. Kalra, S. Goel, M. Dhawan, and S. Sharma. ZEUS: analyzing safety of smart contracts. In *NDSS'18*. The Internet Society.
21. A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena. Exploiting The Laws of Order in Smart Contracts. *CoRR*, abs/1810.11605, 2018.
22. J. Krupp and C. Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *USENIX Security Symposium*, pages 1317–1333. USENIX Association, 2018.
23. L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *CCS'16*, pages 254–269. ACM.
24. M. Marescotti, M. Blicha, A. E. J. Hyvärinen, S. Asadi, and N. Sharygina. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *ISoLA'18*, LNCS 11247, pages 450–465. Springer.
25. I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *ACSAC'18*, pages 653–663. ACM.
26. M. Suiche. Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode, 2017.
27. P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev. Securify: Practical security analysis of smart contracts. In *CCS'18*, pages 67–82. ACM.
28. B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.
29. G. Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.