Running on Fumes

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

Elvira Albert¹, Pablo Gordillo¹, Albert Rubio¹, and Ilya Sergey²

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’
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 with respect to sound static reasoning about resource consumption, correctness, and security of replicated computations: (1) While the EVM specification provides the precise gas consumption of the low-level operations, most of the smart contracts are written in high-level languages, such as Solidity or Vyper. 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 $\infty$ otherwise. (2) As noted in [17], it is discouraged in the Ethereum safety recommendations 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, or in low-level (possibly decompiled) 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 is used for (1), an improved representation of

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.
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 \texttt{PUSH} and the \texttt{JUMP} and clone the involved part of the RBR (between the rule of the \texttt{PUSH} and of the \texttt{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, \texttt{EthIR} provides in the RBR the original bytecode instructions (from which the higher-level ones are obtained) by simply wrapping them within a \texttt{nop} functor (see Fig. 3). Although \texttt{nop} annotations will be ignored by the size analysis, they are needed later to assign a precise gas consumption to every rule.

![Selected rule including nop functions needed for gas analysis](image)

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

\textbf{Example 1.} Figure 3 shows the RBR for \texttt{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 \(s_n\) the decreasing sequence of all \(s_i\) from \(n\) down to 0. \(lv\) 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 \texttt{EthIR*}). The unknown location “?” has become a fresh variable \(\text{fresh}_0\) in \texttt{block1647}. For state variables, \(sv\) 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, \(bc\) includes the needed blockchain state variables \texttt{address}, \texttt{balance} and \texttt{timestamp}.

### 2.3 SACO: size relations for EVM smart contracts

In the next step, we generate \textit{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
Example 2. Consider the following SR (those in brackets) generated for rule $jump1649$ and $block1731$:

$$\text{jump1649}(s_{10}, s_{7}, s_{8}, s_{9}, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{21}) = \text{block1731}(s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{30}, s_{31}, s_{32}, s_{33}, s_{34}, s_{35}, s_{36}, s_{37}, s_{38}, s_{39}, s_{40}, s_{41}, s_{42}, s_{43}, s_{44}, s_{45}, s_{46}) \{ s_{10} < s_{9} \}$$

The size relations for the $jump1649$ 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 $\text{findWinner}$ 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:

\[
\begin{align*}
\text{s}_{i+1} & = 1, \text{nop}(\text{PUSH1}), \text{s}_{i+2} = \text{s}_i, \text{nop}(\text{DUP2}), \text{s}_{i+3} = 1, \text{nop}(\text{PUSH1}), \\
\text{s}_{i+2} & = \text{and}(\text{s}_{i+3}, \text{s}_{i+2}), \text{nop}(\text{AND}), \text{s}_{i+2} = \text{eq}(\text{s}_{i+2}, 0), \text{nop}(\text{ISZERO}), \\
\text{s}_{i+3} & = 256, \text{nop}(\text{PUSH2}), \text{s}_{i+2} = \text{s}_{i+3} * \text{s}_{i+2}, \text{nop}(\text{MUL}), \text{s}_{i+1} = \text{s}_{i+2} - \text{s}_{i+1}, \\
\text{nop}(\text{SUB}) & = \text{and}(\text{s}_{i+1}, \text{s}_i), \text{nop}(\text{AND}), \text{s}_{i+1} = 2, \text{nop}(\text{PUSH1}), \\
\text{s}_{i+2} & = \text{s}_i, \text{s}_i = \text{s}_{i+1}, \text{s}_{i+1} = \text{s}_{i+2}, \text{nop}(\text{SWAP1}), \text{s}_i = \text{s}_{i+1}/\text{s}_i, \text{nop}(\text{DIV})
\end{align*}
\]

\[
\text{nop}(\text{PUSH1}), \text{nop}(\text{DUP2}), \text{nop}(\text{PUSH1}), \text{nop}(\text{AND}), \text{nop}(\text{ISZERO}), \text{nop}(\text{PUSH2}), \\
\text{nop}(\text{MUL}), \text{nop}(\text{SUB}), \text{nop}(\text{AND}), \text{nop}(\text{PUSH1}), \text{nop}(\text{SWAP1}), \text{nop}(\text{DIV})
\]

\[\downarrow\]
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 \texttt{getPlayers} and \texttt{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 $C_{\text{opcode}} : s \mapsto g$ 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 $C_{\text{opcode}}$, 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 \texttt{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 \texttt{CALL} and \texttt{SELFDESTRUCT}. In these cases we use $g = \max(g_1, g_2)$ in $C_{\text{opcode}}$. (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 \texttt{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 $C_{\text{opcode}}$ as a parametric function that uses the involved location. Other bytecode instructions with parametric cost are \texttt{CALLDATACOPY}, \texttt{CODECOPY}, \texttt{RETURNDATACOPY}, \texttt{CALL}, \texttt{SHA3}, \texttt{LOG*}, and \texttt{EXTCODECOPY}.

Given the RBR annotated with the nop information, the size relations, and the cost model $C_{\text{opcode}}$, 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 \texttt{jump1619} and \texttt{block1731}. (ii) The nop instructions determine the gas that the rule consumes according to the gas cost model $C_{\text{opcode}}$ explained above. (iii) Calls to other rules are replaced by calls to the corresponding cost equations. See for instance the call to \texttt{block1619} from rule \texttt{block1731} that is transformed into a call to the cost function $\text{block1619}$ 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 \texttt{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 $C_{\text{mem}}(a)$ 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, $\mu_i'$ and $\mu_i$ 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, $\mu_i$ and $\mu_i'$ 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 $\infty$. 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 ($g_1$ and $g_3$ 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.
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 \( g_3 \) and \( g_1 \) 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).
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

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

Table 3.2. Statistics of gas usage on the analyzed 29,061 smart contracts from Ethereum blockchain
**Table 3.3.** Timing breakdown for **Gastap** on the analyzed 29,061 smart contracts

<table>
<thead>
<tr>
<th>Phase</th>
<th>T_{\text{opcode}} (s)</th>
<th>T_{\text{mem}} (s)</th>
<th>T_{\text{total}} (s)</th>
<th>%opc</th>
<th>%mem</th>
<th>%total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFG generation (Oyente*)</td>
<td>—</td>
<td>—</td>
<td>17,075.55</td>
<td>—</td>
<td>—</td>
<td>1.384%</td>
</tr>
<tr>
<td>RBR generation (EthIR*)</td>
<td>—</td>
<td>—</td>
<td>81.37</td>
<td>—</td>
<td>—</td>
<td>0.006%</td>
</tr>
<tr>
<td>Size analysis (Saco)</td>
<td>—</td>
<td>—</td>
<td>105,732</td>
<td>—</td>
<td>—</td>
<td>8.57%</td>
</tr>
<tr>
<td>Generation of gas equations (Pubs)</td>
<td>141,576</td>
<td>125,760</td>
<td>267,336</td>
<td>11.48%</td>
<td>10.2%</td>
<td>21.68%</td>
</tr>
<tr>
<td>Solving gas equation (Pubs)</td>
<td>395,429</td>
<td>447,502</td>
<td>842,931</td>
<td>32.06%</td>
<td>36.3%</td>
<td>68.36%</td>
</tr>
<tr>
<td>Total time <strong>Gastap</strong></td>
<td></td>
<td></td>
<td>1,233,155.92</td>
<td>—</td>
<td>—</td>
<td>100%</td>
</tr>
</tbody>
</table>

*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**