IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 12, 2020, accepted February 2, 2021, date of publication February 5, 2021, date of current version February 16, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3057565

Static Profiling and Optimization of Ethereum Smart Contracts Using Resource Analysis

JESÚS CORREAS [1], PABLO GORDILLO [1], AND GUILLERMO ROMÁN-DÍEZ [2]

[1]Departamento de Sistemas Informáticos y Computación, Facultad de Informática, Complutense University of Madrid, 28040 Madrid, Spain
[2]Lenguajes, Sistemas Informáticos e Ingeniería de Software, E.T.S. de Ingenieros Informáticos, Universidad Politécnica de Madrid, 28660 Madrid, Spain

Corresponding author: Pablo Gordillo (pabgordi@ucm.es)

This work was supported in part by the Spanish Ministerio de Ciencia, Innovación y Universidades (MCIU) through the Agencia Estatal de Investigación (AEI) and Fondo Europeo de Desarrollo Regional (FEDER) (EU) Project under Grant RTI2018-094403-B-C31, in part by the Comunidad de Madrid (CM) Projects co-funded by the Fondos Estructurales y de Inversión Europeos (EIE Funds) of the European Union under Grant S2018/TCS-4314 and Grant S2018/TCS-4339, and in part by the Universidad Complutense de Madrid (UCM) under Grant CT27/16-CT28/16.

ABSTRACT Profiling tools have been widely used for studying the behavior of the programs with the objective of reducing the amount of resources consumed by them. Most profilers collect the information with dynamic techniques, i.e., execute an instrumented version of the program with some specific input arguments to profile the measures of interest. This article presents a novel static profiling technique for Ethereum smart contracts that, using static resource analysis, is able to generate upper-bound expressions that can be used to produce profiling information about the measure of interest. Unlike traditional profiling tools, we get upper-bounds on the measures of interest expressed in terms of the input arguments or the state variables of the smart contracts. The information that can be obtained by the upper-bounds allows us to detect gas-expensive fragments of a Solidity program or to spot resource-related vulnerabilities at specific program points of the program. Moreover, in this article we propose an automatic optimization of Solidity programs which reduces their *gas* consumption replacing the accesses to state variables by gas-efficient accesses to local variables. We have experimentally evaluated our technique and we have detected that 6.81% of the public functions analyzed can be optimized and 1.43% are vulnerable to execute arbitrary code.

INDEX TERMS Blockchain, Ethereum, resource analysis, smart contracts, static analysis.

I. INTRODUCTION

Ethereum [37] is an open-source platform for decentralized applications and nowadays has become the world's leading programmable blockchain. One of the reasons of this success is that Ethereum smart contracts can be programmed using a Turing complete language and it includes a powerful set of tools for its development. An immutable version of the compiled smart contract can be deployed in the Ethereum platform and will be executed using the Ethereum Virtual Machine (EVM). As other blockchains, Ethereum has its native cryptocurrency named *Ether* and the execution fees for running smart contracts on the Ethereum blockchain are metered in units of *gas*. It is a measure of the amount of computational effort spent on executing each single EVM bytecode operation. The gas consumption of each EVM instruction is detailed in [37]. Miners get paid an amount in *Ether* that results of applying a gas price to the total amount of gas that took them to execute a complete transaction. Using this model, Ethereum prevents the emitters from wasting computational power, discourages the programmers to use gas-expensive operations (e.g. as the cost of replicating data in a decentralized environment is high, storage bytecodes are gas-expensive) and prevents from DoS attacks and non-terminating executions.

Solidity [15] is a programming language to write smart contracts and its compiler produces EVM code to be deployed in the Ethereum platform. The Solidity compiler includes several static analyses that produce useful information during the contract development phase. Among this information it can be found the amount of gas that a function will consume for its execution. The Solidity compiler is able to produce precise *constant* gas bounds, however, when the cost expression depends on input parameters (or information stored in the contract state), the compiler simply returns ∞ as gas bound, and we found it occurs in almost one in every ten public functions [3]. Furthermore, minimal modifications on the code make the compiler unable to detect unbounded loops and it does not warn the programmer about this potential risk.

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Hammad Memon.

In this article, we present the formal details of a resource analyzer and optimization technique that is able to infer parametric bounds and optimize the gas consumption of Solidity smart contracts. The resource analysis works at the level of EVM bytecode taking as input a smart contract (either in EVM, disassembled code, or in Solidity source code) and a public function, and returns a closed-form upper-bound, expressed in terms of the input values of the function of the contract, the state or environment data (e.g. the size of the message sent to start the transaction). Gas is probably the most valuable resource in Ethereum. Despite this fact, our techniques can be parameterized with different kinds of cost models that can be used to obtain interesting measures like the total number of instructions executed by the program or the number of times a function might be executed. Additionally, based on the ideas of program profiling [38], where programs are instrumented to collect execution information, e.g. the memory allocated or the amount of times a piece of code is executed, our analyzer can statically infer a bound on these kind of measures. To do so we use the notion of *static profiler*, which defines the *cost centers* to which the cost of the instruction will be attributed. With these static profilers, our analysis can determine, among other measures, the amount of gas consumed by a particular part of the source code or the number of times the program might access to the blockchain persistent storage. Note that, in contrast to the classical dynamic profiling techniques, our approach does not instrument the code under analysis and it does not perform any execution of the code in order to obtain the measures of the resources under consideration. We extend static resource analysis techniques to generate upper-bounds, expressed in terms of the input arguments of the program, that can be used to get profiling information (e.g. the number of times a function can be executed or the amount of gas consumed by a fragment of the program or a subset of EVM instructions) by replacing the input arguments by concrete values.

The resource analysis we describe in this article has a wide range of applications: (1) as we will show, the use of static profilers in the gas analysis can be used to detect gas-expensive fragments of the Solidity code; (2) the cost models and static profilers can be used to identify well-known vulnerabilities at EVM level; and (3) the combination of different static profilers makes our technique capable of detecting which state variables are over-accessed and is able to automatically reduce the amount of accesses by making a local copy of these state variables.

We have experimentally evaluated more than 40,000 public functions contained in 5,675 real smart contracts. With our analyses we can infer that the upper-bounds of a 9.02% of the public functions analyzed are parametric, 6.81% of the public functions can be optimizable, and at least 1.43% of them are potentially vulnerable.

A. CONTRIBUTIONS

This work is an extension and formalization of [2] including the following new contributions:

1) We extend the techniques presented in [2] by generalizing the notion of cost center to the notion of *static profiler*, including the possibility of combining the profiles within Boolean expressions.
2) We present the theoretical framework of the use of static profilers to obtain upper-bounds for different profiling information of EVM programs without running them.
3) We provide an experimental evaluation of our techniques by applying a variety of analyses with multiple combinations of profilers on 40,219 public functions contained in 1,557 Solidity files.

Our first contribution allows us to make the resource analyzer orthogonal to the profilers, which was not possible in our previous work [2]. The technique presented in [2] only allows us to generate expressions that involve one cost model and one cost center. The novelty of this work consists in the generalization of [2] to the notion of static profiler, which is able to attribute to different cost centers their corresponding cost. Furthermore, by means of these profilers, we can relate several cost models to different cost centers in a single expression by means of Boolean expressions (as it is explained in Section IV). This extension allows us to define a theoretical framework that has several applications: the inference of unbounded gas loops, the detection of gas-related vulnerabilities, or the optimization of Ethereum smart contracts. A preliminary version of the optimization phase appeared in [2]. However, the profilers that we introduce in this article provide the formal basis of our approach. Finally, we significantly extend our previous experimental evaluation by applying multiple analyses, with 8 different configurations of profilers (see Section VI for more details), to more than 5,000 real smart contracts. We also study the accuracy and performance of these analyses. In [2] only 7 Solidity files were analyzed to describe the use of the optimization tool.

B. ORGANIZATION

The rest of the article is organized as follows: Section II reviews the current state-of-the art of analysis and profiling tools of Ethereum smart contracts; Section III briefly describes the EVM language and the resource analysis of EVM programs; Section IV et seq. present the main contributions of this article: Section IV defines the notion of profiler and its use in the detection of resource-related vulnerabilities; Section V describes an automatic optimization of programs whose number of accesses to state variables can be reduced; Section VI experimentally evaluates the techniques described in previous sections; and Section VII contains the conclusions and future work.

II. RELATED WORK

The number of smart contracts deployed in Ethereum blockchain is growing exponentially in the last few years. As a consequence, several tools [12]–[14], [19]–[21], [24]–[27], [29], [30], [34] have been developed to analyze and verify smart contracts both at Solidity and EVM level. Some of these approaches [20], [21], [24]–[26], [30], [34] are dynamic

and try to identify potential vulnerabilities such as unhandled exceptions, transaction state dependencies, integer over- and under-flows or reentrancy vulnerabilities inspecting concrete execution traces using SMT solvers. However, a small part of the current state-of-the art approaches are focused on resource analysis [19], [27].

In the recent years, some tools [12]–[14], [29] focused on optimization of smart contracts have been published. In [13], the authors present the tool GASPER. It identifies 7 gas costly patterns and advises the users that the analyzed Solidity code can be optimized. This work has been extended in two different tools: the tool GasReducer [14] increases the number of identified patterns to 24, and replaces the patterns by predefined EVM sequences that consume less gas than the original ones; the tool GasChecker [12] improves the performance of GASPER proposing a new approach based on a parallelized symbolic execution and cloud computing. Nevertheless, these tools are not fully automatic: the patterns need to be identified manually and they are not open-source or they are not publicly available, so we cannot compare our approach to theirs. ebso [29] tool works at EVM level and tries to superoptimize EVM code applying some simplification rules and generating an SMT encoding based on bit-vector theory. These optimizations are intra-block while our approach analyzes several blocks and is focused on optimizing storage-expensive fragments of Solidity code.

Program profiling is a dynamic program analysis technique for locating places of the code where optimizations are worth to be performed. It is based on collecting information during the execution of the program (number of times a function or code fragment is executed, execution paths, number of instructions executed, number of accesses to specific memory positions, etc.). They are usually implemented either instrumenting the code of the program being analyzed, or using some simulation tool. However, the dynamic nature of profiling makes difficult to apply this technique in many contexts in which either it is difficult to obtain a representative set of input data for executing the program, or its execution for some test cases is excessively expensive or even impossible. Due to these limitations, there are a number of works focused on profiling techniques combined with information obtained by means of static analyses [8], [9], [38]. Static profiling analyzers have been traditionally used for obtaining low-level information such as jump prediction probabilities, and are combined with heuristics on the behavior of the program [39]. Nevertheless, purely static profiling techniques have also been applied to several contexts to obtain bounds of the accumulated cost of the execution of programs [22]. Regarding Ethereum and smart contracts, there are some tools for dynamically profiling contracts such as sol-profiler,[1] sol-function-profiler,[2] solidity-coverage,[3] and eth-gas-reporter,[4] although to the best of our knowledge there is no profiling tool that uses static analysis techniques for obtaining profiling information for smart contracts. There are some approaches [6], [11], [36] focused on other aspects rather than the code such as the behavior and the interaction between smart contracts or the CPU time that each instruction takes to be executed. The tool Visualgas [32] uses a dynamic approach to the analysis of gas cost of smart contracts. In this case, it is based on fuzz testing techniques [5], [23] combined with symbolic execution to generate test cases with high coverage. The approach presented in [33] uses `pprof` and geth to obtain execution measures such as the use of CPU or memory of Solidity code previously instrumented. The tool GasMet [10] uses a set of predefined metrics to evaluate the quality of the code depending on the gas it consumes. In contrast, our approach diverges from the classical dynamic profiling approaches, as it is based on static analysis with cost centers and lets us infer measures that are used to spot profiling information. It allows us to get information about different resources (not only related to gas consumption) at multiple levels of granularity, without instrumenting and executing the EVM bytecode under analysis.

[1]https://github.com/Aniket-Engg/sol-profiler
[2]https://github.com/EricR/sol-function-profiler
[3]https://github.com/sc-forks/solidity-coverage
[4]https://github.com/cgewecke/eth-gas-reporter

III. BACKGROUND: EVM AND RESOURCE ANALYSIS

In this section we briefly introduce some relevant concepts used throughout this article that are the starting point for the contributions presented. We first review some characteristics of EVM programs and compare it with other stack based languages. Then we introduce the use of a resource analysis applied to EVM programs to obtain an upper-bound on the use of a resource of interest, and we finally describe how this analysis can be used to obtain an upper-bound of the gas consumption of a transaction to prevent out-of-gas errors.

A. EVM LANGUAGE

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 detailed description of the language and the complete set of operation codes can be found in [37]. In this section we will overview some relevant aspects of EVM. In comparison to other stack-based languages like Java, EVM presents some relevant differences: (1) EVM language is an untyped language; (2) there is no notion of data functions, the calls between different methods are modeled with (conditional or unconditional) jumps to corresponding addresses of the bytecode; (3) data structures are not visible in the bytecode; (4) the elements stored in the stack, memory[5] or storage are 256-bits words; (5) the jump addresses are not known at compilation time and they are read from the execution stack; (6) the size of the stack is not fixed, a program point of the EVM bytecode may be reached with different sizes of the stack as it will depend

[5]Internally, **EVM** has opcodes to access both single bytes and 32-byte words of memory.

on the execution context. In the Ethereum environment, once the EVM of a smart contract is deployed, it can be invoked by parties of the blockchain (users or other contracts) by means of *transactions*. The execution of a transaction with a contract as recipient implies the execution of the EVM code of a public function of the contract. Each EVM instruction consumes an amount of *gas* specified in [37]. *Gas* is a notion of major importance and the motivation behind the use of gas is twofold: on one hand, it rewards miners operating the blockchain, and on the other hand it is used to avoid denial-of-service attacks on the platform. If the amount of gas sent to a transaction to be executed is less than the gas that it actually consumes, the transaction is aborted. Interestingly, the amount of gas is platform-independent. However, it is not a standard resource model as there are some EVM bytecodes that consume a fixed amount of gas, other bytecodes that consume a fixed amount of gas but it depends on some condition, and others that consume a non-constant amount of gas. In the last two cases, the amount of gas consumed depends on the current state of the execution environment.

Example 1: The following function is an excerpt of a verified smart contract `VestingContract`[6] taken from Etherscan [7] service:

```
1  contract VestingContract {
2
3   uint public totTok; // originally totalTokens
4   uint[] vestingSchedule;
5
6   function updateVestingSchedule(uint256[]
       memory _vestingSchedule) public onlyOwner {
7
8    require(vestingStartTime == 0);
9
10   vestingSchedule = _vestingSchedule;
11
12   for(uint i=0; i<vestingSchedule.length; i++){
13    totTok = totTok.add(vestingSchedule[i]);
14   }
15  }
16   ...
17 }
```

This code shows function `updateVestingSchedule`, which receives an array `_vestingSchedule` as a parameter and copies it to a state variable (Line 10, L10 for short). Additionally, by means of the loop at L12, it accumulates all values received in the array in a state variable `totTok`. The addition is performed by means of library `SafeMath`, widely used in Ethereum programs, which implements the basic mathematical operations with an overflow check.
Interestingly, though this code apparently executes only one loop, the EVM code obtained from this function includes three different iterative parts: (a) the assignment in L10 produces a loop that copies all values contained in `_vestingSchedule` to the state variable; (b) L10 produces another loop that executes when the array stored at `vestingSchedule` is longer than the array stored at `_vestingSchedule`, setting to zero the remaining elements of `vestingSchedule`; and, (c) the loop at L12, which traverses all elements in `vestingSchedule`. ■

[6]https://etherscan.io/address/0×1273c54cc3a7d5320b210437906eda3ea1aa1a36

[7]https://etherscan.io/

B. RESOURCE ANALYSIS

Static resource analyzers aim at inferring a bound on the resource consumption (a.k.a. cost) of executing a program on a given input data. Resource analysis is usually performed in two phases [35]:

- a first phase whose goal is the generation of the cost relations from the program to be analyzed (see [4], [18]);
- and a second phase that aims at computing a closed-form upper-bound from the cost relations [1], [16].

In this article, we aim at producing a cost relation system from EVM programs and, using off-the-shelf solvers (e.g. [1], [7], [16]), obtain upper-bounds from the cost relation system. In the context of Ethereum smart contracts, working with the EVM code presents the following advantages: (i) only 1% of the source code of the deployed smart contracts is available [19] (the blockchain only stores the EVM bytecode of the contract); (ii) there is valuable information that is available at EVM level (e.g., the information needed to compute the gas consumption, or related to verification purposes); (iii) the code may be subject to optimizations made by the compiler, which might be overlooked by a source-code analyzer.

The first phase of our resource analysis of EVM programs is described in detail in [3]. The transformation of an EVM program into a cost relation system is done in three steps: (1) a first step that produces a *stack-sensitive control flow graph* (S-CFG) [3], which replicates those nodes that can be reached with different stacks to capture all possible execution traces of the program; (2) a second step that takes the S-CFG of the smart contract and produces a *rule-based representation* (RBR) of the program [3], which preserves the flow of the program and where recursion is the only form of iteration; and, (3) a third step that produces a *cost relation system* from the RBR [3], which models the cost of the program in a set of cost equations.

Cost equations are syntactically close to recurrence relations. They can be converted into closed-form expressions, written in terms of the input parameter, which bound the cost of the program with respect to a specific cost measure. The structure of invocations in the cost relation system can be directly obtained from the rule base representation. In general, given an EVM program, we can compute a cost relation system composed of equations of the form:

$$C(\overline{x}) = \texttt{cexp} + \sum_{i=1}^{k} Y_i(\overline{y}_i) + C(\overline{z}), \varphi \qquad (1)$$

where C is the equation name and $\overline{x}$ its parameters; `cexp` is the cost produced by the execution of the code related to the equation; $\sum_{i=1}^{k} Y_i(\overline{y}_i)$ is the addition of the cost produced by calls to other equations; $C(\overline{z})$ includes the cost associated with recursive invocations; and φ is a set of linear constraints relating the values of the variables $\overline{x}$, $\overline{y_i}$ and $\overline{z}$.

One fundamental component that is used to produce the cost relation system is the *cost model*. A cost model is a function θ that, depending on the type of resource to be measured, assigns a measure of the cost of executing each instruction in the program. Cost models are used in the computation of the cost associated with each equation `cexp` as follows: given a rule with instructions $b_1, \ldots, b_n$, its cost associated is $\texttt{cexp} = \sum_{i=1,\ldots,n} \theta(b_i)$. In our approach we only consider platform-independent cost models, e.g. the number of instructions or the gas consumed by the program. On the contrary, those resources that depend on the execution environment, like the execution time or energy consumed by a program, are not considered in our approach.

A simple cost model, which assigns cost 1 for all instructions, can be defined to compute an upper-bound on the *number of EVM bytecode instructions* that might be executed by an EVM program:

Definition 1 (number of instructions cost model): Given an EVM bytecode instruction b, the cost model $\theta_{\mathcal{I}}(b)$ is a function defined as follows:

$$\theta_{\mathcal{I}}(b) = 1$$

The set of linear constraints φ in equation (1) and the cost equations are used to compute the upper-bounds. The computation starts with the inference of the *ranking functions* [17], [31] of the recursive equations. The ranking functions allow us to guarantee the termination of the program and bound the number of recursive invocations. Then, the worst case cost of executing an equation is computed by multiplying its maximum number of executions and the worst case cost of executing it, which will be computed by using φ. With this technique, we can compute constant, logarithmic, polynomial or exponential cost expressions.

We use $\mathcal{U}_{\mathcal{I}}$ to refer to the upper-bound computed using the cost model $\theta_{\mathcal{I}}$. In what follows, we will use $\mathcal{U}_{\mathcal{X}}$ to refer to the upper-bound computed by using the cost model $\theta_{\mathcal{X}}$.

Example 2: Using $\theta_{\mathcal{I}}$, we get that an upper-bound on the amount of EVM instructions executed by the function shown in Example 1 is the expression:

$$\begin{aligned}\mathcal{U}_{\mathcal{I}} = 333+ & \\ 21 * \mathsf{nat}(\_vs) + & \quad (a)\\ 17 * \mathsf{nat}(vs - \_vs)+ & \quad (b)\\ 77 * \mathsf{nat}(\_vs)+ & \quad (c)\end{aligned}$$

In what follows, the cost expressions uses $\_vs$ to refer to the length of `_vestingSchedule` and vs refers to the length of the state variable `vestingSchedule`. Function $\mathsf{nat}(x)$ is a function used in cost expressions to avoid negative values: it returns x if $x > 0$ and 0 otherwise. Observe that it is needed, not only to handle negative values in the input parameters, but also to avoid negative values operations like $vs - \_vs$. In the cost expression above we can see the three parametric expressions, within the nat functions, which corresponds to the three loops mentioned in Example 1. Subexpression (a) includes the number of instructions needed for copying the parameter to the state variable, (b) includes the instructions of cleaning the remaining $vs - \_vs$ elements; and (c) corresponds to the instructions of the loop at L12. Observe that, though the loop at L12 is bounded by `vestingSchedule`, its number of iterations in the upper bound is expressed in terms of the length of the input argument $\_vs$. ■

C. GAS ANALYSIS

The most common gas consumption vulnerabilities in Ethereum are caused by iterative operations whose number of iterations are bounded by unknown values. The static analyses included in the Solidity compiler detect constant gas bounds and some patterns that might potentially produce out-of-gas errors, for instance, loops that are bounded by a state or a parameter value, methods that receive an array as a parameter or **`delete`** operations over a state array. These vulnerabilities might be detected at compilation time. Thus, a client invoking a contract whose source code is not available might be potentially vulnerable but it will not be warned by the platform.

The second relevant point to note happens when the gas is bounded by a parametric expression. In these programs, the Solidity compiler does not infer a precise bound, returning ∞ as gas bound for them. This information only warns the programmer in the use of loops, however, if the use of loops cannot be avoided, it is not useful for determining the range of values that are acceptable to run without out-of-gas errors. The computation of the amount of gas consumed by each instruction is defined in [37] and it is a complex model: the gas consumed by some instructions depends on the execution context, i.e., the amount of gas consumed might depend on some values stored in the stack (e.g. `EXP`, `CALLDATACOPY`), on the maximum amount of memory allocated of the value (e.g. `MLOAD`, `MSTORE`), or on the previous values stored in a storage location (e.g. `SSTORE`). As a consequence, the computation of the gas consumed by each instruction requires a complex cost model, $\theta_{\mathcal{G}}$, which allow us to infer an upper-bound on the amount of gas consumed by an EVM program. The details of $\theta_{\mathcal{G}}$ can be found at [3] and we will use it as a black box in the rest of the paper.

Example 3: The computation of the upper-bound using $\theta_{\mathcal{G}}$ returns two expressions $\mathcal{U}_{\mathcal{G}}$ and $\mathcal{U}'_{\mathcal{G}}$: the first one that corresponds to the gas consumed by the EVM instructions executed, and the second one, whose cost depends on the amount of memory used by the program (see [3], [37] for details). When applied to the code of Example 1 the results are:

$$\begin{aligned}\mathcal{U}_{\mathcal{G}} = 22608 + & \\ 3 * nat(\_vs) + & \quad (d)\\ 20070 * nat(\_vs) \ + & \quad (a)\\ 5057 * nat(vs - \_vs) + & \quad (b)\\ 21066 * nat(\_vs) & \quad (c)\end{aligned}$$

$$\mathcal{U}'_{\mathcal{G}} = 3 \cdot (nat(\_vs) + 9) + \left\lfloor \frac{(nat(\_vs) + 9)^2}{512} \right\rfloor$$

As before, we see that the upper-bound $\mathcal{U}_{\mathcal{G}}$ depends on the same loops detected in Example 2 plus a new cost expression

(d) which depends on nat(_vs). This subexpression appears when using $\theta_{\mathcal{G}}$ because the amount of gas needed to copy the input array to memory depends on the length of the array (the details of the cost of `CALLDATACOPY` can be found at [37]). In $\mathcal{U}'_{\mathcal{G}}$, the amount of gas depends on the highest memory address accessed by the program. As the array `_vestingSchedule` is passed as a parameter to the function `updateVestingSchedule` at L6, it is copied to memory before starting the execution of the function. Hence, the memory gas bound $\mathcal{U}'_{\mathcal{G}}$ is also parametric on the length of the input array. By means of these expressions we can determine how much gas is needed to successfully process the transaction that executes function `updateVestingSchedule` and, thus, the upper-bounds might help to avoid potential out-of-gas errors. ■

IV. STATIC PROFILING OF EVM PROGRAMS USING RESOURCE ANALYSIS

In this section we describe how our resource analysis can be used to prevent *resource-related vulnerabilities* and how we can extend it to attribute the cost to different cost centers by using the notion of static profilers.

The application of the resource analysis using the cost models previously described in Section III produces a cost expression which over-approximates the cost accumulated by the execution of the whole program. An interesting and flexible extension of our resource analyzer can be done by incorporating to it the notion of *cost centers* [28]. Cost centers are symbolic artifacts of the form $c(x)$ that can be used in the computation of the cost expressions as we are able to attribute the cost to the cost center responsible of producing it. The combination of cost centers and cost models will allow us to perform a *static profiling* of the program by distributing the resource consumption between the different parts of the program. Our approach splits the computation of `cexp` from equation (1) in two parts: the application of the cost model to get the cost associated to the instruction as it was described in Section III-B and the generation of the cost centers responsible of executing it. We use $\mathcal{P}$ to denote a function responsible of generating the cost center expression of a given program point of the program. The concrete definition of $\mathcal{P}$ will depend on the granularity and the kind of information we aim at profiling, as e.g., the program counter of the bytecode instruction responsible of the cost.

In this way, by using a profiler we enrich the cost expressions of the cost relation system: given a profiler $\mathcal{P}(pp)$, a cost model θ and a rule with a list of program points $pp_1, \ldots, pp_n$ of the form $pp \equiv pc{:}b$, where pc corresponds to the program counter and b to the EVM bytecode instruction, we compute `cexp` as follows:

$$\texttt{cexp} = \sum_{i=1}^{n} \mathcal{P}(pp_i) * \theta(b_i)$$

The use of a profiler produces a closed-form upper-bound that includes expressions of the form $c(x)$, where the values of x will depend on the profiler used. For instance, if the profiler includes the program counter, in expressions of the form $c(x)$, the value of x could be all program counters of the program. We use $\mathcal{U}^{\mathcal{X}}_{\mathcal{Y}}$ to refer to the upper-bound computed by using $\mathcal{P}_{\mathcal{X}}$ and $\theta_{\mathcal{Y}}$. In order to obtain the cost associated to a given cost center λ, i.e., $\mathcal{U}^{\mathcal{X}}_{\mathcal{Y}}|_{\lambda}$, we replace $c(\lambda)$ by 1 and $c(\_)$ by 0 for all other cost centers in the upper-bound. We can easily extend the computation of an upper-bound for a given set of cost centers S, i.e., $\mathcal{U}^{\mathcal{P}}_{\theta}|_{S}$, by replacing $c(\lambda)$ by 1 if $\lambda \in S$ and by 0 otherwise.

Depending on the information we are interested in, we can define different profilers. For instance, the ***EVM** instructions profiler*, $\mathcal{P}_{\mathcal{B}}$, allows us to separate the resource consumption in terms of the different EVM bytecode instructions executed by the program:

*Definition 2 (**EVM** instructions profiler):* Given an EVM program point pp, we define $\mathcal{P}_{\mathcal{B}}$ as follows:

$$\mathcal{P}_{\mathcal{B}}(pp) = c(b)$$

In this profiler we are accounting the cost according to the specific EVM bytecode instruction that is responsible of the corresponding resource consumption. The computation of $\mathcal{U}^{\mathcal{B}}_{\mathcal{I}}$ allows us to infer the amount of times a program point might be executed (e.g. external calls, arithmetic operations, etc.) or whether the program could be executing a high number of memory or storage accesses as we can see in the following example.

Example 4: Given the Solidity code shown at Example 1, by computing

$$\mathcal{U}^{\mathcal{B}}_{\mathcal{I}}|_{\{\texttt{SLOAD},\texttt{SSTORE}\}} = 8 + 6 * nat(\_vs) + nat(vs - \_vs)$$

we get an upper-bound on the number of storage (state variables) accesses performed by the function under analysis. Analogously, by using $\theta_{\mathcal{G}}$, we obtain a bound on the amount of gas consumed due to storage accesses:

$$\mathcal{U}^{\mathcal{B}}_{\mathcal{G}}|_{\{\texttt{SLOAD},\texttt{SSTORE}\}} = 21400 + 40800 * nat(\_vs) + 5000 * nat(vs - \_vs)$$

■

Thanks to their flexibility, the use of profilers allows us to bind the resource consumption with the program point where this consumption is produced by using these program points in the profiler definition.

Definition 3 (program point profiler): Given a program point pp, we define $\mathcal{P}_{\mathcal{S}}$ as follows:

$$\mathcal{P}_{\mathcal{S}}(pp) = c(pc)$$

Interestingly, the use of the program point in the upper-bound computation allows us to integrate the information generated from the compilation of a Solidity program with the results obtained by our resource analyzer at EVM level. The information produced by the Solidity compiler includes a mapping that binds each EVM program point with the corresponding piece of Solidity code that produces it. For instance, with this information, given a line number l in the Solidity code, we will use function *get_line_pps*(l) to refer to the set

of EVM program points generated for line l of the Solidity program. Analogously, we will use $get\_func\_pps(f)$, where f is a Solidity function (public or private), for referring to the EVM program points produced by the code in f. Combining the profilers defined above we can get relevant information about our Solidity program: (1) given a Solidity line of code we can use $\mathcal{U}^{\mathcal{B}}_{\mathcal{G}}|_{get\_line\_pps(l)}$ to compute the amount of gas consumed by a specific line of code (or a set of lines) discarding the gas consumed by other lines of code; or (2) given a Solidity function, we can use $\mathcal{U}^{\mathcal{B}}_{\mathcal{G}}|_{get\_func\_pps(f)}$ to determine the amount of gas consumed within a Solidity function ignoring the cost due to the rest of the functions.

Example 5: The computation of $\mathcal{U}^{\mathcal{B}}_{\mathcal{G}}|_{get\_line\_pps(10)}$ returns a bound on the amount of gas consumed at L10:

$$\mathcal{U}^{\mathcal{B}}_{\mathcal{G}}|_{get\_line\_pps(10)} = 2236 + 20070 * \mathsf{nat}(\_vs) + 5057 * \mathsf{nat}(vs - \_vs)$$

Observe that this upper-bound clearly shows the two hidden loops produced by the array assignment of L10 and its corresponding number of iterations: the first loop, bounded by nat(_vs), which copies the array from memory to the state variable; and the second one, which cleans the potential remaining part of the state variable array when its original size is larger than the input array. ■

A relevant issue to note of the static analyses built by Ethereum on top of the Solidity compiler is that they cannot warn some common gas-efficient practices. It spots a potential vulnerability when a loop is bounded by a state variable (basic variable or array) or by a parameter variable passed to a function. However, very simple modifications like the use of a local variable to save the value stored in the state variable before its use to bound the loop, are not detected by the compiler analyses. Interestingly, we can use the cost model $\theta_{\mathcal{I}}$, not only to bound the number of iterations of the loops, but also to find out which are the input parameters involved in the number of iterations. If the upper-bound is parametric with respect to an input variable, a parameter or a state variable, we are able to spot a potential vulnerability at the corresponding line of the Solidity program.

Example 6: Given the code shown in Example 1, the static analyses included in the compiler determine that there is a loop over a dynamic array with a *non-fixed number of iterations*. However, a very simple modification, like saving the value of `vestingSchedule.length` in a local variable, *hides* the potential vulnerability to the compiler. For instance, if we replace the code at L12 of the running example by the following gas efficient code:

```
    uint len = vestingSchedule.length;
(⋆) for (uint i=0; i<len; i++)
```

The analyses of the Solidity compiler will not spot any potential vulnerability. On the contrary, by means of our static profiling we can produce the following upper bound in the number of instructions executed at the line marked with (⋆):

$$\mathcal{U}^{\mathcal{I}}_{\mathcal{G}}|_{get\_line\_pps(\star)} = 10 + 16 * nat(\_vs)$$

Note that, as before, in spite of the modification, our resource analysis is able to detect that the number of iterations of the loop depends on the length of the longest array. ■

Profilers flexibility allows us to extend the capabilities of the resource analysis by combining multiple profilers in a unique cost expression. This combination can be done by using a logical formula $\mathcal{L}(pp)$ to represent the relations between the different profilers $\mathcal{P}_1(pp), \ldots, \mathcal{P}_n(pp)$ included in the formula, which adheres the following grammar:

$$\mathcal{L} = \mathcal{P} \mid E \vee E \mid E \wedge E$$

Note that we do not include the parameter pp as it is clear from the context. Using a logical formula to model the relation between the different profilers we can express complex properties combining the information included in the cost centers returned by the profilers. Given a formula $\mathcal{L}$ that relates a list of profilers, we use $\mathcal{C}(\mathcal{L})$ to refer to the following transformation:

$$\mathcal{C}(\mathcal{L}) = \begin{cases} \mathcal{P} & \text{if } \mathcal{L} = \mathcal{P} \\ \mathcal{C}(E) * \mathcal{C}(E) & \text{if } \mathcal{L} = E \wedge E \\ max(\mathcal{C}(E), \mathcal{C}(E)) & \text{if } \mathcal{L} = E \vee E \end{cases}$$

In this way, given a rule of the RBR with program points $pp_1, \ldots, pp_n$, a list of profilers $\mathcal{P}_1, \ldots, \mathcal{P}_m$ and a logical formula $\mathcal{L}(pp)$ defined on $\mathcal{P}_1, \ldots, \mathcal{P}_m$, the cost equations are computed as follows:

$$\texttt{cexp} = \sum_{i=1}^{n} \mathcal{C}(\mathcal{L}(pp_i)) * \theta(b_i)$$

Analogously, when an upper-bound is computed using a logical formula $\mathcal{L}$ defined on $\mathcal{P}_1, \ldots, \mathcal{P}_m$, it can receive multiple sets of cost centers $S_{\mathcal{P}_1}, \ldots, S_{\mathcal{P}_m}$, one set per profiler:

$$\mathcal{U}^{\mathcal{L}}_{\theta}|_{S_{\mathcal{P}_1}, \ldots, S_{\mathcal{P}_m}}$$

Example 7: By using this generalization, the conjunction of $\mathcal{P}_{\mathcal{B}}$ and $\mathcal{P}_{\mathcal{S}}$, i.e., $\mathcal{L} = \mathcal{P}_{\mathcal{B}} \wedge \mathcal{P}_{\mathcal{S}}$, allows us to get the amount of storage accesses performed in a specific line of code L10:

$$\mathcal{U}^{\mathcal{L}}_{\mathcal{I}}|_{\{\texttt{SLOAD},\texttt{SSTORE}\}, get\_line\_pps(10)} = 2 + \mathsf{nat}(\_vs) + \mathsf{nat}(vs - \_vs)$$

■

V. OPTIMIZING GAS CONSUMPTION

One of the most interesting applications of resource analysis is the detection of potential optimizations during the development process. During this phase, our analysis might help to spot possible bottlenecks in the code: e.g. functions that might be consuming a high amount of resources and can help the programmer to reduce this consumption.

As mentioned above, the most relevant resource in the Ethereum context is the amount of gas consumed by the program. In this section we present how our tool helps in

the development phase to detect improvable functions and we propose an automatic optimization of those programs accessing to state variables when those accesses can be replaced by gas-efficient stack accesses. In particular, we aim at replacing multiple accesses to the (global) storage data within a fragment of code (each write access costs 20,000 units of gas in the worst case and 5,000 in the best case) by accesses to local variables. To do so, when it is safe, we first copy the data in the storage into a newly defined local variable with the same name of the storage variable. By means of this transformation, the accesses to the storage variable are now performed with the local variable instead of the original storage variable, reducing the amount of gas needed to execute the operations. Finally, when it is needed, we update the storage variable with the data of the local variable. In essence, in this transformation the local variable acts as a cache for the storage variable. Unluckily, as we are copying global data into local variables, this transformation is not safe when other functions can read or modify the storage variable. For guaranteeing the soundness of the transformation we first need to check two conditions:

(a) the storage variable is only accessed in the function to be optimized;
(b) there are no external calls at any function reachable from the function of interest.

The first condition (a) guarantees that there are no transitive calls from the function to be optimized that modify neither read the state variable of interest. To do so, we define a profiler to include in the upper-bound which is the state variable being accessed when executing a storage instruction. Given a program point pp whose instruction b is `SLOAD` or `SSTORE`, we assume the existence of a function $getStVar(pp)$ which returns the state variable s accessed at pp or $\bot$ if the variable cannot be statically computed (e.g. if it is used to traverse an array). We use $getStVar(pp)$ in the definition of the *storage profiler*:

Definition 4 (storage profiler): Given a program point pp, we define $\mathcal{P}_{\mathcal{T}}$ as follows:

$$\mathcal{P}_{\mathcal{T}}(pp) = \begin{cases} c(getStVar(pp)) & \text{if } b \in \{\texttt{SLOAD}, \texttt{SSTORE}\} \\ 0 & \text{otherwise} \end{cases}$$

The following combination of profilers allows us to compute the number of storage accesses performed at the different program points of the program:

$$\mathcal{L}_{st} = \mathcal{P}_{\mathcal{B}} \wedge \mathcal{P}_{\mathcal{S}} \wedge \mathcal{P}_{\mathcal{T}}$$

With $\mathcal{L}_{st}$, given a set of state variables S and a function f, we can compute the amount of storage accesses performed to state variables in S in the code of f. To do so, we compute $\mathcal{U}_{\mathcal{I}}^{\mathcal{L}_{st}}$ and evaluate it as follows:

$$\mathcal{U}_{\mathcal{I}}^{\mathcal{L}_{st}}|_{\{\texttt{SLOAD},\texttt{SSTORE}\},get\_func\_pps(f),S}$$

Analogously, we can also get the amount of accesses to storage locations performed in other functions just by using the set with the *rest* of the program points of the code: $notin\_func\_pps(f)$.

Example 8: The upper-bound obtained by analyzing the running example using $\mathcal{L}_{st}$ allows us to compute the amount of storage accesses attributed to `totTok` in the function **updateVestingSchedule** (upVs for short):

$$\mathcal{U}_{\mathcal{I}}^{\mathcal{L}_{st}}|_{\{\texttt{SLOAD},\texttt{SSTORE}\},get\_func\_pps(upVs),\{\texttt{totTok}\}} = 2 * \mathsf{nat}(\_vs)$$

■

The second condition (b) checks the existence of external calls to avoid potential reentrant calls which might modify the values hold in the state variables. This can be done by means of a flow analysis over the **EVM** code or we can use the upper bound $\mathcal{U}_{\mathcal{I}}^{\mathcal{B}}|_{\{\texttt{CALL},\texttt{STATICCALL},\texttt{DELEGATECALL}\}} > 0$ to evaluate this property.

Definition 5 (optimizable state variable): Given a function f, its **EVM** code and a basic type storage variable s, we say that s is optimizable in f, *optimizable(f,s)*, when the following conditions hold:

leftmargin=*

1) $\mathcal{U}_{\mathcal{I}}^{\mathcal{L}_{st}}|_{\{\texttt{SLOAD},\texttt{SSTORE}\},get\_func\_pps(f),\{s\}} > 2$
2) $\mathcal{U}_{\mathcal{I}}^{\mathcal{L}_{st}}|_{\{\texttt{SLOAD},\texttt{SSTORE}\},notin\_func\_pps(f),\{s,\bot\}} = 0$
3) $\mathcal{U}_{\mathcal{I}}^{\mathcal{B}}|_{\{\texttt{CALL},\texttt{STATICCALL},\texttt{DELEGATECALL}\}} = 0$

Observe that constraint (1) restricts the optimization to those state variables whose number of accesses can be reduced, while restrictions (2) and (3) are safety conditions to ensure that there are no accesses to the state variables of interest in other functions.

Example 9: In the code described in Example 1, we find one optimizable state variable: `totTok` (called `totalTokens` in the original program). Its number of accesses is $2 * \mathsf{nat}(\_vs)$ as we have computed in Example 7; it is not accessed in other methods reachable from `updateVestingSchedule`; and the function is not performing calls to external addresses because invocations to functions defined in libraries (like `SafeMath`) replace the invocation by the code of the invoked library function. ■

Our transformation is performed in the **Solidity** program and it consists in saving in local variables the values of the state variables at the beginning of the function of interest. At the end of the function we restore the state variables with the values stored in the local variables which might be modified along the function execution. Given a **Solidity** function f, we assume that exists a function $getStVars(f)$, which returns the set of basic type storage locations $s_1, \ldots, s_n$ accessed in the code of f. Those accesses that cannot be statically obtained are not returned by $getStVars(f)$. We use function $type(s)$ to get the **Solidity** type of the storage variable s. We define the transformation as follows:

Definition 6 (function optimization): Given a function f of the form $f(\overline{x})\ \overline{m}\ returns\ (\overline{r})\{code\}$ where $\overline{x}, \overline{m}, \overline{r}$ are the parameters, modifiers and returned values of f, respectively, and $S=\{s \mid s \in getStVars(f) \wedge optimizable(f, s)\}$, the

optimized function is of the form:

$$
\begin{array}{lll}
 & f(\overline{x})\ \overline{m}\ returns\ (\overline{r})\{ & \\
(1) & \quad type(s)\ s = get\_s(); & \forall\, s \in S \\
(2) & \quad code & \\
(3) & \quad set\_s(s); & \\
 & \} & \forall\, s \in S \\
 & & \\
(4) & function\ get\_s()\ private\ returns\ (type(s))\{ & \forall\, s \in S \\
 & \quad return\ s; & \\
 & \} & \\
(5) & function\ set\_s(type(s)\ val)\ private\ \{ & \forall\, s \in S \\
 & \quad s = val; & \\
 & \} &
\end{array}
$$

We add some pieces of code for all optimizable state variables in the transformation. The optimization process is applied in five steps: (1) the contents of the state variables are copied to local variables with the same names; (2) the original code is kept unchanged but now it accesses local variables due to the shadowing of the state variables; (3) modified values in local variables are copied back to the storage; (4) a private function is defined to get the state variable value; and (5) another function is defined to copy the value back to the storage. Observe that points (3) and (5) are not needed when the state variable is not changed in the code of the function.

Example 10: Given the running example, the optimization of `updateVestingSchedule` produces the following transformed code:

```
function updateVestingSchedule(uint256[] memory
    _vestingSchedule) public onlyOwner {
  uint totTok = get_totTok();

  require(vestingStartTime == 0);

  vestingSchedule = _vestingSchedule;

  for(uint256~i=0; i<vestingSchedule.length; i
      ++) {
    totTok = totTok.add(vestingSchedule[i]);
  }
  set_totTok(totTok);
}

function get_totTok () private returns (uint) {
  return totTok;
}

function set_totTok (uint val) private {
  totTok = val;
}
```

Note that we have only optimized `totTok` because variable `vestingSchedule` is not a basic variable and cannot be optimized. The computation of $\mathcal{U}_{\mathcal{G}}$ now returns an expression which clearly reduces the total amount of gas:

$$
\begin{array}{ll}
\mathcal{U}_{\mathcal{G}} = 42044 + & (e') \\
\quad 3 * nat(\_vs) + & (d') \\
\quad 20070 * nat(\_vs) + & (a') \\
\quad 5057 * nat(vs - \_vs) + & (b') \\
\quad 860 * nat(\_vs) & (c')
\end{array}
$$

TABLE 1. Statistics of the 40,219 public functions analyzed.

Resource Analysis		
Result	# of func.	% of func.
Constant bound	32,513	80.84%
Parametric bound	3,627	9.02%
Timeout	3,056	7.60%
Errors	1,023	2.54%
Number of public functions	40,219	100%
Optimization		
Result	# of func.	% of func.
Optimizable public functions	2,739	6.81%
Non-optimizable public functions	34,701	86.28%
Timeout	1,778	4.42%
Errors	1,001	2.49%
Number of public functions	40,219	100%
Vulnerable External Calls		
Property	# of func.	% of func.
Public functions with external calls	2,558	6.36%
Public functions that execute `CALL`	2,147	5.34%
Public functions potentially vulnerable	1,688	4.19%

Observe that we get a reduction with respect to $\mathcal{U}_{\mathcal{G}}$ of Example 3 because the accesses to `totTok` are only performed once and the cost of these accesses is moved from (c) in Example 3 to the constant gas cost (e'). ■

VI. EXPERIMENTS

This section provides the experimental results obtained from the analysis of more than 5,000 real smart contracts. Section VI-A contains some statistics generated using the cost models and profilers defined in the previous sections. The results show that they can be used not only to identify storage optimizations but also to detect several well-known security vulnerabilities. Section VI-B compares our approach and the results obtained in our experimental evaluation with different state-of-the art tools.

A. EXPERIMENTAL EVALUATION

The experiments have been executed on an Intel Core i7-7700T at 2.9GHz x 8 and 7.7GB of Memory, running Ubuntu 16.04. Our tool is able to analyze smart contracts written in Solidity (from versions 0.4.0 until 0.7.4[8]) or bytecode for the Ethereum Virtual Machine up to version 1.9.23. We use PUBS [1] to solve the cost relation system produced by our analysis and compute the upper-bound expressions.

We downloaded the last 1,800 verified open-source smart contracts from Etherscan service whose source code was available at 15^{th} October 2020. From them, we removed those contracts that raise a compiler error and those whose source code was stored as a *json* dictionary rather than a Solidity file. After this process, we have obtained 5,675 real smart contracts stored in 1,557 Solidity files. These contracts contain 40,219 public functions that have been analyzed using different cost models and profilers getting the results shown in Table 1.

A 5.23% of the smart contracts were written using the version 0.4 of the compiler, 44.46% using the version

[8]Latest version released up to October 2020

0.5, 35.95% using the version 0.6 and 14.36% using the version 0.7.

The results are split in three sections: **Resource Analysis**, **Optimization** and **Vulnerable External Calls**. Each section contains three columns: the first one shows a description of the results of the analysis applied; the second column shows the number of public functions inferred for each case; and the third column contains the percentage of each case with respect to the total number of public functions analyzed. Using the cost model $\theta_{\mathcal{I}}$ (without a profiler) we can infer if a function is constant or if it is parametric with respect to any of its input parameters. As shown in the first two rows of Table 1, 80.84% of the analyzed functions have a constant upper-bound while 9.02% are parametric. From those functions whose bound is parametric, we get that 98% have a linear bound: 71.37% whose cost depends on the value of a state variable, 13.84% on input variables and 14.79% on data related to the execution context (such as `CALLDATASIZE`). As we have seen, these bounds can be used, not only to prevent the users from a potential out-of-gas error, but also to determine the valid input values that can be used to successfully execute the transaction. The third row shows the number of public functions that reach a timeout (set to 60 seconds). In all cases, the timeout is reached during the process of solving the cost equation system to get the closed-form upper-bound. The last row shows the number of public functions that raise any of the errors produced by the solver. These errors may occur, among other reasons, due to a loss of precision in the decompilation phase (see Section III-B) or limitations of the solver (e.g. the bit-wise operations are not handled by the solver and have to be abstracted).

In section **Optimization** in Table 1, we show the number of public functions that can be optimized according to Definition 5. For a 16.68% of the optimizable public functions we are able to optimize more than one state variable. As before, 4.42% of the functions analyzed reach a timeout and the solver raises an error for 2.49% of them. We have proved that the 2,739 optimized public functions consume less gas than the original ones by using the cost model $\theta_{\mathcal{G}}$. Furthermore, we have checked that (i) all state variables that have been safely optimized correspond to basic type variables; and (ii) that all the *gas-expensive* storage accesses in the public functions analyzed that correspond to basic type variables have been optimized if the conditions described in Section V hold (i.e., it does not infer neither false positives nor false negatives, respectively).

Finally, in the third section of Table 1, **Vulnerable External Calls** we show the number of functions (first row) that execute a bytecode that performs an external call (`CALL`, `DELEGATECALL`, `STATICCALL`) and the number of functions that might execute the `CALL` bytecode (second row). Note that the second row is a subset of the first row of this section. The number of functions that make an external call and those that execute the bytecode `CALL` are computed using the cost model $\theta_{\mathcal{I}}$ and the profiler $\mathcal{P}_{\mathcal{B}}$. For the number of functions that make an external call, the computed upper-bound $\mathcal{U}^{\mathcal{B}}_{\mathcal{I}}|_{\{\texttt{CALL},\texttt{STATICCALL},\texttt{DELEGATECALL}\}}$ has to be greater than 0. The number of functions that execute `CALL` bytecode is a subset of the previous one and is computed using the upper-bound $\mathcal{U}^{\mathcal{B}}_{\mathcal{I}}|_{\{\texttt{CALL}\}}$. In the last row of the section, we consider a function as *potentially vulnerable* if it has a call to an external contract and the call is not generated with the instructions **`send`** or **`transfer`**. These instructions avoid re-entrancy recursive attacks as they only transfer 2,300 units of gas, the minimal amount to execute an external function. called contract. This amount of gas makes impossible to execute any instruction located in the fallback method of the invoked contract. Although the inference of this information is straightforward if the solidity file is available, these instructions are translated into the same bytecode `CALL` at **EVM** level. To identify the `CALL` bytecodes that come from a **`send`** or **`transfer`** instruction we have defined the following cost model $\theta_{\mathcal{SD}}$:

$$\theta_{\mathcal{SD}}(b) = \begin{cases} stack[3] & \text{if } b \equiv \texttt{CALL} \\ 0 & \text{otherwise} \end{cases}$$

This cost model $\theta_{\mathcal{SD}}$ takes as its cost the amount of gas provided to the external call to be executed (stored in the 3 top-most stack location) if the bytecode is `CALL` and 0 otherwise. Thus, using the cost model $\theta_{\mathcal{SD}}$ and the profiler $\mathcal{P}_{\mathcal{S}}$ we identify the source of the bytecode `CALL`. We have checked that we do not get false negatives, i.e., all the 459 public functions out of the 40,219 analyzed that have a **`send`** or **`transfer`** instruction in its source code are identified as *secure*.

TABLE 2. Performance statistics of the analyses executed.

Task	Time (s)
Resource analysis	163,807.26
Optimization analysis	160,843.35
Optimization translation	0.28
External call analysis	156,273.44
Call execution analysis	157,045.32
Secure external call analysis	158,998.66
Critical operation analysis	157,717.42
Arbitrary code analysis	156,816.53
Miners affected analysis	158,204.82

The information related to the efficiency of the tool is shown in Table 2. The time needed in the generation of the intermediate representation (it is executed just one time) is negligible compared to the time taken by the analyses. The resource analysis referenced in Table 1 takes 163,807.26s, the optimization analysis takes 160,843.35s and the optimizer takes 0.28s. Finally, the cost model and profilers used in upper-bounds generated in the vulnerability analyses carried out are similar and the analyses take 158,998.66s in the worst case.

Note that a function takes 4.07s to be analyzed in average. The time that our tool needs to analyze a function depends on the cost model and the profiler used. If we compare the average of the efficiency results presented in [3] using the cost model $\theta_{\mathcal{G}}$ (without any profiler) with those obtained by the other cost models, it is clear that the solver needs less time to infer the upper-bounds.

B. COMPARISON WITH OTHER TOOLS

In addition, we have also have compared some of the results obtained in Table 1 with the output of three open-source tools: Solidity compiler **solc**, the static analyzer of Remix [9] and Oyente [26]. The cost models $\theta_{\mathcal{I}}$ and $\theta_{\mathcal{G}}$ allow us to know if any loop of the functions analyzed depends on a state or input variable. In comparison to Remix and **solc** we claim that our resource analysis produces more precise results, as they only infer a precise bound for the public functions whose gas cost is constant. In the case of constant gas bounds, we get the same precision as **solc** and Remix. However, if the bound is parametric, they only return ∞ as gas bound, showing a warning to the programmer during the development phase (not for already compiled contracts). In addition, the gas static analyzer of Remix is only able to spot potential vulnerabilities when the state or input variables is directly involved in the loop. If it is shadowed using a local variable (as showed in Example 6 above), it is not able to identify it as potentially vulnerable. A 16.07% out of the 3,627 public functions whose upper-bound is parametric (see Table 1) are not detected as potentially vulnerable by Remix static analyses.

Oyente [26] is one of the most popular open-source EVM analyzers. It is focused on finding five types of vulnerabilities: (1) transaction-ordering dependencies occur when the final state of a transaction depends on the execution order of two instructions, (2) timestamp dependencies happen when the EVM instruction `TIMESTAMP` is involved in external calls or conditional statements, (3) exceptions dependencies occur when there are unchecked return values of external calls, (4) reentrancy vulnerabilities, and (5) integer overflows vulnerabilities. Oyente provides programming patterns to fix the identified vulnerabilities. In addition, it is able to compute the gas consumed by a transaction and the number of EVM instructions executed. However, it is based on symbolic execution, the CFG that Oyente generates to build the traces is not complete (losing execution paths), it limits the number of nodes and the depth of the CFG and it unwinds the loop to a fixed limit. In addition, it does not handle conditional gas consumption and some of the EVM bytecodes considered are underpriced (see [3] and [37] for more details). Thus, the tool is neither sound nor complete and the results reported would not be valid for certifying gas consumption.

Regarding the 2,558 vulnerable external calls (reentrancy vulnerabilities) shown in Table 1, Remix static analyzer reaches the same precision as our tool, i.e., it identifies the same 1,688 public functions as potentially vulnerable (those where the external calls are not performed with the instructions **`send`** or **`transfer`**) out of the 2,147 that execute a `CALL`. As Remix works at Solidity level it cannot be used for already deployed contracts. On the contrary, Oyente works at the level of EVM bytecode, but it spots a high number of false positives because it is not able to identify if a `CALL` comes from a **`send`** or **`transfer`** function and, as a consequence, it marks all the functions that might execute a `CALL` as vulnerable.

[9] https://remix.ethereum.org/

TABLE 3. Vulnerable functions according to the definitions of [25] and [24].

Vulnerable functions		
Vulnerability	# of func.	% of func.
Execution of critical operations	2,337	5.81%
Execution of arbitrary code	575	1.43%
Bytecodes affected by miners	1,307	3.25%

Concretely, Oyente considers the 2,147 public functions that execute a `CALL` shown in Table 1 as vulnerable. Thus, it spots the 459 public functions that we identify as *secure*, as potentially vulnerable.

Some other tools with similar purposes such as GASPER [13], GasReducer [14], teEther [25] or Zeus [24] are not open-source or they are not available online so we cannot make a deeply experimental comparison. Anyway, our analysis is able to further assist the contract developer for detecting some additional vulnerabilities: For instance, if we compute the upper-bound $\mathcal{U}^{\mathcal{B}}_{\mathcal{I}}|_{\{\texttt{CALL},\texttt{SELFDESTRUCT}\}}$, we infer the number of critical operations (as defined in [25]) executed in the functions analyzed. We can also use the upper-bound $\mathcal{U}^{\mathcal{B}}_{\mathcal{I}}|_{\{\texttt{CALLCODE},\texttt{DELEGATECALL}\}}$ to known the number of public functions that may execute arbitrary code (also defined in [25]). And thanks to the upper-bound $\mathcal{U}^{\mathcal{B}}_{\mathcal{I}}|_{\{\texttt{TIMESTAMP},\texttt{COINBASE},\texttt{NUMBER},\texttt{DIFFICULTY},\texttt{GASLIMIT}\}}$ we can infer if a public function may execute some of the bytecodes that can be modified by miners (defined in [24]), making the contract vulnerable. The results of the analysis of the 40,219 public functions using the above cost models and profilers can be seen in Table 3.

We have confirmed that there are no more public functions that execute bytecodes related to the vulnerabilities analyzed, i.e., our approach identifies all public functions potentially vulnerable in the dataset studied. In addition, we have manually studied 500 random public functions for all cases shown in Table 3 to confirm that the bytecodes involved in the corresponding vulnerability are executed. We have verified that the Solidity code that involves these bytecodes is not opaque or dead. Thus, the bytecodes involved in the vulnerability might be eventually executed. Note that some tools, like GASPER [13], are able to analyze if a Solidity file has opaque or dead code, but unluckily they are not publicly available to compare their results.

VII. CONCLUSION AND FUTURE WORK

In this work, we present the notion of static profiler applied on Ethereum smart contracts. The flexibility of the profilers allows us to handle those properties that can be modeled within the information statically available at EVM level. We have formalized several cost models and profilers that can be used within a resource analyzer to generate different *sound* upper-bounds on a variety of resources (such as the number of storage instructions, gas consumed by some EVM operations, total ether sent by an external call, etc.). These upper-bounds provide useful metrics that can be used by developers and users. In addition, it can be used to optimize smart contracts (when the source code is available) by analyzing the accesses

to storage. It proposed a sound transformation that replaces the accesses to storage by accesses to memory that consume less gas. We have applied them to analyze more than 40,000 real public function of smart contracts getting that a 9.02% are parametric, a 6.81% of them can be optimized, and 4.19% may be potentially vulnerable.

An interesting direction for future work is to improve the precision of our tool optimizing, not only basic type variables, but also complex type variables such as arrays, structs or maps. In addition, we plan to relax the conditions defined to optimize storage accesses in Section V and generalize the optimization to functions which access to the state variables in other functions of the contract different to the one under analysis. Additionally, we would study the applicability of the optimization at EVM level, which is a more complex case as we would modify the structure of the EVM bytecode and it may affect to the size and the addresses of the original bytecode.

REFERENCES

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla, "Closed-form upper bounds in static cost analysis," *J. Automated Reasoning*, vol. 46, no. 2, pp. 161–203, Feb. 2011.

[2] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "GASOL: Gas analysis and optimization for Ethereum smart contracts," in *Proc. 26th Int. Conf. Tools Algorithms Construct. Anal. Syst.*, in Lecture Notes in Computer Science, vol. 12079. Dublin, Ireland: Springer, 2020, pp. 118–125.

[3] E. Albert, P. Gordillo, A. Rubio, and I. Sergey, "Running on fumes: Preventing out-of-gas vulnerabilities in Ethereum smart contracts using static resource analysis," in *Proc. 13th Int. Conf. Verification Eval. Comput. Commun. Syst. (VECoS)*, in Lecture Notes in Computer Science, vol. 11847. Porto, Portugal: Springer, 2019, pp. 63–78.

[4] D. E. Alonso-Blas and S. Genaim, "On the limits of the classical approach to cost analysis," in *Static Analysis* (Lecture Notes in Computer Science), vol. 7460, A. Miné and D. Schmidt Eds. Deauville, France: Springer, 2012, pp. 405–421.

[5] N. Ambroladze, "Fast and scalable analysis of smart contracts," M.S. thesis, Swiss Federal Inst. Technol., Zürich, Switzerland, 2018.

[6] M. M. A. Aldweesh, M. Alharby, and A. V. Moorsel, "OpBench: A CPU performance benchmark for Ethereum smart contract operation code," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Jul. 2019, pp. 274–281.

[7] R. Bagnara, M. P. Hill, and E. Zaffanella, "The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems," *Sci. Comput. Program.*, vol. 72, nos. 1–2, pp. 3–21, 2008.

[8] C. Boogerd and L. Moonen, "Prioritizing software inspection results using static profiling," in *Proc. 6th IEEE Int. Workshop Source Code Anal. Manipulation*. Washington, DC, USA: IEEE Computer Society, Sep. 2006, pp. 149–160.

[9] C. Boogerd and L. Moonen, "On the use of data flow analysis in static profiling," in *Proc. 8th IEEE Int. Work. Conf. Source Code Anal. Manipulation*, Sep. 2008, pp. 79–88.

[10] G. Canfora, A. D. Sorbo, S. Laudanna, A. Vacca, and C. AaronVisaggio, "Gasmet: Profiling gas leaks in the deployment of solidity smart contracts," *CoRR*, vol. abs/2008.05449, pp. 1–13, Dec. 2020.

[11] J. Charlier, S. Lagraa, R. State, and J. François, "Profiling smart contracts interactions tensor decomposition and graph mining," in *Proc. 2nd Workshop Mining Data Financial Appl. Eur. Conf. Mach. Learn. Princ. Pract. Knowl. Discovery Databases*, Skopje, Macedonia, vol. 1941, Sep. 2017, pp. 31–42.

[12] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "GasChecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Trans. Emerg. Topics Comput.*, early access, Mar. 6, 2020, doi: 10.1109/TETC.2020.2979019.

[13] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*. Washington, DC, USA: IEEE Computer Society, Feb. 2017, pp. 442–446.

[14] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *Proc. 40th Int. Conf. Softw. Eng., New Ideas Emerg. Results*, Gothenburg, Sweden, May 2018, pp. 81–84.

[15] Ethereum. (2018). *Solidity*. [Online]. Available: https://solidity.readthedocs.io

[16] A. Flores-Montoya and R. Hähnle, "Resource analysis of complex programs with cost equations," in *Programming Languages and Systems* (Lecture Notes in Computer Science), vol. 8858. Singapore: Springer, 2014, pp. 275–295.

[17] R. W. Floyd, "Assigning meanings to programs," in *Program Verification*. Dordrecht, The Netherlands: Springer, 1993, pp. 65–81.

[18] A. Garcia, C. Laneve, and M. Lienhardt, "Static analysis of cloud elasticity," in *Proc. 17th Int. Symp. Princ. Pract. Declarative Program.*, Siena, Italy, Jul. 2015, pp. 125–136.

[19] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and A. Smaragdakis, "Madmax: Surviving out-of-gas conditions in Ethereum smart contracts," in *Proc. PACMPL*, 2018, pp. 116:1–116:27.

[20] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of Ethereum smart contracts," in *Principles of Security and Trust* (Lecture Notes in Computer Science), vol. 10804. Thessaloniki, Greece: Springer, 2018, pp. 243–269.

[21] 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," in *Proc. ACM Program. Lang.*, vol. 2, Jan. 2018, pp. 1–28.

[22] R. Haemmerlé, P. López-García, U. Liqat, M. Klemen, J. P. Gallagher, and M. V. Hermenegildo, "A transformational approach to parametric accumulated-cost static profiling," in *Functional and Logic Programming* (Lecture Notes in Computer Science), vol. 9613. Kochi, Japan: Springer, 2016, pp. 163–180.

[23] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, London, U.K., Nov. 2019, pp. 531–548.

[24] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.* Reston, VA, USA: Internet Society, 2018, pp. 1–12.

[25] J. Krupp and C. Rossow, "Teether: Gnawing at Ethereum to automatically exploit smart contracts," in *Proc. USENIX Secur. Symp.* Berkeley, CA, USA: USENIX Association, 2018, pp. 1317–1333.

[26] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 254–269.

[27] M. Marescotti, M. Blicha, A. E. J. Hyvärinen, S. Asadi, and A. N. Sharygina, "Computing exact worst-case gas consumption for smart contracts," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice* (Lecture Notes in Computer Science), vol. 11247. Limassol, Cyprus: Springer, 2018, pp. 450–465.

[28] R. G. Morgan and S. A. Jarvis, "Profiling large-scale lazy functional programs," *J. Funct. Program.*, vol. 8, no. 3, pp. 201–237, May 1998.

[29] J. Nagele and M. A. Schett, "Blockchain superoptimizer," in *Proc. 29th Int. Symp. Logic-Based Program Synth. Transformation (LOPSTR)*, 2019, pp. 1–15. [Online]. Available: https://arxiv.org/abs/2005.05912

[30] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 653–663.

[31] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *Verification, Model Checking, and Abstract Interpretation* (Lecture Notes in Computer Science). Venice, Italy: Springer, 2004, pp. 239–251.

[32] C. Signer, "Gas cost analysis for Ethereum smart contracts," M.S. thesis, Swiss Federal Inst. Technol., Zürich, Switzerland, 2018.

[33] K. Toyoda, K. Machi, Y. Ohtake, and A. N. Zhang, "Function-level bottleneck analysis of private proof-of-authority Ethereum blockchain," *IEEE Access*, vol. 8, pp. 141611–141621, 2020.

[34] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 67–82.

[35] B. Wegbreit, "Mechanical program analysis," *Commun. ACM*, vol. 18, no. 9, pp. 528–539, Sep. 1975.
[36] X. Wei, C. Lu, F. R. Ozcan, T. Chen, B. Wang, D. Wu, and Q. Tang, "A behavior-aware profiling of smart contracts," in *Security and Privacy in Communication Networks Security and Privacy in Communication Networks* (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering), vol. 305. Orlando, FL, USA: Springer, 2019, pp. 245–258.
[37] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Project Yellow Paper, 2014, vol. 151, no. 2014, pp. 1–32.
[38] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *Proc. MICRO. 27th Annu. IEEE/ACM Int. Symp. Microarchitecture*. New York, NY, USA: Association Computing Machinery, Dec. 1994, pp. 1–11.
[39] S. Zekany, D. Rings, N. Harada, M. A. Laurenzano, L. Tang, and J. Mars, "CrystalBall: Statically analyzing runtime behavior via deep sequence learning," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

JESÚS CORREAS received the M.S. degree in computer science and the Ph.D. degree in computer languages and systems from the Universidad Politécnica de Madrid, Spain, in 2000 and 2008, respectively.

He was employed as an Assistant Professor with the Complutense University of Madrid and the Universidad Politécnica de Madrid. He has also worked in private software development companies. He has been a tenured Lecturer with the Complutense University of Madrid, since 2009. His main research interests include static program analysis, constraint declarative programming, and object-oriented concurrent and distributed systems. He has been involved in several national and European research projects.

PABLO GORDILLO was born in Madrid, Spain, in 1992. He received the B.S. degree in mathematics and computer science and the M.S. and Ph.D. degrees in computer science from the Universidad Complutense de Madrid, Spain, in 2015, 2017, and 2020, respectively.

He has been a Postdoctoral Researcher with the Universidad Complutense de Madrid, since 2020. He is also a Research Member of the COSTA Group, since 2014. His research interests include static and dynamic analyses, formal methods, testing and verification of concurrent programs, and distributed systems. He is also working on analysis and verification of Ethereum smart contracts. He has been involved in several national and European research projects.

GUILLERMO ROMÁN-DÍEZ received the B.S. and M.S. degrees in computer science and the Ph.D. degree in software and systems from the Universidad Politécnica de Madrid, Spain, in 2004, 2008, and 2012, respectively.

He was an Assistant Professor with the Universidad Politécnica de Madrid, in 2016, where he has been a tenured Lecturer since 2020. He was employed as a Postdoctoral Researcher with the Universidad Politécnica de Madrid and in private companies involved in software development projects. His main research interests include program analysis, namely, and static resource analysis object-oriented concurrent and distributed systems. He is involved in national and European research projects, and the participation in the design and implementation of the COSTA and SACO systems among others.