Towards Automatic Inference of Gas Bounds for Smart Contracts on the Ethereum Blockchain

Elvira Albert[1], Albert Rubio[2], Ilya Sergey[3], and Thomas Sibut-Pinote[3]

[1] Complutense University of Madrid, Spain
[2] Universitat Politècnica de Catalunya, Spain
[3] University College London, UK

Abstract. We present an application of SACO—a resource Static Analyzer for Concurrent Objects—to infer *gas bounds* on Ethereum smart contracts written in Solidity. We identify a programming pattern in smart contracts, supported by real-world case studies, that requires an accurate estimate of consumed *gas* for the reasons of contract execution safety. Those case studies pose a challenge for the cheap-and-cheerful approach for estimating the amount of gas, implemented in the state-of-the-art Solidity compiler. We then demonstrate how our translation of smart contracts from Solidity into concurrent objects allows one to automatically derive parametric gas bounds.

1 Introduction

Blockchain consensus, a family of distributed protocols for building a replicated transaction log (a *blockchain*), is a technology that made possible the creation of decentralised cryptocurrencies, such as Bitcoin [19]. In Ethereum [22], one of Bitcoin's most prominent successors, the blockchain has also been employed as a medium for implementing user-defined replicated stateful computations associated with funds-exchanging transactions—so-called *smart contracts* [4].

Smart contracts are small programs stored in a blockchain that can be invoked by certain transactions, initiated by the parties involved in the protocol, executing some business-logic as automatic and trustworthy mediators. Typical applications of smart contracts involve implementations of multi-party accounting, voting and arbitration mechanisms, auctions, and puzzle-solving games with distribution of rewards. For the consistency of the blockchain, every transaction must be validated by the majority of involved parties. In practice, it means that every computation involving an interaction with a smart contract is replicated across the system. Having contracts written in a general-purpose Turing-complete language [1, 3] would mean that such replicated computations could potentially diverge, leading to a denial-of-service of the nodes validating them, unless a uniform strategy to handle such diverging executions is adopted.

In Ethereum, the mechanism for dealing with potentially non-terminating smart contract executions is based on a notion of *gas*—a cost semantics of low-level contract instructions. Therefore, when proposing a transaction interacting with a contract, a node *pays* for the cost of its execution using Ether, the internal currency of Ethereum. Thus, the amount of Ether to be converted to gas must be determined statically, before the transaction is issued. If the allocated amount of gas is insufficient, the transaction will be aborted, yet the Ether spent will still be deduced from the account of the node that has proposed the transaction. In any event, the gas-allocated funds will be set to reward the nodes participating in validating, *i.e.*, evaluating, the transaction.

Hence, estimating the gas cost of calling a contract is of crucial importance for issuing a viable transaction in Ethereum. The challenge of computing a correct gas bound becomes particularly acute in the presence of an interplay between multiple distributed parties: for instance, it is not uncommon to call a contract which then transfers a certain amount of funds to *another* party, who will later use them to buy gas for running a *callback* in the same contract. Finally, it is not always possible to provide a constant gas estimate: in the presence of concurrent interaction in a blockchain protocol, the cost of calling a contract might vary dynamically, due to changes in the blockchain's state. Standard compilers for Solidity [1] and Viper [3], de-facto high-level smart contract languages in Ethereum, fail to provide accurate parametric gas bounds.

In this work, we demonstrate how to use SACO [8]—a framework for static resource analysis of *concurrent objects* written in the ABS language [17]—to obtain parametric gas bounds for Ethereum contracts. We showcase our approach in application to the *callback-oriented* implementation style, in which run-time contract safety depends on a statically obtained gas cost of its execution. We also report on using SACO for computing gas bounds for two real-world examples.

2 Inference of Gas Bounds for Smart Contracts

2.1 Encoding Smart Contracts as Concurrent Objects

Smart contracts behave as concurrent (or active) objects—a model for concurrency well studied by the formal methods community (see, *e.g.*, the survey [13] and its references). Like concurrent objects, smart contracts have an internal mutable state and methods that execute atomically (without interrupts). A contract can be accessed by multiple *accounts* or *addresses*, which may belong both to users or to other contracts, and which run transactions on the contract. Transactions are concurrent calls that execute a corresponding method of the contract. These transactions can modify the contract state and execute as in the *cooperative* concurrency model of concurrent objects. This implies that, although each transaction relinquishes control of its execution and thus cannot be interrupted, the order in which transactions are scheduled can lead to different outcomes.

Fig. 1 shows to the left an excerpt of the Solidity implementation of the BlockKing contract, a simple gambling game [5], and to the right the encoding using the concurrent objects language ABS [17]. Although BlockKing is not heavily used, it shows the use of the Oraclize service [10]—a service which enables contracts to communicate with the world outside the blockchain and for which concurrency issues arise. The gamble in BlockKing is as follows: at any given time there is a designated "Block King" (initially the owner of the contract). In the Solidity code, we can observe that the contract state is formed by the data declared in Lines 2-3 (L2-3 for short), where references to the owner and king users are kept, among other contract fields. When a user wants to send money to the contract, the default function executes L5 and invokes method enter L6. Some implicit parameters, such as msg.send and msg.value, are stored in the

```
 1 contract BlockKing is usingOraclize{
 2  address public owner, king, ...;
 3  uint public warriorBlock, randomNum ...;
 4  function BlockKing() {...}
 5  function() {enter();}
 6  function enter() {
 7      if (msg.value < 50 finney) {
 8         msg.sender.send(msg.value);
 9         return;}
10      warrior = msg.sender;
11      warriorGold = msg.value;
12      warriorBlock = block.number;
13      bytes32 myid = oraclize_query(...);}
14  function __callback(bytes32 myid,...) {
15      ...
16      process_payment(); }
17  function process_payment() {
18      uint singleDigit = warriorBlock;
19          while (singleDigit > 1000000) {
20                  singleDigit −= 1000000;}
21          while (singleDigit > 100000) {
22                  singleDigit −= 100000; }
23          ...
24          if (singleDigit == 10) {
25                  singleDigit = 0; }
26          ...}
27      king.send(reward);
28      owner.send(contractAddress.balance);
29   }
30 }
```

```
31 class BlockKing{
32  Address owner, king,..;
33  Int warriorBlock, randomNum..;
34  Oraclize o...;
35  Int balance=1000;
36  init(){...}
37  Unit enter(AddressI msg_sender,
38        Int msg_value, Int block_number) {
39   if (msg_value <50) {
40          msg_sender!send(msg_value);
41     } else{
42          warrior = msg_sender;
43          warriorGold = msg_value;
44          warriorBlock = block_number;
45          o!query(this);} }
46  Unit callback() {
47      ...
48      process_payment();}
49  Unit process_payment() {
50      Int singleDigit = warriorBlock;
51      while (singleDigit > 1000000) {
52          [cost==c(D)]
53         singleDigit = singleDigit−1000000;}
54      while (singleDigit > 100000) {
55          singleDigit = singleDigit−100000;}
56      ...
57      if (singleDigit == 10) {
58          singleDigit = 0;
59      ...
60      king!send(reward);
61      owner!send(balance);}
62 }
```

Fig. 1. Excerpt of BlockKing Contract. Solidity implementation (left). Concurrent objects implementation in ABS (right).

contract fields, as well as the transaction block number that is accessed in L12 and stored in the object state. Then, the Oraclize service is invoked in L13 to obtain a random number between 1 and 9. This invocation raises an observable event, handled by the Oraclize service provider, so it later makes a call to the designated callback point (method __callback). L19-24 in __callback compute the current block number modulo 10, and if this number is equal to the random number provided by Oraclize, the sender becomes the new Block King, and the sender gets a reward L27 and the owner gets the current balance L28.

An important point to note is that if multiple gamblers send transactions in a short period of time, they may overwrite the previous gambler data, such that when callbacks occur, the last gambler will enjoy multiple chances to win the reward (see [20] for a more detailed explanation). The same interleavings are captured by our ABS encoding showed to the right of the figure. The following differences between the Solidity and ABS implementations are purely syntactic:

1. Contracts are concurrent objects: we use "class Name" (see L31) to define a contract and create the contract using new Name().
2. Accounts are concurrent objects of class Address (L32) and the Oraclize service is a concurrent object of class Oraclize that serves the queries (L34).
3. Method init makes default initalization (L36).
4. The implicit parameters msg of smart contracts, and the block number, become explicit parameters of the entry method in the transactions (L37).

5. The `balance` of the contract becomes an explicit field of the ABS class (L35).
6. We use asynchronous method calls, denoted `o!m`, to asynchronously invoke method `m` on object `o` (L61).

Although we have not implemented a syntactic translation from Solidity contract to equivalent ABS models yet, we believe it is possible, due to the straightforward nature of the mappings described above. Such a translation would give a formal semantics to Solidity contracts in terms of concurrent objects [21].

2.2 Gas Bounds Analysis

A main advantage of generating ABS models from smart contracts is that existing advanced dynamic and static tools to reason about qualitative and quantitative properties of ABS models [2] can now be applied to smart contracts. In this section, we focus on the particular quantitative property of gas consumption and show how to use an off-the-self resource analyzer for ABS [7,8].

User-defined cost model. SACO [8] is a static analysis tool for concurrent objects. It includes a generic resource analyzer which computes *upper bounds* for the resource consumption (as a parametric function of the input data sizes) of executing concurrent object systems. The analyzer is generic *wrt.* the type of resource one wants to measure, *aka* the cost model. It already integrates cost models to count number of executed steps, amount of memory allocated, etc. In this work, we employ SACO's support for *user-defined* cost models. A user-defined cost model allows one to write annotations in the code of the form `[cost==c(D)]`. For instance, the annotation in L52 represents the gas consumption of executing the subtraction operation under the annotation. Given this annotation, SACO computes as upper bound `c(D)*(warriorBlock-1)/1000000`, which corresponds to the number of iterations of the `while` loop multiplied by the gas consumption of the subtraction. Therefore, for each type of instruction `I`, we include a cost annotation `[cost==c(I)]` and the analyzer computes an upper bound over-approximating the number of instructions of each type executed in the worst case multiplied by the actual gas consumed by `I`, denoted `c(I)`. Analyzing the `__callback` method, we obtain the following upper bound:

$$c(D)*((warriorBlock-1)/1000000+(warriorBlock-1)/100000)+c(S)*2+c(A)+c(C)+\dots$$

Intuitively, it counts the gas consumed by the decrements the 2 loops executed in `__callback`, plus 2 `send`-operations (represented by $c(S)$) an assignment ($c(A)$), a conditional ($c(C)$), plus the gas consumed by the code that we have omitted.

Smart contracts bounds. The above bound reveals two important features of gas bounds of smart contracts, namely they may depend on:

– *the contract state*: The gas bound above is parametric on the *warriorBlock*. Due to the interleaving of blockchain transactions, one gambler might overwrite the data of a previous gambler, the actual gas cost will be determined by a value set by a *different* gambler than the one being charged for the gas.
– *the state of the blockchain (*e.g.*, the current block number,* etc.*)*: This indeed happens in the running example too. The initialization value of the *warriorBlock* corresponds to the block number of the last transaction that sent money to the contract, which might not correspond to the block number of the transaction that will be charged for the gas.

Thus, inferring gas bounds is not only useful to accurately estimate the amount of gas that must be put for successfully running the transaction [12], but it may also reveal unexpected behaviours in the code that arise due to the concurrent execution of smart contracts [20], *e.g.*, a gambler may be charged of a gas consumption that depends on *other* gambler's parameters or on the blockchain.

3 Validation of Results

As described in Section 2, our analysis works on a contract code written in ABS, translated from Solidity sources. How do the obtained gas bounds relate to those observed at run time on the blockchain? In this section, we validate our approach empirically by comparing the upper bounds we obtain using our method with the actual gas consumption of the __callback methods of two contracts. In order to obtain gas bounds for each instruction `I` in the Solidity code, we use the Solidity compiler `solc` to obtain a corresponding sequence of EVM bytecode [1]. From this bytecode, compute a gas cost estimate for each instruction following the gas cost specifications of Ethereum [22]. We then substitute this gas cost for each `c(I)` in the formula produced by SACO.

BlockKing. The BlockKing contract implements a very inefficient "modulo 10" operation on positive integers through the use of six consecutive while-loops. To analyze it, we use only the naïve annotations described in Section 2.1. We bound the variable blockID by the number of the last Ethereum block in which BlockKing was called, which is 1,278,878. We obtain a bound of the order of magnitude of $6,300,000$ gas, which is unreasonably large. The reason for this is that the size analysis in SACO does not work with modulo operations and is not able to infer the amount by which the value of singleDigit has decreased at the end of each loop. For example, when we get to Line 54 we do not employ the information that singleDigit $< 1,000,000$ even though this is the case. This leads to an overly pessimistic estimate. Explicitly annotating the modulo operations computed by the loops, allows SACO to produce a bound of $102,640$. The actual gas cost of the __callback function execution is around $60,000$ gas.

EthereumPot. The EthereumPot contract [6] implements a simple lottery. During a game, players, represented by arbitrary Ethereum accounts, call a method joinPot to buy (one or more) lottery tickets; each player's address is appended to a variable-length array addresses of current players, and the number of tickets is appended to an array slots. After some time has elapsed, anyone can call a rewardWinner method 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 a __callback method with this random number as an argument. A new instance of the game is then started, and the winner is allowed to withdraw their balance using a withdraw method.

The __callback function was called 12 times in the history of the EthereumPot contract. The biggest gas consumption was $173,870$, which is significantly higher than the other values: this can be explained by the fact that several variables were initialized for the first time with the SSTORE instruction, with the high cost of $20,000$ gas each. The smallest gas consumption was $88,413$ gas.

Players	2	2	2	4	2	2	2	2	3	5	2	3
Gas	173870	105986	96295	107114	88413	95915	90639	90706	98107	101274	98554	88415

We obtain the upper bound $245,935 + 650 \cdot slots$ for the __callback method of the EthereumPot contract, where *slots* is the number of Ethereum addresses who bought tickets. Since __callback ends by a for loop over all slots to find the winner, part of this cost is proportional to the number of players. Note that this formula is simplified and neglects a few operations related to the checking of the cryptographic proof supplied by Oraclize.

Discussion. Indeed, there are many ways to compile a given Solidity instruction to EVM bytecode and thus there is no single way of assigning a gas cost to a Solidity instruction: depending on the version of the compiler and on the level of optimization, this may vary greatly. Furthermore, the gas cost of some opcodes is context-dependent. For example, the opcode SSTORE for storing a value v in a contract-level variable x costs $20,000$ gas to execute when $v \neq 0$ and $x = 0$. However, when $x \neq 0$ and $v = 0$, which is the default value of any uninitialized variable, storing v in x actually amounts to *deleting* x: since this deletion frees up memory, it gives the caller a $15,000$ gas refund. Finally, in other cases updating x to value v only costs $5,000$ gas. Thus, if the value of v cannot be obtained statically, one can only give an upper bound of $20,000$ gas.

Because of this context dependency and due to the mismatch between the estimated costs at the level of Solidity (via SACO) and at the level of EVM, our approach will necessarily obtain upper bounds which are sometimes much higher than the actual gas consumption. As we are reasoning at the level of Solidity code, what we obtain cannot be considered as a rigorous upper bound on *any* compilation of a program to EVM code, but rather as an order of magnitude of an upper bound on what the standard `solc` compiler would produce.

4 Related Work and Conclusions

A number of formalizations of EVM and Solidity were implemented in Coq [16], Isabelle/HOL [9], F* [11], K [15], and in custom tools for static and dynamic analysis of smart contract behaviours [14, 18]. Of all those systems, the closest to ours are Oyente [18] and KEVM [15]. Oyente performs automatic analysis of contracts, yet, it does not derive the gas usage boundaries. The KEVM suite provides a program verifier that is capable of deriving the gas complexity of the program by implementing a simple gas-counting runtime monitor. Their analysis summarizes the gas and memory used for each basic block of an EVM program, that is, the derived boundaries are not given in terms of field sizes of a contract.

In this paper, we presented a proof-of-concept approach for using a state-of-the-art resource analyzer to infer over-approximations of the gas consumption of smart contracts on Ethereum blockchain. We have demonstrated our approach on two case studies by encoding manually the Solidity programs into ABS and using *cost* annotations to track the gas cost of ABS instructions. Our ABS models can be accessed online from: http://costa.ls.fi.upm.es/saco/web, in the folder SmartContracts.

References

1. Solidity. https://solidity.readthedocs.io/en/develop, January 18, 2018.
2. The ABS tool suite. http://abs-models.org.
3. Viper. https://viper.readthedocs.io/en/latest, January 18, 2018.
4. Smart Contracts: Building Blocks for Digital Markets, 1996.
5. The BlockKing contract, 2016. Availableathttps://etherscan.io/address/0x3ad14db4e5a658d8d20f8836deabe9d5286f79e1.
6. The EthereumPot contract, 2017. Available at https://etherscan.io/address/0x5a13caa82851342e14cd2ad0257707cddb8a31b7.
7. E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, and G. P. an d Guillermo Román-Díez. Object-Sensitive Cost Analysis for Concurrent Objects. *Software Testing, Verification and Reliability*, 25(3):218–271, 2015.
8. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *TACAS*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.
9. S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *CPP*, pages 66–77. ACM, 2018.
10. T. Bernani. Oraclize, 2016. http://www.oraclize.it.
11. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *PLAS*, pages 91–96. ACM, 2016.
12. T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER*, pages 442–446. IEEE Computer Society, 2017.
13. F. S. de Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017.
14. S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL*, 2(POPL):48:1–48:28, 2018.
15. E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu. KEVM: A Complete Semantics of the Ethereum Virtual Machine. Technical report, 2017.
16. Y. Hirai. Ethereum Virtual Machine for Coq (v0.0.2). Published online on 5 March 2017, 2017.
17. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *FMCO*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.
18. L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *CCS*, pages 254–269. ACM, 2016.
19. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
20. I. Sergey and A. Hobor. A Concurrent Perspective on Smart Contracts. In *1st Workshop on Trusted Smart Contracts*, volume 10323 of *LNCS*, pages 478–493. Springer, 2017.
21. I. Sergey, A. Kumar, and A. Hobor. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR*, abs/1801.00687, 2018.
22. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. Available at http://gavwood.com/paper.pdf, 2014.