Fuzzing to Estimate Gas Costs of Ethereum Contracts

Daniel Soto, Alexandre Bergel, Alejandro Hevia

Department of Computer Science (DCC), University of Chile, Chile

danielsoto.3004@gmail.com http://bergel.eu https://users.dcc.uchile.cl/∼ahevia/

***Abstract*—This paper studies how a simple approach based on fuzzing testing can help authors of Solidity contracts to accurately estimate the gas cost of services specified in a contract. Our fuzzer creates a private blockchain and randomly generates transactions. Such an environment is meant to simulate large scale behavior that may be seen in a public blockchain. Our fuzzer handles Ethereum starting and target endpoints in a transaction to accommodate requirements expressed in financial contracts. By comparing the gas computation made by the Ethereum Solidity compiler and the actual consumption during our fuzzing, we are able to find discrepancies between predicted and real gas consumption. Our findings are beneficial to transaction authors to correctly predict the computing resources of Ethereum miners.**

I. Ethereum Gas Model

A blockchain is essentially a distributed database. Although originally designed to record financial transactions, a blockchain can ledge transactions about everything of value. Ethereum, a popular computing platform and operating system, exploits this feature by running applications in a consistent yet distributed way, while storing the state of programs on the blockchain [1].

Ethereum relies on the Ethereum Virtual Machine as the executing platform, which can run programs written the programming language Solidity. This language is similar to Java in many aspects. For example, a Solidity contract roughly looks like a Java class and a Solidity service is similar to a public Java method. A service may be executed either by another service, or by a transaction. As such, a transaction is similar to the classical remote method call. Here is an example of a contract:

```
1  contract SimpleStorage {
2     address owner;
3     uint amount;
4     constructor() public {
5        owner = msg.sender;
6     }
7     function pay() public payable {
8        require(owner == msg.sender);
9        amount += msg.value;
10    }
11 }
```

Line 4 - 6 defines a contract constructor, which gets executed when the contract is deployed on a blockchain. Line 5 records the sender of the transaction into a variable called `owner`. As such, the owner variable references an address that uniquely identifies the owner of the contract.

In a public blockchain, anyone can execute a transaction, for any public service. A public service may be designed to produce valuable data (*e.g.,* a record of a financial transaction, as in our example given above), which deserves to be stored in a blockchain. As an incentive for Ethereum network members (called *miners*) to execute transactions, Ethereum offers a monetary system, called *gas*, which has a price set in any transaction. A transaction includes payments for the gas its execution consumes. After executing a transaction, a miner *earns* the reward fee set by the transaction author, and if the execution succeeds, the result of the transaction is placed in the blockchain.

Ethereum also considers safety measures to prevent wasting miners' resources and denial of service attacks. If the transaction execution does not succeed, for example, if the service invoked by the transaction raises an error or takes "too long" to execute, then the miner earns the fee paid by the transaction author, and no record is added in the blockchain. In that case, the transaction author loses money without obtaining the hoped execution result. Indeed, a transaction has to provide a threshold called *gas limit*, that represents the maximum amount of resources used by the miner computing platform. The existence of such a limit is important as it ensures the miner that executing the contract service specified by the transaction is bound in terms of resources (CPU and time).

A gas is a unit describing elementary computational steps. A unit of gas corresponds to the execution of one atomic instruction: a bytecode. For example, multiplication is a simple operation that requires few computational units (5 gas) while a significantly more complex operation, such as getting the balance of a particular account costs 400 gas.[1]

The gas estimation problem. Solidity estimates the upper limit of the gas cost of a given contract by performing a static analysis on (the bytecode of) the contracts. Any imprecision in this estimation, however, may have dramatic consequences. In particular, (i) if the service raises an error or (ii) if gas limit is too low, then the miner gets to keep the fee without actually producing a useful result for the person who produced the transaction. This risk of losing money without having an actual result is well known among Solidity programmers. As a consequence, authors of contracts have a tendency to lower their risk by simplifying their contracts and reusing code that

[1]https://github.com/crytic/evm-opcodes

is known to work well by means of duplication across multiple Solidity codebases.

Related work. Other tools have been developed to estimate gas costs, such as GASTAP [2], which operates on the bytecode of a contract and generates formulas to compute the exact gas costs of executing a transaction, given the state of the contract. In contrast, our fuzzing tool generates a gas cost distribution, independent of the contract's state.

Contributions. This work presents a new approach to accurately estimate the gas cost of services specified in a Solidity contract by using fuzzing testing. Our fuzzing tool is able to discover the range of possible gas costs a service may have at execution. With this tool, we were able to identify a number of patterns in which the popular way of estimating the gas cost is wrong and may therefore be considered a potential threat.

This paper makes the following contributions:

- It defines a simple, although extensible fuzzing model for solidity contracts.
- It classifies most of the contracts available in the blockchain based on their expected gas costs, and actual gas costs.
- It analyses a few extraordinary examples, and proposes hypothesis on why estimations may be off.

New idea and emerging results. In this paper, we put forward the idea of applying fuzzing techniques to the Solidity programming language to estimate gas costs. Such application is not trivial for a number of reasons: (i) Solidity has some syntactic constructs that are unique, e.g., monetary transactions; (ii) any fuzzer needs to define a mock environment in which a blockchain can be properly handled to execute Solidity contracts. As such, classical fuzzers cannot be trivially ported to Solidity. The ability of the Solidity compiler to statically estimate the gas cost is a safety net to prevent wasting monetary and computational resources. Our emerging results indicate that the technique employed by the compiler can be imprecise in some cases, making estimations infinite whenever it can not determine an upper-bound, and we propose a viable and practical solution to this situation.

II. Methodology to estimate gas cost

Our hypothesis is that *fuzzing testing technique is efficient at better estimating the gas cost of Solidity services*. This section describes the methodology we designed to verify this hypothesis.

A. Fuzzing

Fuzzing testing [3] allows automating software testing by generating tests automatically. We use fuzzing testing on Solidity contracts by (i) randomly and uniformly generating values for primitive types, including numerical and boolean types, and (ii) generating transactions that randomly invoke Solidity services. As in most fuzzing techniques, we use a geometric distribution of size-variable values, such as strings and arrays. Our fuzzer is able to invoke a transaction by generating arguments of a contract service's signature, and a random address of the transaction emitter.

We chose to implement our own fuzzer over already available tools like Harvey [4], Echidna [5], ContractFuzzer [6] or GasFuzz [7], to better accomodate our VM choice and grant us a large amount of control over how parameters were generated. This approach also eases the addition of more sophisticated components later on, such as symbolic execution.

The fuzzer we designed and implemented for Solidity follows the traditional guidelines to build fuzzers [3] since Solidity shares many common constructions with most general purpose programming language such as Java or Python. There are key differences though. Since Solidity is designed to express financial transactions in a blockchain, the language provides an explicit way to refer to people's identities. An *address* uniquely identifies a buyer, a seller, or any account which could be either the source or target of a transaction. An address is represented by an unsigned integer, coded on 20 bytes. For example, consider the following code, which transfers a value of 200 to address `x` under a particular condition:

```
address x = 0xd26cef9bfcefa8b4b42e244353f5ec366d21e7ba;
address myAddress = this;
if (x.balance < 200 && myAddress.balance > 200) x.transfer(200);
```

Many Solidity contracts expect the sender of a transaction to match some properties. For instance, the service `pay()` in the example contract given in Section I requires that the owner of the contract is the same as the author of the transaction.

During the transaction generation, our fuzzer (i) generates new addresses or (ii) reuses those previously generated by means of a hyperparameter that represents the frequency of reuses across the transaction generation. Reusing addresses has the effect of simulating multiple transactions emitted by the very same starting endpoints. As such, the transaction `pay()` will succeed if the same address is used to deploy the `SimpleStorage` in the local blockchain.

Accounts are represented as addresses. When an account is initially generated, it has a balance expressed in *ether*. As it is customary while developing and testing Ethereum applications, to avoid wasting actual money on tests, this balance may be obtained from a *faucet address*, a standard mechanism to request (free) ether. This faucet address is initialized by our fuzzer when creating a local blockchain.

B. Generating Transactions

For the transaction emulation, the EVM implementation `py-evm` was used [8]. Although many EVM implementations are available, we chose `py-evm` because it passes all standard VM tests and it is well documented. Each transaction is executed as follows:

1) A contract is randomly selected,
2) A service within the contract is randomly chosen,
3) The arguments for the services and the calling address are generated by the fuzzer, and

4) The generated transaction is sent to the blockchain for execution, along with around 10^7 wei to pay for any possible gas costs.[2]

After the transaction's execution, the obtained gas cost is stored within the blockchain instance used by our fuzzer. This gas cost is then used to analyze patterns as we explain later.

C. Symbolic Execution

By solely relying on execution, our fuzzer may have completely missed some branches of the source code. In consequence, we added to our fuzzer a symbolic analysis over the require statements. This tried to maximize the gas costs obtained during the fuzzing by avoiding early exits due to unfulfilled constraints.

The symbolic execution operated on `require` statements with numerical conditions, and tried to always make those conditions become true. For this, we parsed each `require` call in search for the needed arguments and the expression to evaluate. In case any of these arguments were state variables of the contract, we executed a transaction to obtain the latest values of these arguments. In order to avoid visibility problems, a public getter was generated for every non-public property of the contract before generating the bytecode used by the VM.

D. Contracts and Experiment Execution

We evaluated our fuzzer on a large number of contracts obtained from http://EtherScan.io. We ran our fuzzer on $19,325$ Solidity files. These files defined $94,878$ contracts, $233,369$ public services, totaling $6,075,210$ lines of code. The analyzed contracts were all written in version 0.4 of Solidity.

The experiment was designed as follows. We iterated over the Solidity files. For each file, we created a new blockchain. We uploaded to that new blockchain all the bytecodes produced by compiling the contracts of the file. Each service was executed 15 times on average. The whole process was repeated 10 times, totaling 150 executions per service, and such, for each service in the dataset.

Since the only measured metric was gas consumption, the experiment was independent from the underlying machine's specs. It was executed using versions `0.2.0a42` of `pyevm`, `3.2.0` of `py-solc`, `1.3.0` of `eth-abi`, `0.2.1` of `eth-keys` and `0.1.0a23` of `trinity`.

Overall, we let our fuzzer randomly test more than 6 million LOC of Solidity, a process that took more than three months. We collected the consumed gas of each of the 150 invocations of each service.

III. Results

Cost estimation blueprint. We designed a visual support to summarize the result of our fuzzing algorithm on an individual service. The blueprint is composed of a histogram of the gas cost, represented by the frequency of the service invocation for a given gas cost range. A vertical dashed red line indicates the assessment made by Solidity using static analysis of the gas cost of the service, and a vertical green line for the average cost.

[2]A Wei is a smallest faction of ether, the cryptocurrency coin used to express account balances. We note that *gas* consumption is ultimately expressed in Wei.

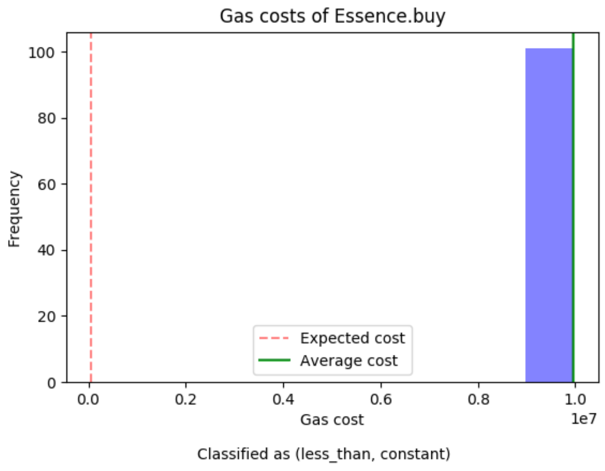

Fig. 1: Gas cost of a service in the *less-than*, *constant* category.

Figure 1 illustrates the blueprint of a service with a constant consumption during the fuzzing test. However, the expected gas cost, as predicted by Solidity, is significantly lower. This example reflects a runtime error. Having an error at runtime has dramatic consequence, in particular, all gas sent within a transaction is consumed when the transaction fails. Our simulation sends a very large amount of gas with each transaction, thus emphasizing the wrong estimation made by Solidity. The source code of the service illustrated in the figure is:

```
function buy() payable public {
  uint amount = msg.value / buyPrice;
  transferFrom(this, msg.sender, amount);
}
function transferFrom(address _from, address _to, uint256 _value)
      returns (bool) {
  var _allowance = allowed[_from][msg.sender];
  balances[_to] = balances[_to].add(_value);
  balances[_from] = balances[_from].sub(_value);
  allowed[_from][msg.sender] = _allowance.sub(_value);
  Transfer(_from, _to, _value);
  return true;
}
```

The runtime error stems from the fact that the code tries to make a transfer towards a wrong address in the simulation blockchain. The service `transferFrom` is a user-defined wrapper to transfer transaction amount that performs particular checks. Since the service `buy()` leads to an error if any of the two provided addresses are inadequate, the gas sent along with the transaction is consumed. As a result, this error causes a gas cost higher than what Solidity estimated. Solidity estimates a service cost based on summing the individual gas cost of each bytecode contained in the contract, and does not consider possible errors, such as the one illustrated above.

Cost trend. The gas consumption histogram may show an overall trend. Consider Figure 2 which shows three buckets in which the frequency decreases for higher costs. Such a trend may be due to the presence of (i) a conditional statement in which our fuzzer is not able to let the execution go through, or (ii) a service that exits early to avoid any runtime errors.

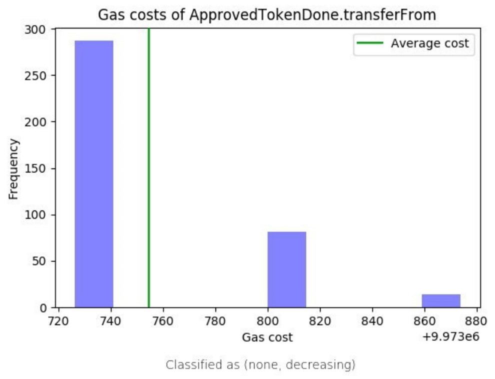

Fig. 2: Blueprint example

The blueprint given in Figure 2 does not have the Solidity estimation (*i.e.,* the dashed vertical line) because Solidity estimated the consumption as *infinite*, indicating a situation for which Solidity cannot determine an upper bound. Such a situation may happen if the contract has to invoke another contract on the blockchain, for which static analysis is not possible.

A cost trend may decrease (*e.g.,* Figure 2), be stable (*e.g.,* Figure 1), or increase, and it is a simple, albeit expressive visual support to indicate the amount of branches the code contains and how restrictive their associated conditions are.

Classification. The cost estimation blueprint relates three aspects of the fuzzing (i) the average gas cost, (ii) the gas cost estimation made by Solidity, and (iii) the general trend in the frequency of gas costs. As such, each blueprint can be classified along two dimensions, the average cost and the cost trend:

- *Average cost*: Comparing the average gas cost obtained during the fuzzing against the Solidity estimation, four situations may happen: (i) the average cost is *less* than the estimation, (ii) it is *equal*, (iii) it is *greater*, or (iv) *none*, capturing the case when Solidity cannot statically estimate the cost, and merely the service may have an *infinite* consumption.
- *Cost trend*: a cost trend belongs to one of the following four categories: *decreasing* (greater gas costs have a lower frequency than lower gas costs), *constant*, *increasing* (greater gas costs have a higher frequency than lower gas costs), or *other* (gas costs do not establish a monotonic trend).

We hypothesize that the overall profile of a gas cost consumption can be adequately characterized by these two dimensions. Furthermore, it highlights some aspects of its control flow structures. Manually inspecting the *less* category for the average costs reveals that this category is likely due to runtime errors (for which the gas fee associated with a transaction is lost when an error occurs).

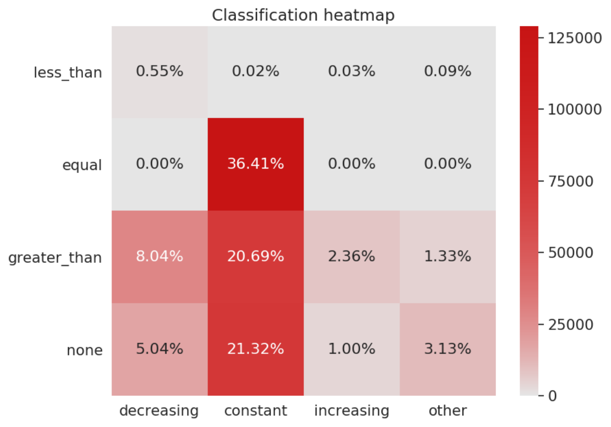

Fig. 3: Heatmap indicating the obtained classifications

Quantitative analysis. Our fuzzer produced 233, 369 blueprints (one per public service), and each was classified in the categories described above. Figure 3 shows how the blueprints are distributed across the categories.

Our results indicate that only **36.12% of the public services have a gas cost equal to the estimate made by Solidity from a static analysis**. A manual inspection on a representative sample of the source codes indicates that services with a constant cost are mostly variable accessors or calls to external services and operations which might result in an error. Note that these methods are often automatically generated by the Solidity compiler. Service accessors for fields having a dynamically-sized type, such as maps, strings, and dynamic arrays, have an infinite gas cost.

The *none* category, defined by services whose estimation was infinity, are mostly composed of services where one of the following happens: (i) a function call to an external contract is made; (ii) a transfer is made; (iii) a dynamically-sized field is returned; (iv) the code contains a loop whose length depends on the state of the contract.

The *decreasing* case of the *none* category mostly corresponds to cases (3) and (4). This is because of how the fuzzer generates dynamically sized fields, where the value generator uses a geometric distribution to determine the length of the sequences.

The *greater-than* category is composed mostly by services that have some kind of error checking with *if* statements, and simply returning if a condition is not true. These services naturally have an average cost, which is lower than the estimation, since they are called with essentially random arguments. As such, there is a very low probability that the execution would pass through the error checking statements. Because Solidity gives an upper bound for the expected gas cost, then the actual gas cost is lower than the estimation because the execution took a shorter branch of the code, mostly exiting early to avoid errors. An example of code that generates this behavior follows.

```
function transferOwnership(address _newOwner) public ownerOnly {
  if(owner != _newOwner) {
    owner = _newOwner;
  }
}
```

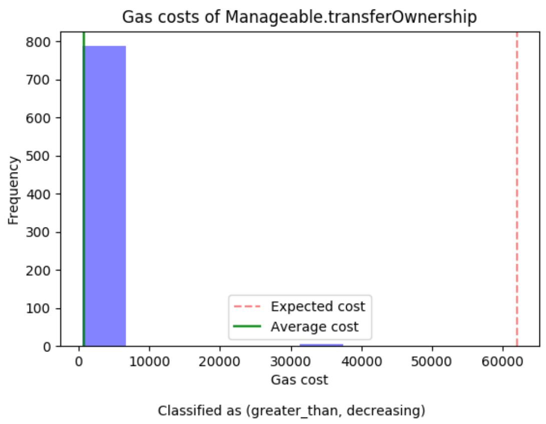

Fig. 4: Gas cost of the service transferOwnership.

The trends *decreasing*, *increasing*, and *other* are tightly knit to the *greater-than* and *none* classifications. Both *decreasing* and *increasing* generally check for errors in their code, but the trend is defined by how restrictive this check is. In the *increasing* trend, most of the executions manage to pass through the require statements. In *decreasing* trends the opposite happens. They are more densely populated because they also include code that operates on strings, arrays, or other objects of variable size. Since the fuzzer generates these values using a geometric distribution, it is more likely to have shorter values, hence the bias for these functions to be classified as decreasing. The *other* classification is also closely related to this one, since the *decreasing* trend takes strict decreasing frequencies. It takes only a couple of outliers to break the pattern and make the function be classified as *other*.

IV. Conclusion and Future Work

We have developed a fuzzing testing tool[3] that allows us to classify each service of a contract using two criteria: its performance against the estimation made by Solidity, and the trend of the gas cost exhibited during the testing. Each classification offers us some insight about what the associated code might do, including the restrictiveness of the error checking, possible optimizations and encountered runtime errors. Our testing tool was designed for ease of use and, as such, no configuration is necessary to use it.

Our fuzzer gives a pragmatic approach to gas estimations, by executing the code multiple times and checking how much each transaction cost. Since the fuzzing is done completely at random, it might violate some implicit contract expected by the service. This might explain the large amount of services that exit during error checks.

Future work will explore whether improving the fuzzer with more advanced techniques allows for faster and better estimations. Also, generating better metrics to ease comparison between the fuzzer and other tools for gas estimation is certainly interesting. In particular, we wish to compare our fuzzer to GASTAP, which use a static approach based on bytecode. Since GASTAP operates directly on the bytecode, and does not execute the code when estimating the gas cost, its runtime is mostly constant for a given contract. In contrast, our tool executes each function, and the amount of times it does so is configurable, so depending on the accuracy desired in the estimations its runtime might be higher or lower than GASTAP's.

We also make a simple recommendation to obtain safer Solidity code. Early returns with *if* statements should be chosen over runtime errors caused by *require* calls, in order to minimize lost gas.

Acknowledgments

We thank Lam Research and the ANID FONDECYT Regular 1200067 for partially sponsoring the work presented in this paper.

References

[1] G. Wood, Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07), accessed: 2018-01-03 (2017).
URL https://ethereum.github.io/yellowpaper/paper.pdf

[2] E. Albert, P. Gordillo, A. Rubio, I. Sergey, Gastap: A gas analyzer for smart contracts, ArXiv abs/1811.10403 (2018).

[3] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, C. Holler, The fuzzing book, in: The Fuzzing Book, Saarland University, 2019, retrieved 2019-09-09 16:42:54+02:00.
URL https://www.fuzzingbook.org/

[4] V. Wüstholz, M. Christakis, Harvey: A greybox fuzzer for smart contracts, CoRR abs/1905.06944 (2019). arXiv:1905.06944.
URL http://arxiv.org/abs/1905.06944

[5] Crytic, Invariant based solidity fuzzer., https://github.com/crytic/echidna.

[6] B. Jiang, Y. Liu, W. K. Chan, Contractfuzzer: Fuzzing smart contracts for vulnerability detection, CoRR abs/1807.03932 (2018). arXiv:1807.03932.
URL http://arxiv.org/abs/1807.03932

[7] F. Ma, Y. Fu, M. Ren, W. Sun, Z. Liu, Y. Jiang, J. Sun, J.-G. Sun, Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability, ArXiv abs/1910.02945 (2019).

[8] E. Foundation, A python evm implementarion., https://github.com/ethereum/py-evm.

[3]Our fuzzer is distributed under the MIT License and is available on https://github.com/danno-s/gas-fuzzer.