# Computing Exact Worst-Case Gas Consumption for Smart Contracts

Matteo Marescotti[1(✉)], Martin Blicha[1,2], Antti E. J. Hyvärinen[1], Sepideh Asadi[1], and Natasha Sharygina[1]

[1] Università della Svizzera italiana (USI), Lugano, Switzerland
{matteo.marescotti,martin.blicha,antti.hyvarinen,
sepideh.asadi,natasha.sharygina}@usi.ch
[2] Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

**Abstract.** The Ethereum platform is a public, distributed, blockchain-based database that is maintained by independent parties. A user interacts with Ethereum by writing programs and having miners execute them for a fee charged on-the-fly based on the complexity of the execution. The exact fee, measured in gas consumption, in general depends on the unknown Ethereum state, and predicting even its worst case is in principle undecidable. Uncertainty in gas consumption may result in inefficiency, loss of money, and, in extreme cases, in funds being locked for an indeterminate duration. This feasibility study presents two methods for determining the exact worst-case gas consumption of a bounded Ethereum execution using methods influenced by symbolic model checking. We give several concrete cases where gas consumption estimation is needed, and provide two approaches for determining gas consumption, one based on symbolically enumerating execution paths, and the other based on computing paths modularly based on the program structure.

## 1 Introduction

Algorithms for reaching consensus in a distributed environment have recently found applications in financial transactions based on distributed, public databases. One of the most famous applications of such systems is the Bitcoin platform. The idea is generalized to executing programs in [17] and has then been applied to other blockchains [12,13], most notably to the Ethereum platform that provides a Turing-complete execution environment where participants run programs in the form of *smart contracts* [4].

Smart contracts resemble *classes* in programming languages such as Java, C++, or Python, in that they contain data fields, called *storage*, and program code, called *functions*, which in turn have variables with a scope local to the functions. They differ from traditional programs in that, once deployed, a smart contract cannot be changed and will be publicly available. The contracts are commonly associated with monetary value, and therefore programming errors in contracts might have financial implications to the contract participants. As a result, the correct behaviour of contracts is of high interest to the participants.


T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11247, pp. 450–465, 2018.
https://doi.org/10.1007/978-3-030-03427-6_33

The execution of the Ethereum platform is carried out by *miners* that mine the transactions between a contract participant and a contract for a fee. The fee is based on the cost of the transaction as specified by the execution environment, in an abstract quantity called *gas*. A participant specifies the price he or she is willing to pay for a unit of gas, and provides an amount of money for the transaction. The miner then keeps the price of the actual gas used in the transaction from the amount as a compensation for mining the transaction, and returns the rest. In the whole Ethereum platform the daily gas costs sum up to roughly 500'000 USD at the moment of writing, and therefore even small changes in gas consumption can have a big cumulative effect.

In general the cost of a transaction depends on the unknown state of the platform, and therefore it is useful to talk about transaction's worst-case gas consumption. Ethereum provides a Turing-complete execution environment, and therefore computing the worst-case consumption is undecidable.[1] We address the central challenge of computing the exact worst-case gas consumption of a transaction through highly efficient methods adapted from symbolic bounded model checking [3] and using efficient SMT solvers [2,7,10,16]. The gas consumption of a transaction is of interest to contract participants for several reasons. In the following, we identify three cases in which computing gas consumption can help in making Ethereum more efficient.

- The Ethereum protocol imposes an upper limit for the amount of gas that a transaction may consume. As a result, if the execution cost of a function increases over time, it may happen that at a certain point a transaction of a program can no longer be carried out [1]. Computing the gas consumption helps identifying such programming errors.
- A reliable gas estimation helps a participant to place a price on the unit of gas in line with the utility of the transaction. An amount that turns out to be insufficient to carry out the transaction results in the participant losing the money without executing the transaction, while an overestimated gas consumption makes the transaction less appealing to a miner and therefore less likely to be executed.
- An approach for computing the exact worst-case gas consumption can be used as an aide to the developer for comparing semantically equivalent smart contracts with respect to their gas consumption. If the tool can show that one implementation has a lower gas consumption than others, the developer can choose to deploy the implementation with the lower gas consumption.

We define the *gas consumption paths* (GCPs) for Ethereum, and exhaustively examine all GCPs of a function using symbolic methods. The paths are identified in the high-level language Solidity, and projected to the low-level assembly code EVM currently used in Ethereum. This approach has several advantages: Due to the combination of high and low-level representations we are able to be precise on the execution paths while maintaining exactness of the gas consumption. The

[1] The protocol imposes, however, a maximum gas consumption for a block, making the computation in principle decidable.

approach is independent of the low-level representation, where gas consumed by the instructions might depend on protocol version, different compilers might produce different code, and even the assembly language is subject to change.

We suggest two algorithms for studying the GCPs. Both use techniques influenced by symbolic model checking [3] to enumerate all paths that can have different gas consumption. The first, Gas Consumption Path Enumeration, collects all constraints that affect the gas consumption, evaluates all combinations of them one-by-one, and simulates those that are satisfiable. The second, Function-Oriented Gas Consumption Path Enumeration constructs GCPs for each function as explicit *cost-equivalence classes*, which are reused through variable renaming to recursively construct more cost-equivalence classes for calling functions. We outline in addition how both algorithms can be parallelized. Both algorithms are capable of computing the exact worst-case behaviour assuming that the underlying bounded model checking formula exactly describes the contract behaviour and that the EVM and Solidity gas consumption paths have one-to-one correspondence.

*Related Work.* The `solc` compiler for the Solidity language provides a gas consumption estimate as part of the compilation. However, the estimator assumes concrete values for transaction parameters and the Ethereum state, therefore merely providing a lower bound for the worst-case gas consumption.

The tool GASPER [5] analyses Ethereum smart contracts compiled into the low-level EVM bytecode and is capable of identifying certain constructs that are costly and can be simplified to equivalent, less costly programs. While some technologies used in GASPER, such as SMT solvers and symbolic computation, are similar to ours, we identify two important differences: We propose to work on the higher-level Solidity language, and our goal is to estimate the worst-case gas consumption instead of identifying code that can be optimized.

Incorrect gas consumption values for EVM instructions enable DoS attacks on Ethereum based on frequently executing under-evaluated instructions. In [6], the authors propose an emulation-based framework to automatically adjust the gas prices of EVM instructions based on measuring their resource consumptions. As part of the emulation the approach measures the gas consumption of functions based on control and data flow, but the emulation is based on random sampling and therefore is bound to be incomplete for all but the simplest contracts. Our approach instead guarantees the completeness of the gas consumption measurement through symbolic computation and could therefore be used for improving the precision of the approach.

Correctness aspects of smart contracts other than gas consumption have been studied using symbolic methods. For instance Oyente [14] extracts the control flow graph from the EVM bytecode of a contract, and symbolically executes it in order to detect some vulnerability patterns, although it is neither sound nor complete. Zeus [11] is a framework for verification of Solidity smart contracts using abstract interpretation and symbolic model checking. The tool works by converting Solidity to LLVM bit code, and verifying reachability properties using the SeaHorn model checker [9].

## 2 Preliminaries

The *Ethereum Virtual Machine* (EVM) is a distributed-consensus-based computer running in the Ethereum blockchain [4]. EVM executes *smart contracts*, programs written in a stack-based byte-code providing a small set of low-level instructions. Smart contracts can be seen as entities that contain scoped program functions which operate on contract-wide storage that is persistent over function calls, and local variables that are only visible inside a function. We define a function in Ethereum, both in EVM byte-code and in solidity, as $f(\boldsymbol{v})$ where $f$ is the name of the function, and $\boldsymbol{v}$ is the set of function's formal parameters. When clear from the context, we omit $\boldsymbol{v}$. The storage, denoted by an array $\mathbb{S}$, is the set of storage variables that the function may access. Accesses to storage are denoted by $\mathbb{S}[i]$ where $i$ is an integer.

**Table 1.** Some EVM instruction costs [18]. The second half of the table lists examples of instructions whose cost depends on the context in which they are executed and the arguments provided.

| Instruction | Gas | Description |
|---|---|---|
| `JUMPDEST` | 1 | Indicates a valid jump destination |
| `POP` | 2 | Pop from the stack |
| `PUSH`$n$ | 3 | Push an $n$-bit item to stack |
| `ADD/SUB` | 3 | Arithmetic Operation |
| `LT/GT/SLT/SGT/EQ` | 3 | Arithmetic comparisons |
| `MLOAD/MSTORE` | 3 | Memory operations |
| `MUL/DIV/MOD` | 5 | Arithmetic Operations |
| `JUMP` | 8 | Unconditional jump to a location at the top of the stack |
| `JUMPI` | 10 | Conditional jump to a location at the top of the stack |
| `SLOAD` | 200 | Load from storage |
| `CALL` | 700 | Call a contract transaction with zero-valued arguments |
| `CALLVAL` | 9,000 | Call a contract transaction with non-zero valued arguments |
| `SSTORE` | 5,000 | Store a zero, or non-zero when previous value is non-zero |
| `SSTORE` | 20,000 | Store a non-zero when previous value is zero |
| `SSTORE` | 15,000 | Added to refund counter when storing a zero and previous value is non-zero. |

While smart contracts correspond to concepts such as instances of Java classes, they differ in an interesting way in some respects. For instance, once

deployed in Ethereum, smart contracts become publicly visible and the contract code cannot be changed. Anybody can interact with EVM through *transactions*, i.e., creating smart contracts or calling their functions, by paying a *miner* that will carry out the transaction.

The complexity of a transaction is measured in its *gas consumption*. Each EVM instruction has an associated gas consumption, a measure that relates the instruction to its storage or execution cost. See Table 1 for examples of some costs. In addition to instruction-specific costs, certain instructions and declarations affect the size of the memory local to a function, called the *active memory* [18]. Let $a$ and $b$ be the sizes of the active memory in bytes, respectively, before and after executing an instruction. The possible change incurs a cost or a refund defined as

$$\Delta C_{mem}(a, b) = 3 \cdot (a - b) + \left\lfloor \frac{a^2}{512} \right\rfloor - \left\lfloor \frac{b^2}{512} \right\rfloor .$$

To execute a transaction through a miner, a user provides a price he or she is willing to pay for a unit of gas in a currency called Ether, and the total amount of Ether that the transaction may consume. Assuming no errors are encountered while running the transaction and the amount paid for the actual gas consumption is sufficient, the transaction is carried out successfully. If carrying out the transaction requires more gas than what is provided, the execution is terminated without a refund.

Due to the memory model of EVM, in some cases the cost of an instruction depends on arguments of the instruction or the state of the contract when executing the instruction. For example:

- The instruction `SSTORE` writes into contract storage. The operation is costly in particular if a non-zero value is written to a storage location that previously contained a zero value. The EVM execution model contains a *refund counter* which is used for rewarding the user for executing instructions that make EVM less expensive. This is reflected in the case where `SSTORE` instruction writes a zero value to a location that previously held a non-zero value, resulting in a refund.
- The instruction cost of the instruction pair `CALL` and `CALLVAL` depend on their arguments. The instructions are used to call a transaction in another contract. While technically two different instructions, they can be interpreted as a single instruction from the perspective of a higher-level language. In this case the cost of a transaction depends on whether the values of the arguments passed in the call are zero.

The cost of a complete transaction in EVM is in part defined by the flow of control dictated by the EVM state, arguments, and the function code. Due to argument and environment dependence of instruction costs, the control flow graph is not sufficient for determining the transaction cost. We generalize the control flow graph to a *gas consumption graph* by adding new edges and nodes based on the instruction argument and environment dependence in a natural

way, and call paths in the gas consumption graph *gas consumption paths* (GCP). All executions of a function that follow the same gas consumption path consume therefore equal amount of gas. Our approach aims at identifying a GCP that maximizes the gas consumption over all GCPs. Instead of working directly on EVM bytecode, we base the analysis on the higher-level Solidity language, arguably the most popular language for writing smart contracts at the time. Therefore we generalize the concept of GCPs to Solidity GCPs. These are not in general the same for instance due to low-level optimizations available for EVM. As a result we do not attempt to compute the gas consumption on the Solidity code, but instead compute exact EVM gas consumption using concrete executions that are guaranteed to cover all Solidity GCPs.

We assume that the Solidity GCPs cover also all EVM GCPs. We want to emphasise this methodological choice as a potential threat to the validity of the results, and will reflect it in the theorems on correctness in the next sections.

To identify potentially different GCPs we employ bounded-model-checking techniques [3] together with SMT solvers [2,7,10,16], by operating on the static single assignment (SSA) level of Solidity where loops have been unwound up to a given limit. The approach can be made complete by increasing the unwinding limit since the Ethereum protocol imposes a maximum gas consumption for a transaction.

## 3 Gas Consumption Path Enumeration

We present an algorithm for enumerating symbolically Solidity GCPs based on the unwound SSA representation of smart contracts. While the number of GCPs is in general exponential in the size of the unwound SSA representation, due to the symbolic representation the algorithm runs in polynomial space.

We first give the translation of a Solidity contract to an unwound SSA (USSA) form in Fig. 1 for an example program adapted from [8]. For brevity, Fig. 1(a) uses a pseudo-code resembling the Solidity language instead of the actual Solidity language.[2] The contract consists of functions `f` and `g`, where `g` calls `f`. Function `g` writes to the storage variable $z$ and uses the solidity `msg.sender.transfer` function here abstracted simply as `transfer`$(z)$. Function `f` does operations on its arguments inside a loop, stores the result into a local variable, and returns the result after the computation.

The search for GCPs is done on the USSA form, given in Fig. 1(b). The form consists of a sequence of *guarded assignments* having the form $c \rightarrow b = e(x)$ or $c \rightarrow b =^s e(x)$, where $c$ is a conjunction of Boolean-valued expressions, and $e(x)$ is an operation over variables $x$. We distinguish between assignments where the left side of the equality is a variable in memory ($=$) and a storage location ($=^s$) since depending on the values these have different costs (see Table 1). Similarly the costs of some instructions depend on their arguments. For this purpose we define the function ArgCond that maps an instruction to its cost condition. For

[2] For a compilable Solidity contract see Fig. 2.

```
int z;
func g(x, y):
  if (x >= y)
    if (y >= 0)
      z = x + y
    transfer(z)
  z = f(x, y)

func f(a, b):
  int i = 0
  while (i < a + b):
   if (i < a):
     i = i + a
   else:
     i = i + b
  return i
```

(a) Pseudo-solidity

$$x_1 \geq y_1 \wedge y_1 \geq 0 \rightarrow z_1 =^s x_1 + y_1; \quad (1)$$
$$x_1 \geq y_1 \rightarrow \texttt{transfer}(z_1); \quad (2)$$
$$true \rightarrow \mathtt{f}_{a_1} = x_2; \quad (3)$$
$$true \rightarrow \mathtt{f}_{b_1} = y_2; \quad (4)$$
$$true \rightarrow \mathtt{f}_{i_1} = 0; \quad (5)$$
$$(\mathtt{f}_{i_1} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \wedge (\mathtt{f}_{i_1} < \mathtt{f}_{a_1}) \rightarrow \mathtt{f}_{i_2} = \mathtt{f}_{i_1} + \mathtt{f}_{a_1}; \quad (6)$$
$$(\mathtt{f}_{i_1} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \wedge (\mathtt{f}_{i_1} \geq \mathtt{f}_{a_1}) \rightarrow \mathtt{f}_{i_3} = \mathtt{f}_{i_1} + \mathtt{f}_{b_1}; \quad (7)$$
$$(\mathtt{f}_{i_1} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \rightarrow \mathtt{f}_{i_4} = \mathtt{ite}((\mathtt{f}_{i_1} < \mathtt{f}_{a_1}), \mathtt{f}_{i_2}, \mathtt{f}_{i_3}); \quad (8)$$
$$(\mathtt{f}_{i_1} \geq \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \rightarrow \mathtt{f}_{i_5} = \mathtt{f}_{i_1}; \quad (9)$$
$$true \rightarrow \mathtt{f}_{i_6} = \mathtt{ite}((\mathtt{f}_{i_1} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}), \mathtt{f}_{i_4}, \mathtt{f}_{i_5}); \quad (10)$$
$$(\mathtt{f}_{i_6} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \wedge (\mathtt{f}_{i_6} < \mathtt{f}_{a_1}) \rightarrow \mathtt{f}_{i_7} = \mathtt{f}_{i_6} + \mathtt{f}_{a_1}; \quad (11)$$
$$(\mathtt{f}_{i_6} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \wedge (\mathtt{f}_{i_6} \geq \mathtt{f}_{a_1}) \rightarrow \mathtt{f}_{i_8} = \mathtt{f}_{i_6} + \mathtt{f}_{b_1}; \quad (12)$$
$$(\mathtt{f}_{i_6} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \rightarrow \mathtt{f}_{i_9} = \mathtt{ite}((\mathtt{f}_{i_6} < \mathtt{f}_{a_1}), \mathtt{f}_{i_7}, \mathtt{f}_{i_8}); \quad (13)$$
$$(\mathtt{f}_{i_6} \geq \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \rightarrow \mathtt{f}_{i_{10}} = \mathtt{f}_{i_6}; \quad (14)$$
$$true \rightarrow \mathtt{f}_{i_{11}} = \mathtt{ite}((\mathtt{f}_{i_6} \leq \mathtt{f}_{a_1} + \mathtt{f}_{b_1}), \mathtt{f}_{i_9}, \mathtt{f}_{i_{10}}); \quad (15)$$
$$true \rightarrow \mathtt{f}_{ret_1} = \mathtt{f}_{i_{11}}; \quad (16)$$
$$true \rightarrow z_2 =^s \mathtt{f}_{ret_1}; \quad (17)$$

(b) USSA approximation (bound = 2)

**Fig. 1.** Converting a contract into a USSA

instance, $\text{ArgCond}(a+b) = \emptyset$, and $\text{ArgCond}(\texttt{transfer}(x)) = \{x = 0\}$. The cost implied by $\Delta C_{mem}$ only depends on the control flow path and therefore requires no special treatment.

The pseudo-code of the enumeration-based algorithm is given in Algorithm 1. The algorithm takes as input an entry point function $f(\boldsymbol{v})$ and constructs the USSA starting from $f$, in-lining recursively all functions called from $f$ (line 1). The USSA is then traversed to construct a set of Boolean expressions $C$ by adding each conjunct from each guard $c$ of the USSA assignment in lines 4–9. Additional Boolean expressions are added to $C$ for each storage assignment $=^s$ (line 7), and for each instruction whose cost depends on its arguments (line 9). The function $pre(x_i) = x_{i-1}$ maps a USSA variable $x_i$ to its previous instantiation. In case $x_i$ is the first instantiation (i.e., $i = 1$), $pre(x_i)$ is a "fresh" variable not appearing in the USSA.

In the second phase the algorithm exhaustively queries the SMT encoding of the USSA form for each Boolean combination of expressions from $C$ and obtains values for $\boldsymbol{v}$ and $\mathbb{S}$ that cover these cases in case of satisfiability. The cost of each value combination for $\boldsymbol{v}$ and $\mathbb{S}$ is then queried by simulating the transaction, and the highest gas estimate is returned as the exact worst-case bound.

**Input** : Entry function $f$; unwind limit $n$
**Output**: A set of Boolean expressions $C$
Let $U$ = the USSA form starting from $f$ unwound up to $n$
Let $C = \emptyset$
**foreach** *guarded assignment* $a \in U$ **do**
  Let $c_1 \wedge \ldots \wedge c_k$ be the guard of $a$
  $C = C \cup \bigcup_{i=1}^{k} \{c_i\}$
  **if** $a$ *is of form* $c_1 \wedge \ldots \wedge c_k \rightarrow y =^s e(x)$ **then**
    $C = C \cup \{(e(x) = 0) \wedge (pre(y) = 0), (e(x) \neq 0) \wedge (pre(y) = 0)\}$
  **end**
  $C = C \cup \text{ArgCond}(e(x))$
**end**
**foreach** *truth value combination for the elements of* $C$ **do**
  **if** $C \wedge U$ *is satisfiable* **then**
    Measure the gas consumption of $f$ on environment corresponding to the satisfying truth assignment
    Update the maximum if necessary
  **end**
**end**

**Algorithm 1.** Enumeration-based algorithm to compute GCPs of a function $f$.

*Example 1.* Running Algorithm 1 on the USSA form on Fig. 1(b) gives

$$C = \{x_1 \geq y_1, y_1 \geq 0, (x_1 + y_1 = 0) \wedge (z_0 = 0), (x_1 + y_1 = 0) \wedge (z_0 \neq 0), z_1 = 0,\\ \mathtt{f}_{i_1} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}, \mathtt{f}_{i_1} < \mathtt{f}_{a_1}, \mathtt{f}_{i_6} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}, \mathtt{f}_{i_6} < \mathtt{f}_{a_1},\\ (\mathtt{f}_{ret_1} = 0) \wedge z_1 = 0), (\mathtt{f}_{ret_1} = 0) \wedge z_1 \neq 0)\},$$

where the first two constraints $x_1 \geq y_1$ and $y_1 \geq 0$ and the whole of the second row constraining the local variables of the functions $f_{i_j}, f_{a_j}, f_{b_j}$ come from the if-conditions; the conjunctive constraints $(x_1 + y_1 = 0) \wedge (z_0 = 0), (x_1 + y_1 = 0) \wedge (z_0 \neq 0)$ come from the argument and environment dependency of `SSTORE` (see Table 1), and the constraint $z_1 = 0$ comes from the argument dependency of `CALL` and `CALLVAL`, that is, ArgCond(`transfer`$(z_1)$); and the third row comes similarly from the argument and environment dependency of `SSTORE`.

The constraint set $C$ is then provided to an SMT solver together with an SMT representation of the USSA form. Each combination of truth values for the constraints in $C$ is queried from the USSA form, resulting in the worst case $2^{11} = 2048$ SMT queries. Note that due to the incremental implementation of SMT solvers in practice the number of queries might be (exponentially) smaller, depending on the order of the queries. In certain scenarios also the input $\boldsymbol{v}$ of the function might be known, reducing the number of queries to a fraction of the worst case.

From the results of the satisfiable queries the algorithm will extract concrete values for $\boldsymbol{v}$ and $\mathbb{S}$, which are then used for computing exact gas consumptions for the corresponding gas consumption paths.

The USSA form presented in Fig. 1 does not acknowledge the invariant $z \geq 0$, and is therefore more permissive than the original contract. Obtaining such

*contract invariants* is non-trivial and out of the scope of this paper. To obtain exact worst-case gas consumption, contract invariants need to be conjoined to the USSA.

By construction of Algorithm 1 and the definition of GCPs, we immediately have the following theorem:

**Theorem 1.** *Given a function $f$, assuming a USSA for $f$ that exactly describes the contract behaviour, and that there is a one-to-one mapping between the Solidity and the EVM code, Algorithm 1 return the worst-case gas consumption of $f$.*

## 4 Function-Oriented GCP Enumeration

In this section we present an algorithm for *Function-Oriented GCP Enumeration* (FGCP), an approach to computing GCPs that prunes locally the immediately unsatisfiable gas consumption paths. The basic GCP Enumeration presented in Sect. 3 in-lines every function call and computes GCPs from the encoding of the whole program. The function-oriented approach computes the paths gradually, starting from the low-level instructions and refining the set of GCPs discovered so far in a recursive manner. We expect local pruning of GCPs to be particularly efficient for contracts that call a given function multiple times, since the approach is able to reuse previously computed, function-specific GCPs.

To present the function-oriented approach, we change slightly the notation used in Sect. 3. We introduce *cost equivalence classes* that extend the notion of cost condition from a single instruction to a block of instructions and user-defined functions. The cost equivalence classes capture the conditions under with a function behaves differently with respect to gas consumption. They correspond exactly to the GCPs of the function. We use the term function to refer to both low level instructions, such as arithmetic operations, and user-defined functions, since cost-equivalence classes do not distinguish between the two. We do not distinguish between $=$ and $=^s$, but instead introduce a separate function SSTORE that is used for updating the storage $\mathbb{S}$. Finally, we introduce a separate function-oriented version of the static single assignment form, called FSSA, that is based on guarded function calls instead of guarded assignments.

**Definition 1 (Environment).** *Given a function $f(\boldsymbol{v})$ and storage $\mathbb{S}$, the environment of an execution of $f$ is an evaluation $\upsilon$ for $\boldsymbol{v}$ and $\sigma$ for $\mathbb{S}$.*

Given a function $f$ and its environment, the execution of $f$ is deterministic and results in a new storage state.

**Definition 2 (Cost-equivalence class).** *Given a function $f(\boldsymbol{v})$ and storage $\mathbb{S}$, a cost-equivalence class is a formula representing environments $\varphi(\mathbb{S}, \boldsymbol{v})$, such that the cost of executing $f$ on any environment satisfying $\varphi$ is the same.*

Algorithm 2 computes a set of cost-equivalence classes for the input function $f(\boldsymbol{v})$. Note that the set of classes computed by the algorithm is not guaranteed to be the minimal, namely there may be different classes representing executions with equal costs.

We define with $\mathbb{C}$ the map from function to a set of its cost-equivalence classes, such that every environment satisfies exactly one formula. Thus, given a function $f(\boldsymbol{v})$, the cost equivalence classes of $f$ is the finite set

$$\mathbb{C}[f(\boldsymbol{v})] = \{\varphi_1(\mathbb{S}, \boldsymbol{v}), \ldots, \varphi_n(\mathbb{S}, \boldsymbol{v})\}$$

such that $\bigvee_{i=1}^{n} \varphi_i$ is a tautology and for all $i \neq j, \varphi_i \wedge \varphi_j$ is unsatisfiable.

Initially, all the basic functions are defined in $\mathbb{C}$ having their classes inserted manually following their cost specification. For instance, in Ethereum storing a value in the storage is performed by the operation SSTORE, which cost depends on both the value and the storage location [18]. In particular, setting a storage location from zero to a non-zero value costs more than all the other cases. Thus, according to the EVM gas consumption specifications, $\mathbb{C}[\text{SSTORE}(l, v)] = \{(\mathbb{S}[l] = 0 \wedge v \neq 0), (\mathbb{S}[l] \neq 0 \vee v = 0)\}$.

Algorithm 2 assumes that $\mathbb{C}$ contains all the functions in the input function's call tree. Such functions are both basic functions and user defined functions for each of which a previous execution of the algorithm created its classes. We assume there is no recursion.

**Definition 3 (FSSA: Function-oriented SSA).** *Given a function $f(\boldsymbol{v})$ and its USSA representation, the FSSA representation is a list of guarded function calls, one for each function call in $f$ and having the form $c \rightarrow g(\boldsymbol{l} \mapsto \boldsymbol{v_g})$ where $\boldsymbol{l} \supseteq \boldsymbol{v}$ are the local USSA variables representing the inlining of the call mapped*

```
Input  : A FSSA f(v), the cost-equivalence classes ℂ.
Assume: Every function in f is in ℂ.
Initially: ℂ[f] ← {⊤}.
Let Tr_f(𝕊, v) the USSA of f, having local SSA variables l.
foreach c → g(l ↦ v_g) in f do
    with Tr_f compute
        π(𝕊, v) := path constraint of the call g(v_g).
        M(𝕊, v, v_g) := the mapping from v to v_g of the call g(v_g).
    end
    Let s = ∅
    foreach φ(𝕊, v) in ℂ[f] do
        if ¬π ∧ φ is SAT then s ← s ∪ {¬π ∧ φ};
        foreach ψ(𝕊, v_g) in ℂ[g] do
            Let φ'(𝕊, v) = π ∧ φ ∧ M ∧ ψ
            if φ' is SAT then
                s ← s ∪ {φ'}
            end
        end
    end
    ℂ[f] ← s
end
```

**Algorithm 2.** The FGCP algorithm to compute the set $\mathbb{C}[f]$ of cost equivalence classes of $f$.

*to the arguments $\boldsymbol{v_g}$ needed for executing $g$, and $c \in \boldsymbol{l}$ is the USSA guard of the call.*

The FSSA provides the necessary information for building the *call specific* mapping $M$ on line 5 of Algorithm 2. In particular, $M$ maps the current call site to the previously computed cost-equivalence classes of the callee. Therefore $M$ enables building the cost-equivalence classes of a callee function $g$ (from Definition 2 defined over its variables $\boldsymbol{v_g}$), in terms of $\boldsymbol{v}$. A new cost-equivalence class in terms of the caller variables is built by conjoining $M$ and $\psi$ in line 11, resulting in a formula defined over $\mathbb{S}$ and $\boldsymbol{v}$. Such operation is always possible because the USSA provides a formula for computing USSA local variables $\boldsymbol{l}$ in terms of $\boldsymbol{v}$. Then, a simple rewriting following each FSSA call $\boldsymbol{l} \mapsto \boldsymbol{v_g}$ will therefore build the new class in terms of $\boldsymbol{v}$. An example of FSSA is given in Fig. 2.

**Theorem 2.** *Given a function $f$, assuming that the USSA formula $Tr_f$ used in Algorithm 2 exactly describes the contract behaviours and that the EVM and Solidity gas consumption paths have one-to-one correspondence, Algorithm 3 returns the maximum gas consumption of $f$.*

Theorem 2 ensures that the size of each classes set in $\mathbb{C}$ is finite, and that every possible behaviour is considered. This proves termination and completeness of the algorithm.

*Proof Sketch.* The property that every environment satisfies exactly one class in $\mathbb{C}$ is an invariant during the execution of Algorithm 2. The property is maintained inductively. In line 11 the algorithm creates the new classes $\varphi'$ for $f$ from the classes $\psi$ of the callee $g$. Each $\varphi'$ is mutually exclusive provided that all $\psi$ in $\mathbb{C}[g]$ are mutually exclusive, because every $\psi$ appears in the conjunction. Furthermore, the disjunction of $s$ is a tautology meaning it is complete, if the disjunction of all $\psi$ in $\mathbb{C}[g]$ is also complete. The models excluded by $\pi$ being in the conjunction in line 11, are considered by the class $\neg\pi$ added to $s$ in line 9. □

**Input** : A function $f(\boldsymbol{v})$, the cost-equivalence classes $\mathbb{C}$.
**Output** : The maximum cost $c$.
1 Let $c = 0$
2 **foreach** $\varphi(\mathbb{S}, \boldsymbol{v})$ *in* $\mathbb{C}[f]$ **do**
3 | Let $\sigma(\mathbb{S}), \upsilon(\boldsymbol{v}) =$ an environment in $\varphi$
4 | Let $c'$ the cost of executing $f(\upsilon)$ with storage $\sigma$
5 | **if** $c' > c$ **then** $c \leftarrow c'$;
6 **end**
7 **return** $c$

**Algorithm 3.** The algorithm to compute the maximum gas consumption.

Algorithm 3 computes the costs of every cost-equivalence class and returns the maximum. Definition 2 ensures that every environment satisfied by the same

equivalence class has the same cost. Thus, on line 3 the SMT solver is queried for a model of each class $\varphi$, which is guaranteed to be satisfiable by line 13 of Algorithm 2. We split the environment in two parts: $\sigma$ assigning storage locations' values, and $\upsilon$ assigning values to the input argument $\boldsymbol{v}$. Then on line 4 the function $f$ is executed on the specific environment and the cost of such execution is returned. If the cost is higher than the current maximum, on line 5 the current maximum is updated to the new value.

### 4.1 Parallelization Opportunities

Often the complexity and intrinsic sequentiality of model checking algorithms prevent parallelization. This results in missing the opportunity to exploit the modern hardware infrastructures, increasingly directed toward higher degrees of parallelism. Algorithms 1, 2 and 3 are immediately suitable for parallelization.

Due to the worst-case exponential number of SMT queries that Algorithm 1 needs to perform we believe that the part most profiting from parallelization is the evaluation of truth assignments and simulating the execution on the block starting at line 11. Since the USSA form $U$ remains the same over the queries, the parallelization may be enhanced with a clause-sharing scheme similar to [15].

Algorithm 2 can be parallelized by asynchronously executing the building of all formulas and SMT queries inside the **foreach** at line 8. Each independent process can safely execute line 13 because inserting a new formula in the set $s$ affects neither the future nor running executions. Executing line 16 and proceeding to the next function call can be done as soon as all independent executions are terminated. Algorithm 3 can be easily parallelized with using the MapReduce paradigm by defining proper *map* and *reduce* procedures. In this particular case the procedure *map* maps classes to their costs, while reduce compares the costs in order to compute the maximum.

## 5 Example

In this section we provide the example contract `C`, and we simulate the execution of Algorithms 2 and 1 on `C`.

### 5.1 Function-Oriented GCP Enumeration

The contract in Fig. 2 uses two basic functions, namely `ADD` and `STORE`. Following the Ethereum gas specification we define

$$\begin{aligned}\mathbb{C}[\texttt{ADD}(x, y, r)] &= \{\top\}, \text{and}\\ \mathbb{C}[\texttt{SSTORE}(l, v)] &= \{(\mathbb{S}[l] = 0 \wedge v \neq 0), (\mathbb{S}[l] \neq 0 \vee v = 0)\}.\end{aligned}$$

The execution of Algorithm 2 on $\mathtt{g}(u)$ and $\mathbb{C}$ will result in the classes

$$\mathbb{C}[\mathtt{g}(u)] = \{M_7 \wedge (\mathbb{S}[l] = 0 \wedge v \neq 0), M_7 \wedge (\mathbb{S}[l] \neq 0 \vee v = 0)\}.$$

```
1  contract C {
2    int a;
3    function f(bool c, int z)
4    {
5      if (c)
6      {
7        g(z);
8        z = z + 1;
9        g(z);
10     }
11   }
12
13   function g(int u)
14   {
15     a = u;
16   }
17 }
```

```
1  f(bool c, int z):
2    c → g(z ↦ u)
3    c → ADD(z ↦ x, 1 ↦ y, z1 ↦ r)
4    c → g(z1 ↦ u)
5
6  g(int u):
7    ⊤ → SSTORE(id(a) ↦ l, u ↦ v)
8
9  ADD(int x, int y, int r):
10   ⊤ → r = x + y
11
12 SSTORE(int l, int v):
13   ⊤ → 𝕊[l] = v
```

**Fig. 2.** Left: an example contract with two functions. Right: the encoding to FSSA. Lines not representing an implication are only intended to show which function the following implications refer to. The macro `id()` returns the storage id of the variable.

where $M_7(\mathbb{S}, u, l, v) := (l = \texttt{id}(a) \wedge v = u)$. Note that $M_7$ describes the mapping of the specific function call in line 7 of the FSSA in Fig. 2, which is the only function call in `g`, having path constraint $\pi := \top$. The transition relation $Tr_{\texttt{g}}$ of `g` is

$$Tr_{\texttt{g}}(\mathbb{S}, u) := \mathbb{S}[\texttt{id}(a)] = u.$$

After simplifying, the classes of `g` are

$$\mathbb{C}[\texttt{g}(u)] = \{(\mathbb{S}[\texttt{id}(a)] = 0 \wedge u \neq 0), (\mathbb{S}[\texttt{id}(a)] \neq 0 \vee u = 0)\}.$$

We now consider an execution of Algorithm 2 on $\texttt{f}(c, z)$, a function with 3 FSSA guarded calls at lines 2, 3 and 4 of Fig. 2 right, all having $\pi := c$. The USSA transition relation of `f` is

$$Tr_{\texttt{f}}(\mathbb{S}, c, z) := c \rightarrow (\mathbb{S}[\texttt{id}(a)]_1 = z \wedge z_1 = z + 1 \wedge \mathbb{S}[\texttt{id}(a)]_2 = z_1),$$

and the mappings for each function call in `f` are

$$\begin{aligned} M_2(\mathbb{S}, c, z, u) &:= (u = z), \\ M_3(\mathbb{S}, c, z, x, y, r) &:= (x = z \wedge y = 1 \wedge r = z + 1), \text{and} \\ M_4(\mathbb{S}, c, z, u) &:= (u = z + 1). \end{aligned}$$

The resulting classes for `f` are

$$\begin{aligned} \mathbb{C}[\texttt{f}(c, z)] = \{ & \neg c, \\ & c \wedge \mathbb{S}[\texttt{id}(a)] = 0 \wedge z \neq 0 \wedge z \neq -1, \\ & c \wedge \mathbb{S}[\texttt{id}(a)] = 0 \wedge z = 0, \\ & c \wedge \mathbb{S}[\texttt{id}(a)] = 0 \wedge z = -1, \\ & c \wedge \mathbb{S}[\texttt{id}(a)] \neq 0\}. \end{aligned}$$

Algorithm 2 computes a total of 5 classes. This set is not the minimal because both classes $\mathbb{C}[\mathtt{f}]_3$ and $\mathbb{C}[\mathtt{f}]_4$ cause exactly one write from zero to non-zero, resulting in the same cost. The minimal set would then be of size 4. However, by trivially combining all the cases, the total number of combinations is 16. The proposed algorithm is therefore able to reduce the number of possible classes consistently with respect to trivial enumeration, keeping the size of $\mathbb{C}$ reasonable.

### 5.2 Symbolical GCP Enumeration

The USSA form for contract $C$ in Fig. 2 is

$$
\begin{aligned}
&c_1 \rightarrow \mathtt{g}_{u_1} = z_1;\\
&c_1 \rightarrow a_1 =^s \mathtt{g}_{u_1};\\
&c_1 \rightarrow \mathtt{g}_{u_2} = z_1 + 1;\\
&c_1 \rightarrow a_2 =^s \mathtt{g}_{u_2};\\
&a_3 = \mathtt{ite}(c_1, a_2, a_0);
\end{aligned}
$$

Running Algorithm 1 on the USSA gives the set

$$
\begin{aligned}
C = \{&c_1, (a_0 = 0) \wedge (\mathtt{g}_{u_1} = 0), (a_0 \neq 0) \wedge (\mathtt{g}_{u_1} = 0),\\
&(a_1 = 0) \wedge (\mathtt{g}_{u_2} = 0), (a_1 \neq 0) \wedge (\mathtt{g}_{u_2} = 0)\}.
\end{aligned}
$$

The size of the set is five, resulting in the worst case $2^5 = 32$ SMT queries.

## 6 Summary and Future Work

In this paper we have presented a solution to the problem of estimating the gas consumption for Ethereum smart contracts based on techniques inspired by bounded model-checking techniques.

We have defined a *gas consumption path* which extends a program path in a natural way taking into account the fact that the same operation consumes different amount of gas depending on the values of its arguments and the current environment. We have presented two different algorithms to identify gas consumption paths of a given function. For each gas consumption path we are able to obtain, using SMT solver, the state of the environment that forces the execution to take the given path. Finally, we can use the functionality provided by EVM to compute the exact gas consumption of the function under the obtained state of the environment.

The main application is in computing the worst-case gas consumption that provides useful insights for the developers and may lead to uncovering a flaw in the design of the smart contract or may provide useful information when choosing between two alternative implementations. Worst-case gas consumption is of interest also for a user who wants to call a method of a contract with certain arguments, but the state of the environment is not known.

As a next step we plan to implement the presented algorithms on top of the EVM framework APIs. The implementation will serve us to compare and evaluate our proposed approach on real-world smart contracts.

**Acknowledgement.** This work has been supported by the SNSF project 166288. The authors would like to thank Leonardo Alt for the useful discussions about Solidity language and compiler, and Michael Huth for his insights into distributed ledger systems.

## References

1. GovernMental's 1100 ETH jackpot payout is stuck because it uses too much gas (2017). https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/
2. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
4. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. White paper (2014). https://github.com/ethereum/wiki/wiki/White-Paper
5. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, pp. 442–446 (2017)
6. Chen, T., et al.: An adaptive gas cost mechanism for ethereum to defend against under-priced DoS attacks. In: Liu, J.K., Samarati, P. (eds.) ISPEC 2017. LNCS, vol. 10701, pp. 3–24. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72359-4_1
7. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
8. Fedyukovich, G., D'Iddio, A.C., Hyvärinen, A.E.J., Sharygina, N.: Symbolic detection of assertion dependencies for bounded model checking. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 186–201. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_13
9. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
10. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: an SMT solver for multi-core and cloud computing. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 547–553. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_35
11. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: NDSS (2018)
12. Lundbæk, L.N., Beutel, D.J., Huth, M., Kirk, L.: Practical proof of kernel work & distributed adaptiveness (2017). https://www.xain.io/pdf/XAIN_Yellow_Paper.pdf

13. Lundbæk, L.-N., Callia D'Iddio, A., Huth, M.: Centrally governed blockchains: optimizing security, cost, and availability. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 578–599. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_29
14. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269. ACM (2016)
15. Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: Clause sharing and partitioning for cloud-based SMT solving. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 428–443. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_27
16. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
17. Szabo, N.: Smart contracts (1994). http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html
18. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Paper **151**, 1–32 (2014)