Imperial College
London

MEng Interim Report

Department of Computing

Imperial College of Science, Technology and Medicine

Solbolt – a compiler explorer and smart contract gas usage tracker for Solidity

Author:
Richard Jun Wei Xiong

Supervisor:
Dr. William Knottenbelt

January 16, 2022

Submitted in partial fulfillment of the requirements for the MEng Computing (Artificial Intelligence and Machine Learning) of Imperial College London

Contents

1

Introduction

Whereas most technologies tend to automate workers on the periphery doing menial tasks, blockchains automate away the center. Instead of putting the taxi driver out of a job, blockchain puts Uber out of a job and lets the taxi drivers work with the customer directly.

– Vitalik Buterin [?]

1.1 Overview

In recent years, Ethereum has emerged as a promising network for building next-generation, decentralised applications (DApps). Ethereum makes use of blockchain technology to achieve consensus in a distributed, public, and immutable ledger. It supports the running of arbitrary code, called smart contracts, in its own execution environment, called the Ethereum Virtual Machine (EVM). These smart contracts are typically written in a high level programming language, most notably Solidity, and are often used to power the transactional backends of various DApps. They are also used to define the behaviour of utility tokens or various non-fungible tokens (NFTs), which are simply smart contracts that extend a standard interface. Examples would be decentralised exchanges such as Uniswap [?] and Sushiswap, decentralised borrowing and lending platforms such as Aave [?] and Compound [?], algorithmic stablecoins such as Dai [?] and TerraUSD [?], and NFT-powered gaming ecosystems such as Axie Infinity [?] and the Sandbox [?].

Part of the appeal and increasing usage of smart contracts is that it allows transactions and exchanges to take place in a trustless, non-custodial, and cryptographically secure fashion. For example, in the case of a decentralised exchange, there is no need for two untrusted parties to hand over custody of their assets to a trusted intermediary, who would then conduct the actual exchange. Smart contracts are instead able to conduct the exchange of tokens or assets in a single, atomic transaction, which will either succeed, or revert and leave both parties unaffected. This powerful ability is imparted by Ethereum's immutable and decentralised nature, as no single party can control or alter the state of the blockchain directly.

The result is an explosion in smart contract development and total value locked (TVL), from around $600 million to over $150 billion in the span of two years [**?**]. However, this has also brought on scalability issues that limit the overall usefulness of the Ethereum network. In simple terms, smart contract transactions each require a certain amount of "gas" in order to be processed, depending on its complexity. Users bid for the price of gas they are willing to pay, in ether, and miners typically choose to process the transactions with higher bids. Since only a limited number of transactions can be processed in each block, this results in an efficient market economy between miners and users.

However, when more users start utilising the network, the gas prices naturally start rising as demand grows. Lately, due to the boom in decentralised finance (DeFi) platforms and NFTs, the gas price has surged by almost 10 times its price compared to two years ago [**?**]. Transactions are costing up to $63 on average in November [**?**], and this has made the Ethereum blockchain inhibitive to use for most users [**?**]. Uniswap, a popular decentralised exchange for trading tokens on the Ethereum network, has spent more than $8.6 million per day in gas fees alone, as reported last November [**?**]. Therefore, there is a clear need not only for better scaling of the Ethereum blockchain, but also for developers creating DApps on the Ethereum blockchain to optimise their smart contract code for gas usage.

Despite the enormous cost savings that can be achieved via more gas-efficient smart contracts, most of the development on Ethereum smart contract analysis currently surrounds verification of smart contracts rather than gas optimisation [**?**]. The Solidity compiler itself does offer an estimate for gas usage [**?**], although only at a course-grained function-call level, making analysis of code within each function difficult. It also only performs a rudimentary best-effort estimate, and outputs infinity whenever the function becomes too complex. Given that Solidity is a language under active development with numerous breaking changes implemented at each major version release [**?**] [**?**], the few tools designed with gas estimation in mind are also either no longer maintained or broken, or inadequate for fine-grained, line-level gas analysis. We will discuss some of these existing tools in the background section later.

Currently, most Solidity developers simply make use of general tools such as the Remix IDE, Truffle and Ganache for smart contract development [**?**]. Such tools are not specialised for detailed gas usage profiling, and developers may potentially miss out on optimisations that may save up to thousands of dollars worth of gas [**?**], or worse, possibly render a contract call unusable due to reaching the gas limits of each block [**?**]. Gas optimisation is a step that every developer has to carefully consider when deploying an immutable contract on the blockchain, and therefore creating a tool that calculates the gas usage for each bytecode instruction and line of code could greatly enhance and simplify such analysis, not only for smart contracts on the Ethereum blockchain, but also for all EVM compatible chains such as the Binance

Smart Chain, Polygon, and the Avalanche Contract Chain [**?**].

1.2 Planned contributions

With the goal of gas optimisation in mind, this project aims to improve on current smart contract development tools by creating a smart contract compiler explorer tool, which can also display a heatmap of the gas usage of each bytecode instruction, while mapping them to the corresponding lines in the user's source code. This is inspired by Godbolt [**?**], a popular compiler explorer tool for C++ for finding optimisations in the GCC compiler. The following summarises the intended contributions this project plans to achieve.

1. **Compilation and mapping of each line of user Solidity code into their corresponding EVM bytecode and Yul intermediate representation.**

   Similar to Godbolt which is able to easily map and visualise which assembly instructions correspond to its high-level C++ code, we want to achieve a similar result with EVM assembly bytecode and its corresponding Solidity code. We also want to be able to generate the Yul intermediate representation for the Solidity code, which is a recently introduced language between Solidity and EVM assembly.

2. **Parametric static analysis of gas consumption for each EVM instruction**

   The schedule for gas consumption of EVM instructions is carefully laid out in the Ethereum Yellow Paper [**?**]. Using this, we want to conduct a static analysis of gas consumption for each line of Solidity code, instead of simply at a function call level. To do this, we would need to generate a control flow graph of EVM instructions for each function call, and then calculate the worst-case gas bounds for each instruction. We would also like to be able to infer parametric bounds for gas consumption instead of simply constant bounds, and allow users to define these external parameters to obtain a more precise estimate.

3. **Generation of heatmaps of gas usage per line of code**

   To make visualising gas usage simpler, we plan on generating a heatmap on top of the direct mapping from EVM bytecode instructions to Solidity code, which would display at a glance which lines of code are consuming the most amount of gas, and may potentially have room for optimisation.

1.3 Outline of report

In Chapter

2

Background

2.1 The Ethereum Blockchain

The Ethereum Blockchain can be described as a decentralised, transaction-based state machine [**?**]. It runs on a proof-of-work system, where, simply put, miners run a computationally difficult algorithm, called Ethash [**?**], to find a valid nonce for a given block through trial and error. Once a valid nonce is generated, it is very easy for other clients to verify, but almost impossible to tamper with, since changing even one transaction will lead to a completely different hash [**?**]. The highly decentralised nature of the network makes it incredibly costly and difficult to issue malicious blocks or reorder previous blocks.

The world state of the Ethereum blockchain is a mapping between 160-bit addresses and the corresponding account state. This world state can be altered through the execution of transactions, which represent valid state transitions. Formally:

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T)$$

where is Υ the Ethereum state transition function, T is a valid transaction, and σ_t and σ_{t+1} are neighbouring states [**?**]. The main innovation of Ethereum is that Υ allows any arbitrary computation to be executed, and σ is able to store any arbitrary state, not limited to just account balances. This leads us to the concept of smart contracts, which are arbitrary code deployed and stored on the Ethereum blockchain, that can be triggered by contract calls in the form of transactions submitted to the blockchain.

2.2 Smart contracts

The idea of smart contracts was first introduced in 1994 by Nick Szabo [**?**], where he describes it as a "computerised transaction protocol that executes the terms of a contract". Contractual agreements between parties can be directly embedded within the systems that we interact with, which are able to self-enforce the terms of the contract in a way that "minimise[s] exceptions both malicious and accidental, and minimise[s] the need for trusted intermediaries", akin to a digital vending machine [?].

The Ethereum blockchain is one of the first widely successful and adopted execution environment for smart contracts. Due to the decentralised and tamper-resistant properties of Ethereum, smart contracts deployed on Ethereum can operate as autonomous actors [?], whose behaviours are completely deterministic and verifiable. Christidis et al also likens them to stored procedures in a relational database [?]. As an example, consider the following smart contract for a simple sale of a token X:

1. Alice wants to sell, for example, 10 X tokens (in this case they could be non-fungible tokens). She writes a buy token function, that accepts 5 Y tokens for each X token sold, and then automatically transfers the correct number of X tokens to the buyer's address.
2. Bob wants to purchase 2 X tokens. He can then submit a transaction calling the buy token function, with 10 Y tokens from his own account balance. The smart contract will then execute, and Bob will automatically receive 2 X tokens once the transaction is complete. Bob does not need to worry if Alice will honour her part of the contract, because the exchange is atomic and coded within the smart contract itself.
3. Alice can then withdraw the amount of Y tokens she received from the sale, by calling a withdraw profit function. This function would also check that the caller matches Alice's preset address, so that no one else is able to steal her profit.
4. The smart contract can also keep track of the number of X tokens sold, such that sales will revert after the limit of 10 is reached. It can also automatically revert the transaction if the wrong amount of Y tokens are sent, or even refund Bob the correct amount if he sends too many Y tokens.

Smart contracts can therefore be used to describe any arbitrary contractual agreement, and be used in applications such as voting and governance [?], peer-to-peer marketplaces [?], collateralised borrowing platforms [?] [?], or even NFT ticketing [?] [?]. It is important to note that smart contracts must be completely deterministic in nature, or else each node in the decentralised network will output different resulting (but valid) states.

2.3 The Ethereum Virtual Machine

2.3.1 Overview

Smart contracts on Ethereum run in a Turing-complete execution environment called the Ethereum Virtual Machine (EVM). The EVM runs in a sandboxed environment, and has no access to the underlying filesystem or network. It is a stack machine that has three types of storage – storage, memory, and stack [?], explained in detail in section **??**.

The EVM runs compiled smart contracts in the form of EVM opcodes [**?**], which can perform the usual arithmetic, logical and stack operations such as `XOR`, `ADD`, and `PUSH`. It also contains blockchain-specific opcodes, such as `BALANCE`, which returns address balance in wei.

An EVM transaction is a message sent from one account to another, and may include any arbitrary binary data (called the *payload*) and Ether [**?**]. If the target account is a *contract account* (meaning it also stores code deployed on the blockchain), then that code is executed, and the payload is taken as input.

The EVM also supports message calls, which allows contracts to call other contracts or send Ether to non-contract accounts [**?**]. These are similar to transactions, as they each have their own source, target, payload, Ether, gas and return data.

2.3.2 Storage

Every account contains a `storageRoot`, which is a hash of the root node of a Merkle Patricia tree that stores the *storage* content of that account [**?**]. This consists of a key-value store that maps 256-bit words to 256-bit words [**?**]. This storage type is the most expensive to read, and even more costly to write to, but is the only storage type that is persisted between transactions. Therefore, developers typically try to reduce the amount of storage content used, and often simply store hashes to off-chain storage solutions such as InterPlanetary File System [**?**] or Arweave permanent storage [**?**]. A contract can only access its own storage content.

Memory is the second data area where a contract can store non-persistent data. It is byte-addressable and linear, but reads are limited to a width of 256 bits, while writes can be either 8-bit or 256-bits wide. [**?**] Memory is expanded each time an untouched 256-bit memory word is accessed, and the corresponding gas cost is paid upfront. This gas cost increases quadatically as the memory space accessed grows.

The *stack* is the last data area for storing data currently being operated on, since the EVM is stack-based and not register-based [**?**]. This has a maximum size of 1024 elements of 256-bit words, and is the cheapest of the three types to access in general. It is possible to copy one of the topmost 16 elements to the top of the stack, or swap the topmost element with one of the 16 elements below it. Other operations (such as `ADD` or `SUB`) take the top two elements as input, and push the result on the top of the stack. It is otherwise not possible to access elements deeper within the stack without first removing the top elements, or without moving elements into memory or storage first.

2.3.3 Gas consumption

Gas refers to the ”the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network” [**?**]. It is the fee that is paid in Ether in order to successfully submit and execute a transaction on the blockchain. This mechanism is introduced in order to avoid malicious actors from spamming the network, either by submitting many transactions at once in a denial of service attack, or by running accidental or intentional infinite loops in smart contract code. It also incentivises miners to participate in the network, as part of the gas fees (in the form of tips as introduced in EIP-1559 [?]) is given to the miner .

Each block also has a block limit, which specifies the maximum amount of computation that can be done within each block. This is currently set at 30 million gas units, [?]. Transactions requiring more gas than the block limit will therefore always revert, which means that all executions will either eventually halt, or hit the block limit and revert.

The schedule for how gas units are calculated is described in detail in Appendix G and H of the Ethereum Yellow Paper [?]. In summary, each transaction first will require a base fee of 21000 gas units, which ”covers the cost of an elliptic curve operation to recover the sender address from the signature as well as the disk and bandwidth space of storing the transaction” [?]. Then, depending on the opcodes of the contract being executed, additional units of gas will be used up. Most opcodes require a fixed gas unit, or a fixed gas unit per byte of data. An exception to this is the gas used for memory accesses, where it is calculated as such [?]:

$$C_{mem}(a) \equiv G_{memory} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

Here, G_{memory} is the gas unit paid for every every additional word when expanding memory. a is the number of memory words such that all accesses reference valid memory. Therefore, memory accesses are only linear up to 724B, after which it becomes quadratic and each memory expansion costs more.

Another exception would be the `SSTORE` and `SELFDESTRUCT` instructions. For these instructions, the schedule is defined as follows [?]:

Name	Value	Description
G_{sset}	20000	Paid for an SSTORE operation when the value of the storage bit is set to non-zero from zero.
G_{sreset}	2900	Paid for an SSTORE operation when the value of the storage bit's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into a *refund counter*) when the storage value is set to zero from non-zero.
$G_{\text{selfdestruct}}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
$R_{\text{selfdestruct}}$	24000	Refund given (added into a *refund counter*) for self-destructing an account.

The refund counter tracks the units of gas that is refunded to the user upon successful completion of a transaction, which means that the transaction does not revert or invoke an out-of-gas exception. Only up to half of the total gas used by the transaction can be refunded using the refund counter. This mechanism is introduced to encourage freeing up storage resources used by the Ethereum blockchain.

Therefore, we can see that the estimation of gas usage for any given function is not trivial or easily predictable, and often depends on the current state of the blockchain, as well as the input parameters given. It would also require detailed modelling of the gas and refund counter, as well as other internals of the EVM implementation.

2.4 Solidity

2.4.1 The Solidity language

Solidity is a statically-typed, object-oriented, high-level programming language for developing Ethereum smart contracts [**?**]. It is a language in active development with numerous breaking changes in each major release, all of which are documented extensively online [**?**]. We will summarise some of its notable features in this section.

Inheritance. Solidity supports inheritance, by extending other contracts. These contracts can be interfaces (*abstract contracts*), with incomplete implementations of function signatures. A prime example of this would be the ERC-20 and ERC-721 interfaces [**?**] [**?**], which are standard interfaces that Ethereum tokens and non-fungible tokens respectively are expected to follow.

Libraries. Solidity also supports the use of libraries, which can contain re-usable functions or structs that are referenced by other contracts. OpenZeppelin for example has a wide range of utility libraries for implementing enumerable sets, safe math, and so on [**?**].

Function scopes. Solidity allows developers to define the scopes of each function – `internal`, `external`, `private`, or `public` [**?**]. `internal` functions can only be called by the current contract or any contracts that extend it, and are translated into simple jumps within the EVM. The current memory is not cleared, making such function calls very efficient. `private` functions form a subset of `internal` functions, in that it is stricter by only being visible to the current contract. `external` functions can only be called via transactions or message calls, and all arguments must be copied into memory first. This means that they can only be called "internally" by the same contract via the `this` keyword, which effectively makes an external message call to itself. `public` functions are a superset of `external` functions, in that they can be called internally or externally.

Storage types. Solidity supports the explicit definition of where variables are stored – `storage`, `memory`, `calldata`, or `stack`. `calldata` is an immutable, non-persistent area for storing *function arguments* only, and is reccomended to be used whenever possible as it avoids copies or modification of data. `storage` and `memory` stores the variable in the respective data areas, as described in section **??**.

2.4.2 The Solidity compiler and EVM assembly

The Solidity compiler translates the high-level Solidity code into low-level EVM bytecode, as well as generating other metadata such as the contract Application Binary Interface (ABI), and coarse gas estimates for each function. It supports a built-in optimiser, which takes an `--optimize-runs` parameter. It may be interesting for developers to observe what differences this parameter makes within the bytecode.

To illustrate the operation of the compiler, let us examine the following example contract, as well as the (truncated) output from the Solidity compiler.

```
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function one() public pure returns (uint) {
        return 1;
    }
}
```

Listing 2.1: An example Solidity contract

```
======= test.sol:C =======
...
sub_0: assembly {
        /* "test.sol":33:123  contract C {... */
      mstore(0x40, 0x80)
      callvalue
...
    tag_1:
      pop
      jumpi(tag_2, lt(calldatasize, 0x04))
      shr(0xe0, calldataload(0x00))
...
    tag_5:
        /* "test.sol":87:91  uint */
      0x00
        /* "test.sol":111:112  1 */
      0x01
        /* "test.sol":104:112  return 1 */
      swap1
      pop
        /* "test.sol":51:120  function one() public pure returns (uint) {... */
      swap1
      jump  // out
        /* "#utility.yul":7:84   */
...
        /* "#utility.yul":7:84   */
    tag_9:
        /* "#utility.yul":44:51   */
...

    auxdata: [...]
}
```

Listing 2.2: EVM assembly from the Solidity compiler

As seen in Listing **??**, the Solidity code is translated into subroutines with multiple tags as jump destinations, similar to the GCC compiler. It also includes control flow instructions such as `JUMPI` and `JUMP`. Since the EVM is a stack machine, each `JUMPI` and `JUMP` instruction does not have a corresponding target as an argument. Instead, the jump target is specified by the value at the top of the stack, which makes control flow analysis somewhat more complicated. Valid jump addresses are also specified via the JUMPDEST instruction. Debug information is embedded into the output, which describes the code fragment that a particular segment of assembly maps to, similar to the -g flag in the GCC compiler.

A detailed explanation of how the EVM assembly maps to the Solidity instructions is included in section **??**, where we examine this in relation to the Yul intermediate representation.

An additional #utility.yul file is also generated, which contains extra Yul code (see Section **??**) that are created automatically by the compiler, and its addition is triggered by the usage of specific language features, such as the ABI encoder for function paramters and return values. They are not specific to the user's source code, but are likely to be executed in control flow jumps from other parts of user code.

2.4.3 Yul

Yul is an intermediate language that is designed to be compiled into bytecode for different backends [**?**], such as future EVM versions and EWASM, and acts as a common denominator for all current and future platforms. Yul is also designed to be human readable, with high level constructs such as for and switch, but still suitable for whole-program optimisation.

As an example, we examine the Yul intermediate representation (IR) output from the Solidity compiler, using the same Solidity code in Listing **??**:

```
/// @use-src 0:"test.sol"
object "C_10" {
    ...
    /// @use-src 0:"test.sol"
    object "C_10_deployed" {
        code {
            ...
            /// @ast-id 9
            /// @src 0:51:120  "function one() public pure returns (uint) {..."
            function fun_one_9() -> var__4 {
                /// @src 0:87:91  "uint"
                let zero_t_uint256_1 := zero_value_for_split_t_uint256()
                var__4 := zero_t_uint256_1

                /// @src 0:111:112  "1"
                let expr_6 := 0x01
                /// @src 0:104:112  "return 1"
                var__4 := convert_t_rational_1_by_1_to_t_uint256(expr_6)
                leave

            }
            /// @src 0:33:123  "contract C {..."

        }
       ...
    }
}
```

Listing 2.3: Yul IR from the Solidity compiler

We see that the Yul code far more resembles the original Solidity code. Here, we also observe how the stack is being used in our simple one() function:

1. First, a temporary local variable `zero_t_uint256_1` is pushed onto the stack with value `0x00`. Then, it is assigned to the return variable `var__4`. In EVM assembly, this maps to the `0x00` instruction on line 15 of Listing **??**.
2. Next, another temporary local variable `expr_6` is pushed onto the stack with value `0x01`. It is then assigned to the return variable `var__4` again, and the function exits, returning the value in `var__4`. In EVM assembly, this maps to the `0x01` instruction on line 17. Then, to assign the new value to `var__4`, we first swap it with the old value (currently on the 1st slot in the stack below `0x01`), on line 19. Finally, we pop the old value off the stack, and swap the next jump instruction address onto the top of the stack, in lines 20 and 22.

Similar to the EVM assembly, the generated Yul code also contains debug information about the mapping to the original source code. This can be very useful for understanding how the final EVM assembly operates, as seen in our previous example. As such, it would be interesting to generate a similar mapping for this project to the Yul intermediate representation, in addition to the final EVM assembly.

3

Gas bounds analysis of EVM instructions

Now that we understand the basics of the Ethereum blockchain as well as the generation of EVM instructions from Solidity code, let us now examine how we can derive gas estimates from it. Although many of the opcodes have a fixed gas cost, a large part of gas used still may be derived from storage and memory costs, which require more detailed analysis. Therefore, a naive implementation of simply adding up the fixed costs of each instruction will not be sufficiently precise or sound for our analysis.

To solve this, we propose making use of and extending Mythril, a symbolic execution engine for EVM bytecode, in order accurately infer gas bounds for each line of Solidity code. We will discuss the steps involved in detail within the following sections.

3.1 Construction of basic blocks

After the generation of the EVM bytecode, we must first define the notion of a basic blocks, which are the maximal straight-line sequence of consecutive instructions where there are no branches in apart from the first instruction, and no branches out apart from the last instruction [?]. For EVM instructions, this can be defined as such [?]:

Definition 3.1.1 (basic blocks). *Given an EVM program $P \equiv b_0, ..., b_p$, we have*:

$$blocks(P) \equiv \left\{ B_i \equiv b_i, ..., b_j \;\middle|\; \begin{array}{l} (\forall k.i < k < j, b_k \notin Jump \cup End \cup \{\texttt{JUMPDEST}\}) \wedge \\ (i = 0 \vee b_i \equiv \texttt{JUMPDEST} \vee b_{i-1} \equiv \texttt{JUMPI}) \wedge \\ (j = p \vee b_j \in Jump \vee b_j \in End \vee b_{j+1} \equiv \texttt{JUMPDEST}) \end{array} \right\}$$

where

$$Jump \equiv \{\texttt{JUMP}, \texttt{JUMPI}\}$$
$$End \equiv \{\texttt{REVERT}, \texttt{STOP}, \texttt{INVALID}\}$$

The generation of basic blocks can therefore be done in linear time, by parsing each EVM instruction line by line. After this, we can also then generate the set of valid jump addresses, which are simply the set of addresses with *`JUMPDEST`* instructions.

3.2 Symbolic execution

Next, in order to understand the maximum gas costs incurred by each basic block during transaction execution, we can use symbolic execution to explore as many of the possible execution branches that can be reached. This requires a detailed model of the internals of the EVM, such as how it handles the stack, and fortunately Mythril is a tool that has such capabilities. Mythril [**?**] is originally designed by ConsenSys for verification of smart contracts and analysing which path conditions are able to reach certain states to induce undesired behaviour, such as killing a smart contract. However, its `LASER` symbolic execution engine can also be used for our purposes of gas estimation. It also has the ability to output a control flow graph of the paths it took during symbolic execution.

To do this, we propose extending its gas meter module to also keep track of the total gas used by each basic block. Then, we are able to output a control flow graph with the corresponding gas usage information, and then calculate the maximum and minimum gas used by each block out of all the possible paths.

There are also certain tuneable hyperparameters that Mythril offers, such as a bound on the number of times a loop can be explored, as well as setting the states of storage variables in a smart contract and the global blockchain state. We would like to also expose these settings for the user, and allow a user to define the intial state used for symbolic execution to obtain a more accurate estimate.

There are also other tools such as Oyente [**?**] (and EthIR, an extension of it,) that are able to perform symbolic execution and produce a control flow graph. However, when testing their capabilities, it was found that they were no longer actively maintained, and contained many errors that required patching before they worked. They also did not support the latest Solidity version, and were missing numerous newly introduced opcodes, such as SHR (shift right) and SHL (shift left).

3.3 Loop bounds inference

Finally, when symbolically executing a smart contract, we will likely run into the problem of path explosion. For large and complex smart contracts, the number of possible execution paths will grow exponentially with an increase in bytecode size. However, smart contracts are typically much simpler compared to executable desktop programs, because of the gas limit imposed by each block. They are also guaranteed to halt, either by successfully completing the execution, or by running out of gas. Even so, the number of paths that may need to be traversed before hitting the block gas limit may still be intractible, and require some heuristics for optimisation. Therefore, we propose to infer loop boundaries for simple loop patterns that can be evaluated statically, by extending the Solidity compiler that traverses the Yul intermediate representation.

DERIVES$_F ROM(var) => sequence of parameters it derives from$

var := expr =¿ DERIVES$_F ROM(var) = OP(expr) mstore(addr, expr) => DERIVES_F ROM(m_a ddr$
$OP(expr)//same for sstore$

if (expr) [statements] =¿ OP(expr) ?

OP(func(expr,...)) = DERIVES$_F ROM(ret_v ar) OP(mload(addr)) = DERIVES_F ROM(m_a ddr_v ar)//s$
$OP(expr1) - OP(expr2)$

OP(var) = DERIVES$_F ROM(var) OP(const) = const$

DERIVES$_F ROM(expr_3 0) = OP(lt(cleanup_{tu} int256(expr_2 8), cleanup_{tu} int256(expr_2 9))) =$
$OP(cleanup_{tu} int256(expr_2 8)) < OP(cleanup_{tu} int256(expr_2 9)) = OP(expr_2 8) < OP(expr_2 9) =$
$OP(_4) < OP(_5) = OP(var_{i2} 5) < OP(var_l oopEndIndex_1 9) = OP(var_{i2} 5) < OP(expr_2 2) =$
$OP(var_{i2} 5) < OP(expr_2 2) = OP(var_{i2} 5) < OP(checked_s ub_{tu} int256(expr_2 0, convert_{tr} ational_{1b} y_{1t} o_{tu} i$
$OP(var_{i2} 5) < OP(expr_2 0) - OP(convert_{tr} ational_{1b} y_{1t} o_{tu} int256(expr_2 1))) = DERIVES_F ROM(var_{i}$
$OP(var_a rgument_1 5) - OP(expr_2 1)) = DERIVES_F ROM(var_{i2} 5) < OP(var_a rgument_1 5) -$
$0x01) = FOR(var_{i2} 5, OP(convert_{tr} ational_{0b} y_{1t} o_{tu} int256(expr_2 6)), OP(_2)) < var_a rgument_1 5 -$
$0x01) = FOR(var_{i2} 5, OP(expr_2 6), OP(increment_{tu} int256(_3))) < var_a rgument_1 5 -$
$0x01) = FOR(var_{i2} 5, 0x00, OP(increment_{tu} int256(_3))) < var_a rgument_1 5 - 0x01) =$
$FOR(var_{i2} 5, 0x00, OP(add(_3, 1))) < var_a rgument_1 5 - 0x01) = FOR(var_{i2} 5, 0x00, OP(_3) +$
$OP(1)))) < var_a rgument_1 5 - 0x01) = FOR(var_{i2} 5, 0x00, OP(var_{i2} 5) + 1))) < var_a rgument_1 5 -$
$0x01)$

// until hit a cycle I guess?

Then we know the number of times this will run =¿ var$_a rgument_1 5 - 0x01 - (0 +$
$n * 1) = 0 var_a rgument_1 5 - 1 - n = 0 n = var_a rgument_1 5 - 1$

4

Related works

5

Ethical considerations