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 25, 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]

1.1 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 [2] and Sushiswap, decentralised borrowing and lending platforms such as Aave [3] and Compound [4], algorithmic stablecoins such as Dai [5] and TerraUSD [6], and NFT-powered gaming ecosystems such as Axie Infinity [7] and the Sandbox [8].

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 [9]. 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 [10]. Transactions are costing up to $63 on average in November [11], and this has made the Ethereum blockchain inhibitive to use for most users [12]. 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 [13]. 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 [14]. The Solidity compiler itself does offer an estimate for gas usage [15], 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 [16] [17], 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 [18]. 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 [19], or worse, possibly render a contract call unusable due to reaching the gas limits of each block [20]. 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 [21].

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 [22], 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. **Analysis of gas consumption via symbolic execution for each EVM instruction**

   The schedule for gas consumption of EVM instructions is carefully laid out in the Ethereum Yellow Paper [23]. Using this, we want to conduct an analysis of gas consumption for each line of Solidity code, instead of simply at a function call level. To do this, we propose using symbolic execution to traverse each possible execution path, and then aggregating the gas costs for each basic block. To address the problem of path explosion, we also propose the implementation of a non-iterative loop bounds analysis that can generate a parametric loop bound using polytopes. Then, we can 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, we provide a summary of smart contracts on Ethereum, the internals of the Ethereum Virtual Machine and its gas schedule, as well as a background on Solidity and the Yul intermediate representation. In Chapter 3, we describe our proposed implementation to estimate gas costs using symbolic execution, and a non-iterative algorithm using polytopes to statically infer loop bounds. In Chapter 4, we briefly examine two related projects, as well as their results and limitations. In Chapter 5, we propose the method we intend to measure the project's quality and effectiveness by. In Chapter 6, we examine the possible ethical issues that may arise from this project. Finally, in Chapter 7, we discuss the proposed timeline for the completion of this project, including stretch goals and fallback goals under various circumstances.

2

Background

2.1 The Ethereum Blockchain

The Ethereum Blockchain can be described as a decentralised, transaction-based state machine [23]. It runs on a proof-of-work system, where, simply put, miners run a computationally difficult algorithm, called Ethash [24], 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 [25]. 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 [23]. 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 [26], 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 [27].

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 [28],

whose behaviours are completely deterministic and verifiable. Christidis et al also likens them to stored procedures in a relational database [28].

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 [29], explained in detail in section 2.3.2.

The EVM runs compiled smart contracts in the form of EVM opcodes [30], 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 [30]. 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 [30]. 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 [23]. This consists of a key-value store that maps 256-bit words to 256-bit words [30]. 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 [31] or Arweave permanent storage [32]. 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. [30] 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 [30]. 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” [33]. 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 [34]) 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, [33]. 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 [23]. 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” [35]. 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 [23]:

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

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 [23]:

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 [36]. It is a language in active development with numerous breaking changes in each major release, all of which are documented extensively online [15]. 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 [37] [38], 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 [39].

Function scopes. Solidity allows developers to define the scopes of each function – `internal`, `external`, `private`, or `public` [40]. `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 2.3.2.

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 2.5, 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 2.4.3, 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 2.4.3) 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 [41], 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 2.1:

```
1  /// @use-src 0:"test.sol"
2  object "C_10" {
3      ...
4      /// @use-src 0:"test.sol"
5      object "C_10_deployed" {
6          code {
7              ...
8              /// @ast-id 9
9              /// @src 0:51:120  "function one() public pure returns (uint) {..."
10             function fun_one_9() -> var__4 {
11                 /// @src 0:87:91  "uint"
12                 let zero_t_uint256_1 := zero_value_for_split_t_uint256()
13                 var__4 := zero_t_uint256_1
14
15                 /// @src 0:111:112  "1"
16                 let expr_6 := 0x01
17                 /// @src 0:104:112  "return 1"
18                 var__4 := convert_t_rational_1_by_1_to_t_uint256(expr_6)
19                 leave
20
21             }
22             /// @src 0:33:123  "contract C {..."
23
24         }
25         ...
26     }
27 }
```

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 2.5.

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.

Performing analysis on the Yul IR also has the advantage over Solidity code that it reveals certain *"hidden loops"* when dealing with variable sized data types like `string` and `bytes`. For example, observe the following Solidity function:

```
1  pragma solidity >=0.7.0 <0.9.0;
2  contract TestLoop {
3      string private storedString;
4      constructor() {}
5
6      function hiddenLoop(string calldata newString) external {
7          storedString = newString;
8      }
9  }
```

Listing 2.4: Where is the a loop within this contract?

Upon first examination, there seem to be no loops involved in the `hiddenLoop` function. However, as it assigns a variable data type `string`, a loop may be generated automatically for this by the Solidity compiler, but is not apparent within the Solidity code itself. Looking at the Yul code however, this "hidden" loop is more explicit:

```
1  function copy_byte_array_to_storage_from_t_string_calldata_ptr_to_t_string_storage(
       slot, src, len) {
2      let newLen := array_length_t_string_calldata_ptr(src, len)
3      // Make sure array length is sane
4      if gt(newLen, 0xffffffffffffffff) { panic_error_0x41() }
5
6      let oldLen := extract_byte_array_length(sload(slot))
7
8      // potentially truncate data
9      clean_up_bytearray_end_slots_t_string_storage(slot, oldLen, newLen)
10
11     let srcOffset := 0
12
13     switch gt(newLen, 31)
14     case 1 {
15         let loopEnd := and(newLen, not(0x1f))
16
17         let dstPtr := array_dataslot_t_string_storage(slot)
18         let i := 0
19         for { } lt(i, loopEnd) { i := add(i, 0x20) } {
20             sstore(dstPtr, calldataload(add(src, srcOffset)))
21             dstPtr := add(dstPtr, 1)
22             srcOffset := add(srcOffset, 32)
23         }
24         if lt(loopEnd, newLen) {
```

```
25              let lastValue := calldataload(add(src, srcOffset))
26              sstore(dstPtr, mask_bytes_dynamic(lastValue, and(newLen, 0x1f)))
27          }
28          sstore(slot, add(mul(newLen, 2), 1))
29      }
30      default {
31          let value := 0
32          if newLen {
33              value := calldataload(add(src, srcOffset))
34          }
35          sstore(slot, extract_used_part_and_set_length_of_short_byte_array(value,
                newLen))
36      }
37  }
38
39  function fun_hiddenLoop_17(var_newString_9_offset, var_newString_9_length) {
40
41      let _1_offset := var_newString_9_offset
42      let _1_length := var_newString_9_length
43      let expr_13_offset := _1_offset
44      let expr_13_length := _1_length
45      copy_byte_array_to_storage_from_t_string_calldata_ptr_to_t_string_storage(0x00,
            expr_13_offset, expr_13_length)
46      let _2_slot := 0x00
47      let expr_14_slot := _2_slot
48
49  }
```

Listing 2.5: Yul IR that reveals the hidden loop

Here, we see that the string is rather represented as an offset into the calldata and its length in line 39. Then, we see that our hiddenLoop function calls the ”copy byte array to storage” function in line 45. This function then has a switch statement in line 13, and if the length of the string is greater than 32 bytes, it is stored using a for loop in line 19, where the first 32 byte word stores the length of the string, and the following words store the actual contents itself.

3

Gas bounds analysis

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 [42]. For EVM instructions, this can be defined as such [43]:

Definition 3.1.1 (basic blocks)**.** *Given an EVM programme $P \equiv b_0, ..., b_p$, 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 [44] 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 [45] (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 programmes, 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 loop patterns that can be evaluated statically, by extending the Solidity compiler that traverses the abstract syntax tree of the Yul intermediate representation. To our knowledge, no other existing tool performs this kind of static analysis on Yul IR.

The method chosen is adapted from Lokuciejewski et al [46], where they describe a non-iterative approach to analyse loop boundaries of certain loops that match their constraints. These constraints are that the exit conditions must either depend on a constant, non-modified variable within the loop, or at most a single modified variable, and that all condition statements are affine expressions of variables. We can adopt a similar method for loops in the Yul language.

First, we take advantage of the fact that all computation on the EVM is deterministic. Therefore, any variable must either have been declared statically within the source code, or depends on a set of operations that acts on a finite set of parameters, such as the function arguments, or global parameters within the blockchain, such as the block number and timestamp. For our analysis, we transform any calls to external contracts into a parameter as well.

Then, we can define the following relationship for each variable V, and each time n it is newly assigned to (if n is omitted, assume we are retrieving the last assignment):

$$
\begin{aligned}
&DerivesFrom(V : \texttt{Var}, n : \texttt{Const?}) \\
&\quad : \texttt{Expr} \\
&\equiv \begin{cases}
\texttt{Unknown} \\
\texttt{Const} \\
\texttt{Var}, n' \\
Induct(\texttt{Var}) \\
BinOp(left : \texttt{Expr}, right : \texttt{Expr}) \\
If(cond : \texttt{Expr}, branchTrue : \texttt{Expr}, branchFalse : \texttt{Expr})
\end{cases}
\end{aligned}
\tag{3.1}
$$

This relationship helps us identify how a variable is calculated at each point in the programme. Next, for each expression in a Yul programme, we also have the following operator Op:

$$
\begin{aligned}
&Op(e : \texttt{Expr}) \\
&\quad : \texttt{Expr} \\
&\equiv \begin{cases}
\texttt{Const} & \text{if } e \text{ is type } \texttt{Const} \\
e & \text{if } e \text{ is type } \texttt{Var} \text{ and } DerivesFrom(e) \text{ does not exist} \\
DerivesFrom(e) & \text{if } e \text{ is type } \texttt{Var} \text{ and } DerivesFrom(e) \text{ exists} \\
DerivesFrom(retVar) & \text{if } e \text{ is } Func(Args) \text{ returns } retVar \\
BinOp(Op(left), Op(right)) & \text{if } e \text{ is } BinOp(left, right) \\
newExtVar & \text{if } e \text{ is a call to an external function}
\end{cases}
\end{aligned}
\tag{3.2}
$$

This relationship applies and simplifies each operation into the simplest $DerivesFrom$ relationship possible. Then, we walk the abstract syntax tree for the Yul IR using the pseudocode in the following subsections, which can be done at the same time as EVM code compilation.

3.3.1 Assignment statements

For assignment statements, we simply call the Op operator on the expression, in order to simplify it into a $DerivesFrom$ relationship of constants, operators and parameters, which we push to the current $DerivesFrom$ data structure.

```
procedure Walk(DerivesFrom, [s : s_tail], LoopConstraints)
  switch s
    case s matches (LET v := expr)
```

```
        DerivesFrom.push(v, Op(expr))
        Walk(DerivesFrom, Conds, s_tail, LoopConstraints)
  end procedure
```

For example, say we have the following Yul function:

```
  function fun_simpleLoop_39(var_argument_11) {
    let _1 := var_argument_11
    let expr_16 := _1
    let expr_17 := 0x01
    let expr_18 := add(expr16, expr_17)
}
```

Then, we obtain the following $DerivesFrom$ relationships:

$$\begin{aligned} DerivesFrom(\texttt{\_1}) &\equiv Op(\texttt{var\_argument\_11}) \\ &\equiv \texttt{var\_argument\_11} \end{aligned}$$

$$\begin{aligned} DerivesFrom(\texttt{expr\_16}) &\equiv Op(\texttt{\_1}) \\ &\equiv DerivesFrom(\texttt{\_1}) \\ &\equiv \texttt{var\_argument\_11} \end{aligned}$$

$$\begin{aligned} DerivesFrom(\texttt{expr\_17}) &\equiv Op(\texttt{0x01}) \\ &\equiv \texttt{0x01} \end{aligned}$$

$$\begin{aligned} DerivesFrom(\texttt{expr\_18}) &\equiv Op(add(\texttt{expr16}, \texttt{expr\_17})) \\ &\equiv add(Op(\texttt{expr16}), Op(\texttt{expr\_17})) \\ &\equiv add(DerivesFrom(\texttt{expr\_16}), DerivesFrom(\texttt{expr\_17})) \\ &\equiv add(\texttt{var\_argument\_11}, \texttt{0x01}) \end{aligned}$$

3.3.2 If statements

For if statements, we first simplify the condition using the Op operator, and then clone the current $DerivesFrom$ struct. We then use the cloned struct to walk the true branch of the if statement. In Yul, there are no else statements, so control flow returns to the original programme point before the branch.

Then, we iterate through the variables within the $DerivesFrom$ struct of the upper scope, and then check if they have been modified by the if branch. If they have, we push an If expression as the variable's $DerivedFrom$ relationship, with the simplified condition obtained before.

```
procedure Walk(DerivesFrom, [s : s_tail], LoopConstraints)
  switch s
    case s matches (IF (c) {if_stats})
      // Create new scope
      DerivesFrom_ = DerivesFrom.clone()
      c_ = Op(c)
      Walk(DerivesFrom_, if_stats)

      // For variables in upper scope
      for v in DerivesFrom do
        v_prev = DerivesFrom.pop(v)
        v_if = DerivesFrom_.get(v)
```

```
        if (v_prev != v_if)
          DerivesFrom.push(v, If(c_, v_if, v_prev))
      Walk(DerivesFrom, Conds, s_tail, LoopConstraints)
end procedure
```

3.3.3 For loop statements

Finally, for loops, we have to first check if certain constraints are met before we are able to analyse it statically. The constraints we have for loops that can be statically analysed are:

1. There is only one induction variable for each loop.
2. The induction variable can only be initialised to a constant variable, or an affine expression of another previous induction variable from a previous loop
3. The loop condition can only depend on this induction variable and other constants or variables that remain unchanged
4. The induction variable can only be modified within the post-loop header, and not anywhere else in the body
5. The induction variable must be modified by a constant or a variable that remains unchanged

To enforce these constraints, we first obtain a set of all variables that are modified within the pre- and post-loop headers, as well as the loop body. We check that any variables set in the pre-loop header are not `Unknown` (which means they were modified within a previous outer loop), and that they are either constant or an affine expression of an $Induction(\texttt{Var})$ from an outer loop.

Then, we obtain a set of all loop conditions, by first including the condition within the loop header, and then traversing the loop body to find any break statements, and then including the conditions needed to reach it. After that, we extract all the variables involved in the loop conditions.

Next, we check for the intersection of variables modified in the pre- and post-loop headers and involved in the loop conditions. There should only be one variable left, which will be our induction variable. We then check that the induction variable as well as the other variables involved in the loop conditions are not modified anywhere else within the loop body.

If the constraints are satisfied, we can analyse the loop using the induction variable and the mutations performed on it, described in a later section, to obtain a set of loop constraints. These are passed on when walking the inner loop body, so that they can be composed with any nested loops. We then create a new $DerivesFrom$ struct for the inner loop scope, and push an `Unknown` value for the $DerivesFrom$ relationship of any variables modified within the loop. For the induction variable, we push a special $Induction(\texttt{Var})$ variable to note that it is an induction variable. We now walk the inner loop bodies in the same way.

If the constraints are not satisfied, we simply exit the loop altogether, because any nested loops will also not be analysable. In these cases, we will need to fallback into using symbolic execution with a bound on loop unfolding.

Finally, when exiting the loop, we append fresh variables to any of the variables that were modified within the loop. This may be improved on in the future for variables that meet

certain constraints, where we can statically derive their final values as another $DerivesFrom$ relationship between the number of loop invocations and previously defined variables.

```
procedure Walk(DerivesFrom, [s : s_tail], LoopConstraints)
  switch s
    case s matches (FOR (pre) (c) (post) (loop))
      // Checks if loop can be analysed
      canBeAnalysed = True

      loopHeaderVars = new Set()
      // Check loop header assignments
      // These must be either constant, or
      // set to an affine expression of one induction variable
      DF_Pre = emptyDerivesFrom()
      Walk(DF_Pre, pre)
      for v in DF_Pre
        if (DF_Pre.get(v) is not Unknown and is affine)
          loopHeaderVars.insert(v)

      modifiedPostVariables = new Set()
      // Find possible induction variables in post
      DF_Post = emptyDerivesFrom()
      Walk(DF_Post, post)
      for v in DF_Post
        modifiedPostVariables.insert(v)

      loopCondsVariables = new Set()
      // Find loop conditions,
      // and get the set of variables they derive from
      v, conds = findVars(c)
      v_, conds_ = findBreakCondVars(loop)
      loopConditions = unionConds(conds, conds_)
      loopCondsVariables.insert(v ++ v_)

      inductionVars = intersect(loopHeaderVars, modifiedPostVariables,
          loopCondsVariables)

      modifiedLoopVariables = new Set()
      // Find variables modified within a loop
      modifiedLoopVariables = findModifiedLoopVars(loop)

      // check if can be analysed
      if inductionVars.length != 1 or !disjoint(loopCondsVariables,
          modifiedLoopVariables)
        canBeAnalysed = False

      // If loop cannot be analysed, then any nested
      // loops also cannot be analysed, so we skip them
      if canBeAnalysed
        inductionVar = inductionVars[0]

        // Get loop constraints
        LoopConstraints_ = calculateLoopBounds(inductionVars[0], DF_Pre,
            DF_Post, LoopConstraints)
        logConstraints(LoopConstaints_)

        // Create new DerivesFrom and walk loop body
        // Induction variable is set to special value
        // composition with nested loops
        DF_Loop = union(DF_Pre, DerivesFrom)
```

```
        DF_Loop.push(inductionVars[0], getNewInductionVar(inductionVars[0])
            )
        for v in modifiedLoopVariables
          DF_Loop.push(v, Unknown)

        Walk(DF_Loop, loop, LoopConstaints_)

      // Push new fresh variables for each variable modified
      // in loop, continue with rest of control flow
      for v in modifiedLoopVariables
        DerivesFrom.push(v, getNewVar())

      Walk(DerivesFrom, Conds, s_tail, LoopConstraints)
end procedure
```

Although these constraints seem rather strict, we observe that they are often satisfied by many well-formed Yul programmes. This is partly because of the gas constraints when developing in Solidity, and therefore complex loops are not often used by developers.

3.3.4 Loop constraint polytopes

Next, in order to infer loop bounds as a parametric equation, the `calculateLoopBounds` will output a number of constraints in the form of a *polyhedron*, which is an N-dimensional geometrical object defined by a set of linear inequalities [46]:

Definition 3.3.1 (polyhedrons). *An N-dimensional polyhedron P is defined by*:

$$P := \{x \in \mathbb{Z}^N | Ax = a, Bx \geq b\}$$

where $A, B \in\in \mathbb{Z}^{m \times N}$ and $a, b \in \mathbb{Z}^m$ and $m \in \mathbb{N}$.

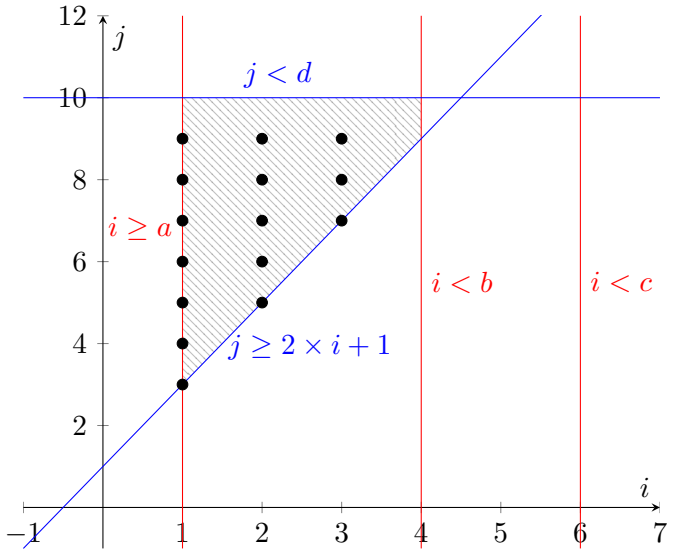
In our case, the constraints are also *polytopes*, since we have $|P| < \infty$. To obtain this, we construct the first inequality with the initial value the induction variable is set to, and check to see if it is incremented or decremented in each loop body. Assuming it is set to a initially and incremented each time, we have for induction variable i that $i \geq a$.

Then, for each of the loop conditions acting on i, we normalise them and derive another set of inequalities. For example, if the loop continues on the condition that $i < b$, and breaks when $i == c$, we then have $i < b$, $i < c$ and $i > c$ as additional inequalities. We can then find the smallest intersection between all of these inequalities, which is parameterised by a, b and c, and the loop bounds can be derived from there together with the amount that the induction variable is incremented or decremented by. These parameters can then be defined by the user arbitrarily, and the loop bounds can be recalculated directly without needing to re-compute the loop analysis.

In the case of nested loops, the previous loop constraits from the outer loop can be included together with the constraints of the current loop. For example, say the loop described previously had an inner loop with the inequalities $j > 2 \times i + 1$, $j < d$. Then, we can perform a similar intersection, this time in 2 dimensions instead of just 1, as visualised in Figure 3.1. Each loop execution is also marked with a dot within the shaded intersection area.

This information can then be output as debug metadata in the final EVM assembly. The final step involves making use of the mapping from Yul IR to the generated EVM assembly. We can

Figure 3.1: Polytope calculated for a nested loop



modify the symbolic execution engine to read the debug metadata for each tag generated from the Yul IR.

Then, instead of relying on symbolic execution, if the user specified concrete values for the paramters involved, we can calculate the loop bounds directly, and estimate the gas cost as the cost for running the block once, multiplied by the number of times the loop is executed.

3.3.5 Abstract Interpretation

An alternative method we can use is Abstract Interpretation via the interval domain, similar to the original implementation in [46]. This method iteratively calculates at every programme point, the interval of values each variable can take at each programme point. Then, programme slicing and a similar polytope analysis is carried out for loops that meet certain constraints, before deriving the final loop bounds.

This method is more involved in that it requires a control flow graph to be produced before any analysis can take place. In addition, it does not output a parametric solution, so therefore the analysis process needs to be repeated for each time the user specifies new values for the function parameters. However, it has the advantage that the user does not need to specify concrete values, but can also specify a range of values each function parameter can take at the function entry point.

4

Related works

Most Solidity analysis tools currently still revolve around programme verification, such as Oyente, Mythril, and Manticore. These tools focus on analysing reachable vulnerable states within a smart contract, and checks for common exploits such as reentrancy and unprotected self destructs, and ultimately differ from the objectives of this project. MadMax is a verification tool that focuses on detecting states that may result in an out of gas error, but also does not focus on calculating gas bounds for smart contract functions, and therefore remains out of scope. Here, we will discuss two related projects, GASTAP and VisualGas, which both focus on a similar objective of estimating gas costs with two different approaches.

4.1 GASTAP

4.1.1 Summary

GASTAP is a tool developed by Albert et al [47] that claims to be able to infer parametric gas bounds for smart contracts. These inferred bounds may depend on the function arguments provided, or on the blockchain and smart contract state. GASTAP conducts its analysis on the EVM bytecode level, for which the schedule of gas fees is defined.

GASTAP achieves this in 3 stages. First, it generates a *stack-sensitive control flow graph (S-CFG)* for the EVM bytecode. To do this, it first calculates the set of basic blocks in an EVM programme, and detects the set of valid jump destinations (marked by the `JUMPDEST` instruction). Then, they proceed via symbolic execution for each block to produce the stack state after executing each EVM instruction. The key idea here is that since `JUMP` instructions jump to the programme counter at the top of the stack, control flow when reaching a basic block therefore then depends on the set of stack items at that point which hold valid jump destinations. Thus, if two execution traces reach the same basic block with the same set and locations of jump destinations in their stacks, their nodes in the control flow graph can be safely merged. Else, a new node for the new stack state is pushed into the control flow graph, and the process repeats iteratively until the graph is consistent. Their earlier paper also provides a proof for how this model is sound and over-approximates the jumping information for a programme [43]. The tool they used to perform this step – EthIR, an extension of the Oyente tool – is also open source and available on their GitHub repository.

Next, GASTAP produces a *rule-based representation (GAS-RBR)* from the S-CFG generated in the previous step. Each for each block, a new rule is constructed, with the edges indi-

cating invocations of the generated rules. The original EVM instructions are wrapped in a *nop* functor within the GAS-RBR, to allow for precise calculation of the gas costs later on. This representation essentially abstracts the relationships between different items within the stack, as well as the boolean guards involved involved in making conditional jumps. Then, in order to calculate gas bounds, GASTAP also defines a set of semantics for the GAS-RBR, and separates the gas costs into two parts – the *opcode cost* and the *memory cost*. The opcode semantics takes the EVM opcode, the state of the stack, and the mapping of state variables when reading the opcode to return the corresponding gas cost of the instruction. The memory semantics similarly returns the highest memory slot used by an EVM instruction, since memory gas cost is only determined by each expansion of memory slot as introduced in Section 2.3.3.

Finally, the current GAS-RBR allows for the calculation of gas costs for concrete executions, for which all parameters involved are known concretely. However, in order to calculate parametric bounds, GASTAP first transforms the local memory and storage accesses by the GAS-RBR into local variables. Then, it makes use of a resource analysis tool called SACO to solve the constraints set by the GAS-RBR, and output a final parametric gas cost of a function call. This is the main contribution of this project, as there is no other tool available to our knowledge that is able to produce a parametric bound on gas costs.

4.1.2 Results

GASTAP is then evaluated against 34,000 Ethereum smart contracts, which was completed in 407.5 hours, or around 16 days of execution time. It was also found that GASTAP had a failure rate of 0.85%. Then, the gas bounds generated by GASTAP was evaluated against 4000 transactions of 300 top-valued Ether smart contracts. The average precision obtained was not published, but the overhead calculated from GASTAP was about 10% to 50% of the real transaction gas costs, with some overheads of up to 600% recorded. The authors claim this is because GASTAP has to take into account all possible input values, in order to calculate the worst case bounds.

4.1.3 Limitations

Some limitations of the GASTAP tool is that it only calculates gas bounds on a function-call level, similar to the Solidity compiler but with possibly greater precision. Our project aims to improve on this by calculating gas used for each line of Solidity code.

In addition, the GASTAP demo is no longer working or being maintained, and their latest supported version of Solidity is 0.7.1, which was released in September 2020. In addition, the SACO analyser used was developed by themselves and is closed-source, with no available binaries to the public, and we could not verify their claims.

4.2 VisualGas

4.2.1 Summary

VisualGas is a tool developed by Signer [48] with a similar goal for allowing developers to visualise the gas costs incurred by each line of code, and test best and worst case executions before deployment. However, in contrast to the static resource analysis of GASTAP, and our proposed implementation of symbolic execution, VisualGas makes use of dynamic

programme analysis to estimate gas bounds of each instruction. [48] claims that this method is more precise than static analysis, although may possibly lead to less coverage.

VisualGas was implemented in 3 parts. First, since dynamic analysis is performed, VisualGas needs a way to collect traces of executions of a given smart contract. It uses a Go-Ethereum client locally to execute transactions, and collects traces in the form of what instruction was executed at which programme counter (PC), the remaining amount of gas, the gas cost of each step, the call stack depth and memory state, as well as any storage slots accessed at each step. This trace collection is repeated for every new test input state that is generated. Then, it uses the source mapping of build artifacts generated by Truffle, and processes it to map programme counters to specific lines of code. The gas costs of each trace is then aggregated and further processed to take into account refunds and external calls. Finally, a histogram of gas costs for each line of code across all executed traces is output. For programme counters that were never executed, only the static gas costs are calculated, but this may not be accurate or sound.

Next, in order generate the test inputs, a fuzzer by Ambroladze et al [49] is used. This generates transactions to run all public functions within a smart contract, and uses a feedbackloop to adjust arguments so as to achieve high code coverage. It also takes into account the timestamp of the deployment transaction, and fuzzes the block timestamps to execute functions that require a certain time to have passed.

Finally, to visualise the analysis, a webserver built on Flask was used, which links all three analysis components together.

4.2.2 Results

VisualGas reports a 88% average (and 94% median) code coverage for their fuzzer with a limit of 5,000 transactions, when testing against a dataset of 30,400 contracts with less than 6,000 EVM instructions. They also report that larger contracts seemed to have lower coverage at that limit.

As for gas analysis, VisualGas did not evaluate their tool against a large dataset of contracts, or provide any measurement of precision for their analysis. Instead, they performed a case study on a TimedCrowdSale smart contract [50], and reported a running time of 45s and 82s for executing 5000 transactions and 10000 transactions respectively. They also claim a 99% code coverage for the SimpleToken contract, and a 95% coverage for the MyTimedCrowdSaleContract. However, no measurement of precision was provided for the contracts tested as well.

4.2.3 Limitations

Although VisualGas is closest to this project in terms of objectives, and aims to produce a gas cost estimate for every line of code, their implementation does not attempt to be sound or precise, considering the lack of evaluation for the precision of gas costs calculated. Rather, VisualGas at best serves to only provide a rough visualisation of which parts of code appear to be the most gas-consuming. In addition, their use of dynamic analysis in the form of programme traces results in a relatively average low code coverage of 88%, with a relatively high running time required to collect all traces. This is not ideal, since a large part of gas costs might still be hidden away within the code paths that were not traversed. Our

implementation addresses this via symbolic execution, which reasons about a programme path-by-path rather than input-by-input, and can be significantly more efficient.

VisualGas also did not state their latest supported Solidity version, although images from their paper suggests this is 0.4.25, which was released more than 3 years ago. There is also no source code provided, and therefore we are unable to test their tool.

4.3 Summary of Research

From our findings, we discover that no current existing tool performs the objectives of this project well. GASTAP has a sophisticated static analysis framework and can notably derive parametric gas bounds in terms of function arguments, but only provides these at a transaction level rather than a per line-of-code level. VisualGas performs dynamic analysis to derive gas costs for each line of code, but has lower coverage and the precision of their estimates remains unknown.

Our project therefore aims to improve on these tools, by providing a gas cost estimate for each line of code, and enabling high code coverage via symbolic execution. We also plan to enhance the gas estimation of loops by performing static analysis on the Yul intermediate representation, to derive a parametric bound on the number of loop invocations.

5

Evaluation

In order to evaluate the quality and effectiveness of the tool created from the project, we plan to investigate the following categories:

1. **Speed**, in terms of the median and average time needed to analyse a contract
2. **Coverage**, in terms of the percentage of EVM bytecode for which the gas estimates can be derived
3. **Precision**, in terms of the gas estimates calculated
4. **User experience**, in terms of the features of the tool and how easy they are to use

For items 1 and 2, we aim to evaluate our tool against a dataset of over 70,000 unique Ethereum smart contracts [51], and collect statistics regarding run time and code coverage. These will be compared with other existing tools such as GASTAP and VisualGas. The tools are not open source, and therefore we are only able to compare with the published statistics, and are unable to verify the results ourselves in a similar setting.

Then, for item 3, we aim to evaluate the gas bounds derived from our tool against the gas cost of real transactions, using 100 real transactions from 100 contracts randomly chosen from the dataset. Then, we can compare the precision of these gas bounds with those generated by the Solidity compiler. We can also compare the precision in terms of overhead with GASTAP. These can only be evaluated at a transaction level, since no other existing tool is able to calculate precise gas bounds at a line of code level.

Finally, for item 4, we aim to evaluate the features of our tool against other existing tools, such as the Remix IDE and VisualGas. We also plan on gathering user feedback based on the following metrics:

1. **Responsiveness**, or the total time wasted on waiting as a percentage of total time spent on performing a task
2. **System Usability Scale**, which is a questionnaire of ten items for rating overall user experience
3. **Error Occurence Rate**, which is the number of times an error occurs for each task performed
4. **Average Time on Task**, for the first attempt as well as for repeat attempts

5. **Hint rate**, which is the number of times a user has to ask for hints on how to perform a particular task

We plan on testing the tool with real users to gather these metrics, by asking them to perform the following tasks:

- Import a Solidity project
- Generate the EVM code mappings for the Solidity code
- Generate the Yul code mappings for the Solidity code
- Generate a heatmap of gas costs for the Solidity code

To consider the project a success, we would ideally like to achieve a speed comparable or better than existing gas analysis tools, a higher coverage than existing tools, and comparable or better precision than existing tools. This is because most existing tools only evaluate gas costs at a transaction level, and breaking it down into line of code level may incur some loss of precision. We would also like to create a rich user experience that has a high usability score, and is therefore simple to navigate but powerful to use.

6

Ethical considerations

Overall, there does not seem to be any major ethical concerns regarding the nature of this project. We have also investigated the licenses for the libaries we intend to incorporate within our tool, namely Mythril and the Solidity compiler, which respectively have the MIT and GPL-3.0 licenses, and should be permissive enough for our nature of work. However, there are still certain key points of consideration. First of all, because the project deals with decompiling Solidity code into its Yul intermediate representation and EVM code, and generating the mappings between them, they may reveal or expose vulnerabilities within the compiler generated code. With Ethereum smart contracts nowadays holding up to millions of dollars worth of funds, if such information is revealed to malicious actors, it could result in a permanent and irrevocable loss of funds.

In addition, this feature may also allow malicious actors to more easily obfuscate their code, by possibly implanting fragments of malicious inline assembly into template contract codes, claiming they serve another purpose. Then, users who do not analyse the code in detail and simply reuse them may at best lead to bugs in execution, or at worst loss of funds.

Next, there is no guarantee provided by our project about the precision or reliability of the gas analysis generated. Therefore, if the gas analysis is not accurate, and other projects do not carefully consider the gas used by their contract calls and simply rely on estimates derived from our tools, it may result in out of gas errors in future transactions, which may in turn lead to funds being locked forever and lost. We therefore intend to waive any liabilities derived from the use of our tool.

Also, generating the Yul intermediate representation is still an experimental function of the Solidity compiler that we make use of. Therefore, it is not assured to be completely correct, and users should not use the bytecode generated by our code directly when deploying contracts.

Finally, when collecting user data during the evaluation of our project, we must also consider the potential privacy risks and the right to withdraw from the survey. Users will not be asked to provide any personal information, and the data collected will be anonymised and aggregated, to prevent any leaks of sensitive information.

7

Project Plan

As of the writing of this report, the following foundational steps have been completed:

- Testing of the Solidity compiler for its capabilities and scope of debug logs emitted
- Testing of various symbolic execution engines (Mythril and Oyente) to investigate their capabilities
- Creation of a basic web application that is able to decompile a Solidity contract using the javascript Solidity compiler
- Research on previous gas estimation techniques for EVM, as well as various loop bounds analysis methods for symbolic execution

Moving forward, the plan for the project is as follows:

1. Building the compilation pipeline for the basic web app that automatically compiles Solidity code into both EVM and Yul IR. **Suggested timeline: 1.5 weeks (31 Jan - 9 Feb)**
2. Development of the mapping feature that allows compiled EVM and Yul code to be mapped to the corresponding Solidity code. **Suggested timeline: 1.5 weeks (10 Feb - 21 Feb)**
3. Extending the Mythril symbolic execution engine to provide aggregate gas estimates for each basic block. **Suggested timeline: 2 weeks (22 Feb - 7 Mar)**
4. Development of the heatmap feature that takes the gas estimates calculated and displays it in a heat map, on top of the compiled EVM and Yul IR. **Suggested timeline: 2 weeks (7 Mar - 21 Mar)**
5. Extension of the Solidity Yul compiler with our loop analysis algorithm, to emit parametric loop bounds within the output EVM. **Suggested timeline: 3 weeks (21 Mar - 11 Apr)**
6. Extension of web app to take into account the emitted parametric loop bounds during gas estimate calculation. **Suggested timeline: 1.5 weeks (14 Apr - 21 Apr)**
7. *(Stretch)* Development of a symbolic debugger using the Mythril engine, that allows the user to step through an execution and examine the values within the stack, the programme counter, and the gas used. **Suggested timeline: 4 weeks**

8. *(Fallback)* Development of a feature that builds and displays the control flow graph for both Yul and EVM bytecode **Suggested timeline: 2 weeks**
9. Conduct evaluation on the web application, and collect user feedback **Suggested timeline: 2 weeks (10 May - 24 May)**
10. Writing of draft report, and meet with supervisor **Suggested timeline: 2 weeks (24 May - 7 Jun)**
11. Finalise report and create and rehearse for final presentation **Suggested timeline: 2 weeks (7 Jun - 20 Jun)**

For items 5 and 6, if it is found to be unable to achieve the stated goal in time, or if the previous steps take longer than expected, the project will fallback into developing step 8 instead, which should be more manageable to complete. If the project goes very well and there is extra time left, we will focus on developing step 7, which would be an additional (but useful) feature for the tool. We have also included 2 weeks of buffer time for courseworks and examinations.

Bibliography

[1] Yoav Vilner. Some rights and wrongs about blockchain for the holiday season, 2018. URL https://www.forbes.com/sites/yoavvilner/2018/12/13/some-rights-and-wrongs-about-blockchain-for-the-holiday-season/. pages 1

[2] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, Dan Robinson. Uniswap v3 core. URL https://uniswap.org/whitepaper-v3.pdf. pages 1

[3] Aave - protocol whitepaper. URL https://github.com/aave/protocol-v2/blob/master/aave-v2-whitepaper.pdf. pages 1

[4] Robert Leshner, Geoffrey Hayes. Compound: The money market protocol. URL https://compound.finance/documents/Compound.Whitepaper.pdf. pages 1

[5] MakerDAO. The maker protocol: Makerdao's multi-collateral dai (mcd) system. URL https://makerdao.com/en/whitepaper/. pages 1

[6] Evan Kereiakes, Do Kwon, Marco Di Maggio, Nicholas Platias. Terra money: Stability and adoption. URL https://assets.website-files.com/611153e7af981472d8da199c/618b02d13e938ae1f8ad1e45_Terra_White_paper.pdf. pages 1

[7] Axie infinity whitepaper. URL https://whitepaper.axieinfinity.com/. pages 1

[8] The sandbox whitepaper. URL https://installers.sandbox.game/The_Sandbox_Whitepaper_2020.pdf. pages 1

[9] Defi llama: Ethereum tvl. URL https://defillama.com/chain/Ethereum. pages 1

[10] Ethereum average gas price chart. URL https://etherscan.io/chart/gasprice. pages 2

[11] Harry Robertson. Ethereum transaction fees are running sky-high. that's infuriating users and boosting rivals like solana and avalanche. URL https://markets.businessinsider.com/news/currencies/ethereum-transaction-gas-fees-high-solana-avalanche-cardano-crypto-blockchain-2021-12 pages 2

[12] Ishan Pandey. Ethereum's high gas fees is limiting on-chain activities. URL https://hackernoon.com/ethereums-high-gas-fees-is-limiting-on-chain-activities. pages 2

[13] Samuel Wan. Uniswap activity sends ethereum gas fees sky high. URL https://www.newsbtc.com/news/ethereum/uniswap-activity-sends-ethereum-gas-fees-sky-high/. pages 2

[14] Monika di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78, 2019. doi: 10.1109/DAPPCON.2019.00018. pages 2

[15] *Solidity Documentation Release 0.8.12*, 2021. URL `https://buildmedia.readthedocs.org/media/pdf/solidity/develop/solidity.pdf`. pages 2, 7

[16] *Solidity v0.7.0 Breaking Changes*, 2021. URL `https://docs.soliditylang.org/en/v0.7.0/070-breaking-changes.html`. pages 2

[17] *Solidity v0.8.0 Breaking Changes*, 2021. URL `https://docs.soliditylang.org/en/v0.8.10/080-breaking-changes.html`. pages 2

[18] ConsenSys. *A Definitive List of Ethereum Developer Tools*, 2021. URL `https://media.consensys.net/an-definitive-list-of-ethereum-developer-tools-2159ce865974`. pages 2

[19] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiaopu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing*, 9(3): 1433–1448, 2021. doi: 10.1109/TETC.2020.2979019. pages 2

[20] Ethererik. Governmental's 1100 eth jackpot payout is stuck because it uses too much gas, 2016. URL `https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/`. pages 2

[21] Bitcoin Suisse. Compatible competition - empowering or encroaching on ethereum?, 2016. URL `https://www.bitcoinsuisse.com/research/decrypt/compatible-competition-empowering-or-encroaching-on-ethereum`. pages 2

[22] Matt Godbolt. Optimizations in c++ compilers. *Commun. ACM*, 63(2):41–49, jan 2020. ISSN 0001-0782. doi: 10.1145/3369754. URL `https://doi.org/10.1145/3369754`. pages 2

[23] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Dec 2021. URL `https://ethereum.github.io/yellowpaper/paper.pdf`. pages 3, 4, 5, 6

[24] Ainur R. Zamanov, Vladimir A. Erokhin, and Pavel S. Fedotov. Asic-resistant hash functions. In *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 394–396, 2018. doi: 10.1109/EIConRus.2018.8317115. pages 4

[25] Ethereum. Proof-of-work (pow), . URL `https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/`. pages 4

[26] Nick Szabo. Smart contracts, 1994. URL `https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html`. pages 4

[27] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), Sep. 1997. doi: 10.5210/fm.v2i9.548. URL `https://firstmonday.org/ojs/index.php/fm/article/view/548`. pages 4

[28] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016. doi: 10.1109/ACCESS.2016.2566339. pages 4, 5

[29] Ethereum. Ethereum virtual machine (evm), . URL `https://ethereum.org/en/developers/docs/evm/`. pages 5

[30] Solidity. *Introduction to Smart Contracts*, 2021. URL `https://docs.soliditylang.org/en/latest/introduction-to-smart-contracts.html`. pages 5

[31] Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014. URL `http://arxiv.org/abs/1407.3561`. pages 5

[32] Samuel Williams, William Jones. Arweave lightpaper, Apr. 2018. URL `https://www.arweave.org/files/arweave-lightpaper.pdf`. pages 5

[33] Ethereum. Gas and fees, . URL `https://ethereum.org/en/developers/docs/gas/`. pages 6

[34] Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, Abdelhamid Bakhta. Eip-1559: Fee market change for eth 1.0 chain. Apr 2019. URL `https://eips.ethereum.org/EIPS/eip-1559`. pages 6

[35] Ethereum. Design rationale, . URL `https://eth.wiki/en/fundamentals/design-rationale`. pages 6

[36] Ethereum. Smart contract languages, . URL `https://ethereum.org/en/developers/docs/smart-contracts/languages/`. pages 7

[37] Fabian Vogelsteller, Vitalik Buterin. Eip-20: Token standard. URL `https://eips.ethereum.org/EIPS/eip-20`. pages 7

[38] William Entriken, Dieter Shirley, Jacob Evans, Nastassia Sachs. Eip-721: Non-fungible token standard. URL `https://eips.ethereum.org/EIPS/eip-721`. pages 7

[39] OpenZeppelin. Openzeppelin contracts. URL `https://github.com/OpenZeppelin/openzeppelin-contracts`. pages 7

[40] Solidity. Expressions and control structures, . URL `https://docs.soliditylang.org/en/latest/control-structures.html`. pages 7

[41] Solidity. Yul, . URL `https://docs.soliditylang.org/en/latest/yul.html`. pages 9

[42] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012. ISBN 978-0-12-383872-8. pages 12

[43] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Analyzing smart contracts: From EVM to a sound control-flow graph. *CoRR*, abs/2004.14437, 2020. URL `https://arxiv.org/abs/2004.14437`. pages 12, 20

[44] Bernhard Mueller. Smashing ethereum contracts for fun and real profit. *HITBSecConf*, 2018. URL `https://github.com/b-mueller/smashing-smart-contracts/`. pages 13

[45] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978309. URL https://doi.org/10.1145/2976749.2978309. pages 13

[46] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *2009 International Symposium on Code Generation and Optimization*, pages 136–146, 2009. doi: 10.1109/CGO.2009.17. pages 13, 18, 19

[47] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román Díez, and Albert Rubio. Don't run on fumes — parametric gas bounds for smart contracts. *Journal of Systems and Software*, 176:110923, 02 2021. doi: 10.1016/j.jss.2021.110923. pages 20

[48] Christopher Signer. Gas cost analysis for ethereum smart contracts. 2018. URL https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/312914/TMeineOrdnerMaster-ArbeitenHS18Signer_Christopher.pdf. pages 21, 22

[49] Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. Learning to fuzz from symbolic execution with application to smart contracts. pages 531–548, 11 2019. ISBN 978-1-4503-6747-9. doi: 10.1145/3319535.3363230. pages 22

[50] Christopher Signer. Timedcrowdsale. URL https://github.com/ABBDVD/TimedCrowdsale. pages 22

[51] Martin Ortner and Shayan Eskandari. Smart contract sanctuary. URL https://github.com/tintinweb/smart-contract-sanctuary. pages 24