Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

SOLBOLT – a Compiler Explorer and Smart Contract Gas Usage Tracker for Solidity

Available at www.solbolt.com

Author:
Richard Jun Wei Xiong

Supervisor:
Prof William Knottenbelt

Second Marker:
Dr Maria Vigliotti

June 20, 2022

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

Abstract

With the recent boom in interest for DeFi platforms and NFTs on smart contract enabled blockchains such as Ethereum, the cost of performing transactions in terms of gas prices has increased tremendously. In order to make these transactions more accessible for general usage, there is a clear need not only for the better scaling of blockchains, but also for developers to create more gas efficient applications. However, most of the developer tools that exist today mainly focus on security verification rather than gas analysis for Solidity smart contracts. As such, the main goal of this project is to create a powerful yet user-friendly tool, that generates detailed gas estimates and detects potential issues, so as to help developers more easily pinpoint which parts of their contracts can be refactored to save gas.

The tool we created, SOLBOLT, makes use of symbolic execution by extending the Mythril engine, via its plugin and hooks system. SOLBOLT is able to: (i) generate a detailed code mapping from the compiled EVM bytecode to the original Solidity bytecode, via a simple user interface built using React Typescript and inspired by the popular GCC compiler explorer tool Godbolt, and a backend running the Solidity compiler and Mythril engine, served using Flask and a Celery task queue; (ii) estimate the gas usage for each code mapping, by using a custom gas meter plugin and gas hook that we extended the original Mythril engine with, which stores the total gas used by each code section for each symbolic transaction; (iii) estimate the gas usage for each function and each loop iteration, by developing a function tracker plugin which made use of the EVM dispatcher routine to infer the function that was taken by each symbolic execution path, as well as a loop gas meter plugin which stores the current executional trace and is able to detect if the trace is currently in a loop; (iv) allow users to directly import verified contracts from the Etherscan API; and (iv) automatically detect possible gas inefficient patterns and suggest how they can be fixed, namely the detection of storage mutations within a loop.

SOLBOLT is then evaluated over 1,392 smart contracts, by comparing the estimated gas costs for each function to the actual transaction gas cost. The median estimated gas cost was found to be 129.3% of the concrete gas costs, while the median coverage was 99.1%. These results are then compared to other previous studies, and were found to be comparable with the current state-of-the-art. Finally, a case study was performed to determine how a developer might use the tool to optimise a smart contract.

Acknowledgments

First of all, I would like to express my thanks and gratitude to my supervisor, Dr William Knottenbelt, for all his insightful advice and support throughout the year despite the current difficult circumstances. I would also like to thank my second marker, Dr Maria Vigliotti, for her assistance in helping to set the scope of this project. Finally, I would like to thank those who have helped me with the technical details of the Solidity compiler over at the Ethereum Gitter forum, for being so patient and kind with my questions. I am incredibly proud to be able to complete my degree with a project that I am keenly passionate about, and would love to work more on in the future.

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 Motivation

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 NFT-powered gaming ecosystems such as Axie Infinity [6] and the Sandbox [7].

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 [8]. 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 [9]. Transactions are costing up to $63 on average in November [10], and this has made the Ethereum blockchain inhibitive to use for most users [11]. 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 [12]. 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 [13]. The Solidity compiler itself does offer an estimate for gas usage [14], 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 [15] [16], 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 Section 3.

Currently, most Solidity developers simply make use of general tools such as the Remix IDE, Truffle and Ganache for smart contract development [17]. 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 [18], or worse, possibly render a contract call unusable due to reaching the gas limits of each block [19]. More recently, in the case of the Otherside NFT sales, up

to $100 million in gas could have been saved with a just few lines of optimisations [20]. Clearly, 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 Objective

The goal of this project is therefore to create a tool that helps Solidity developers:

- better understand how the compiled EVM bytecode relates to their Solidity source code, and how different optimisation options affect the compiled bytecode
- easily and intuitively understand the gas profile of their smart contract, and the parts of their code that use up the most amount of gas
- quickly locate and identify potential gas savings, as well as find possibly gas inefficient patterns

1.3 Contributions

With this 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 contributions this project has achieved:

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

   Similar to Godbolt which is able to easily map and visualise which assembly instructions correspond to its high-level C++ code, we created a tool that achieves a similar result with EVM assembly bytecode and its corresponding Solidity code. This tool is easy to explore and read via an intuitive user interface built using React Typescript, and is as fine-grained as the compiler allows us to.

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, our tool conducts an analysis of gas consumption for each line of Solidity code, instead of simply at a function call level. This is performed using symbolic execution, commonly used for security analysis, to traverse each possible execution path, and then aggregating the gas costs for each code section, via a gas meter plugin and gas hook which we built specifically for this purpose. Our tool is also able to estimate gas costs for each function and loop iteration, via a function tracker plugin that makes use of the EVM dispatcher routine to determine which function the current execution path has taken, as well as a loop gas meter that makes use of a special loop finding algorithm to determine the iteration gas cost. This is the first time we know of such an analysis method being used for gas estimation. We also perform a detailed evaluation over the accuracy and coverage that this method provides.

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

   To make visualising gas usage simpler, our tool is able to generate 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. The tool also calculates additional profiling information, such as the histogram of the number of code sections that consume a certain amount of gas.

4. **Automatic identification of gas inefficient patterns**

   Our tool is also able to automatically identify some common gas-hungry code patterns, namely the detection of storage mutations within a loop, and provide suggestions on how to improve them.

1.4 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 incorporated 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 compiling Solidity code into EVM code, and generating the mappings between them, they may possibly 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 tool, 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.

1.5 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 Mythril symbolic execution engine. In Chapter 3, we briefly examine two related projects, as well as their results and limitations. In Chapter 4, we describe the overall architecture and implementation of our tool, SOLBOLT, for the estimation of gas costs using symbolic execution. In Chapter 5, we evaluate the performance and usability of our tool, firstly via a systematic test on 1,392 Ethereum contracts to determine its accuracy and robustness to real world usage, and secondly via a case study of a gas-inefficient contract to understand how a developer might make use of the insights provided by our tool. Finally, in Chapter 6, we discuss a summary of this project's achievements as well as its limitations, as well as some ideas for future work that can be carried out.

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 [14]. 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 2.2, 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.

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. 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`.

Due to the additional information that Yul provides in terms of control flow and stack allocation, the project originally planned to perform static loop analysis on the generated Yul intermediate representation. This idea was, however, abandoned for the following reasons:

1. Yul is only released recently in Solidity compiler v0.7.5 and above, and this severely limited the dataset of contracts available for evaluation purposes.

2. Performing the static analysis required some additional instrumentation on the compiler level, and in order to support all the compiler versions required for evaluation, one has to modify and recompiler every compiler version between v0.7.5 and v0.8.14 (the latest as of today).

3. Yul is still an experimental feature, and will likely be subject to frequent changes in terms of its language semantics. As such, it is possible that any static anlysis performed using the current state of Yul may be rendered obselete in the near future. Performing analysis on EVM assembly however, is more likely to still remain relevant for the time being, as it is the fundamental machine code that the Ethereum Virtual Machine runs on, and any changes to it will require a hard fork of the entire blockchain.

As such, instead of performing loop bounds inference on the Yul intermediate representation, the project has instead performed loop detection on the EVM assembly, and calculated per iteration gas costs on any possible loops, which will be explained in detail later in the implementation section.

2.5 Mythril

2.5.1 Symbolic execution

Symbolic execution is a technique of programme analysis that involves the execution of a programme using symbolic inputs rather than concrete inputs [42]. Then, it proceeds down each possible execution path if the conditions for it are satisfiable using a SMT solver, in order to determine some security properties of a programme, such as whether a particular code path is reachable or not, or if a certain assertion can be violated during the execution.

Compared to other techniques such as dynamic analysis, which evaluates a programme input-by-input rather than path-by-path, symbolic execution may provide increased efficiency and coverage, especially if the input space is large. This may be the case for Ethereum smart contracts, where inputs are often 256-bit unsigned integers, user-defined structs, or even dynamic arrays of strings and bytes. Analysing a smart contract via dynamic analysis may require thousands of concrete transactions to achieve a desirable level of coverage [43], while symbolic analysis may only require traversing a few hundred paths based on our testing.

A common problem when using symbolic execution for programme analysis is path explosion. The number of possible execution paths increases exponentially with an increase in code size, and can possibly be infinite in the case of unbounded loops. However, due to having to meet the gas limit of Ethereum blocks, smart contracts are typically much simpler compared to executable desktop programmes. The Ethereum Virtual Machine is also much easier to model due to its deterministic and sequential nature with a limited number of opcodes, unlike a traditional operating system with its complex system calls (which may also be asynchronous in nature) and concurrency handling. As such, these observations make symbolic execution a suitable analysis method for smart contract gas analysis, and we later find that even using a simple bounded loop strategy allows us to achieve a very high coverage for most smart contracts evaluated.

There are several symbolic execution engines that already exist for the Ethereum Virtual Machine. For example, tools such as Oyente [44] (and EthIR, an extension of it,) are able to perform symbolic execution and produce a control flow graph of the basic blocks executed. 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). There are also other libraries such as Manticore [45], but they were poorly documented and difficult to extend.

Ultimately, the symbolic execution engine chosen for the project is Mythril [46],

which is a symbolic execution engine originally designed by ConsenSys for the verification of smart contracts and analysis of possible path conditions that are able to induce undesired behaviour, such as killing a smart contract. This project then extended Mythril's original capabilities, making use of its detailed of the EVM and how it handles each opcode, to track gas usage along each execution path and for each piece of code.

As a large part of the work done by this project is on directly extending the Mythril execution engine, it is helpful to understand how its internals work.

2.5.2 Global states

Mythril keeps track of the different program execution states via "global states", each of which is a complete snapshot of the Ethereum virtual machine at a given program point. Each global state contains the following key information:

1. **World State**

   The World State, as described in the Ethereum yellow paper, contains information about each account and their balances, as well as their storage state and bytecode if they are smart contract accounts. It also keeps track of the current transaction sequence.

2. **Environment**

   The Environment keeps track of the current execution environment for the symbolic executor, and contains the current active smart contract account, symbolic or concrete values for the sender, origin and calldata, as well as values for the gas price.

3. **Machine State**

   The Machine State, represented by μ in the Ethereum yellow paper, keeps track of the current internal state of the Ethereum Virtual Machine, such as the stack, the programme counter, the state of the memory, as well as the amount of gas used so far.

4. **Annotations**

   Annotations can also be attached to the global state, which are objects that can be accessed and modified by custom plugins to keep track of information relevant to the plugin. Plugins are passed from one global state to the next after performing the current opcode, and reset with each new transaction. These are used extensively by this project for extending Mythril with custom gas analysis plugins.

2.5.3 Strategies

Given a work list of global states to explore and execute, the symbolic executor can employ the following strategies in order to decide which paths to execute first:

1. **Breadth First Search**: Selects the next state from the front of the work list
2. **Depth First Search**: Selects the next state from the end of the work list
3. **Naive Random Search**: Selects the next state randomly from the work list with equal probability
4. **Weighted Random Search**: Selects the next state randomly from the work list with probability inversely proportional to the depth of that state within the list

After selecting the next state and executing its current opcode, the symbolic executor may obtain zero new states (if a `REVERT`, `STOP` or `RETURN` was executed), one new global states (for most opcodes), or multiple global states (if there was a conditional branch). It then appends the new states (if any) to the back of the work list, and repeats the process until the work list is empty.

Each strategy can also be extended with the bounded loops strategy (which stops execution once a given loop bound is reached), or call depth limit strategy (which limits the depth of the call stack).

2.5.4 The LASER EVM

The entire symbolic execution pipeline is then implemented as the LASER EVM. In summary, the LASER EVM proceeds via the following steps:

1. The symbolic executor first instruments the LASER EVM with the necessary plugins, and initialises the world state, either with symbolic values or with user-configured concrete values.
2. The LASER EVM sends a contract creation transaction, which creates the active smart contract account that we want to perform our analysis on. This also returns a list of initial open global states, which the symbolic executor will explore.
3. The LASER EVM next begins executing the first transaction on the contract that was created. It selects the first global state to execute using the current strategy, and then calls the function responsible for modelling the current opcode. One or more new global states are then created from the current global state, and appended to the end of the work list.
4. If `STOP` or `RETURN` was reached, a transaction end signal is sent, and a new open global state is created by copying the previous world state of the ending

transaction. These will be executed in the next transaction. If `REVERT` was reached, all changes are instead reverted.

5. Execution continues until the work list is empty. Then, the entire process repeats with the new open global states being executed in the next transaction, until the transaction limit defined by the user is reached.

2.5.5 Plugins and Hooks

The LASER EVM can be easily extended with additional functionality using Mythril's plugin and hooks system. Additional custom plugins can be instrumented into the LASER EVM, by registering certain functions to be called by hooks. These hooks are then invoked at several particular points of execution, such as:

- At the start and end of the entire symbolic execution pipeline
- At the start and end of each symbolic transaction
- When a new global state is being added or executed
- During the execution of an opcode (but before the new global state is returned), for a particular opcode or for all opcodes
- Before or after the execution of an opcode (with the initial global state or new global states returned), for a particular opcode or for all opcodes

Each hook, when invoked, calls all the functions that are registered with it, and usually also passes the current global state or list of global states, depending on its type. The functions (which are part of a plugin) can then make use of custom annotations within the global state to keep track of relevant properties about the current execution state or perform custom tasks.

This project makes extensive use of the modularity and ease of customisation provided by this plugin system, in order to extend Mythril's capabilities for gas cost analysis, as will be explained in chapter 4.

3

Related works

Most Solidity analysis tools currently still revolve around programme verification, such as Oyente, Mythril, and Manticore [44] [46] [45]. 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 [47], 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.

3.1 GASTAP

3.1.1 Summary

GASTAP is a tool developed by Albert et al [48] 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 [49]. 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 indicating 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.

3.1.2 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.

3.1.3 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 is able to improve on this by calculating gas used for each line of Solidity code. In addition, GASTAP is unable to calculate parametric gas bounds for recursive functions by their assumptions. Our tool is able to not only calculate per-iteration gas costs for recursive loops, but also hidden loops generated by the compiler.

Moreover, 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. The SACO analyser they used was also developed by themselves and is closed-source, with no available binaries to the public, and we could not verify their claims.

3.2 VisualGas

3.2.1 Summary

VisualGas is a tool developed by Signer [43] 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. [43] 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 [50] is used. This generates transactions to run all public functions within a smart contract, and

uses a feedback loop 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.

3.2.2 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 [51], 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.

3.2.3 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 low average 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.

3.3 Discussion

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 is therefore able to improve on the detail of insights provided by these tools, by calculating a gas cost estimate for each line of code, and enabling high code coverage via symbolic execution. Our symbolic execution engine, while unable to determine the parametric gas costs as a closed-form formula, is also able to track the per-iteration gas usage for any loops encountered, including recursive and hidden loops.

4

Design and Implementation

4.1 Feature summary

This project has successfully built and deployed SOLBOLT, a compiler explorer and gas analysis tool, which is available for all to use on the live website at solbolt.com [52]. Next, we shall examine some of its key features, with reference to a summary graphic shown in Figure 4.1.

4.1.1 Solidity Compilation

The tool supports the compilation of contracts written in Solidity version 0.4.17 to 0.8.13, as well as support for EVM version from `homestead` to `berlin`. This is, to our knowledge, the most up-to-date and widest available compiler support for any Solidity gas analysis tool so far. The user also has the option to enable optimisation, as well as customise advanced options such as enabling the peephole optimiser, the inliner, the common subexpression eliminator, and more, as seen in Figure 4.2a. This may be helpful for developers or researchers investigating the effects of different optimisation options.

The tool also supports the compilation of contracts spread over multiple source files, as well as compiling multiple contracts from the same source. For verified contracts that are already deployed on chain, the tool also has Etherscan integration, and can directly load a contract from a verified Ethereum address for compilation, using Item 1.5 as indicated in the summary graphic.

After successful compilation of the source, the EVM assembly opcodes will be displayed on the right "Assembly viewer" pane. Sections of the opcodes will be colour-coded based on its mapping to the Solidity source code, similar to the Godbolt tool that inspired this project. Hovering over a section will also highlight and bring to

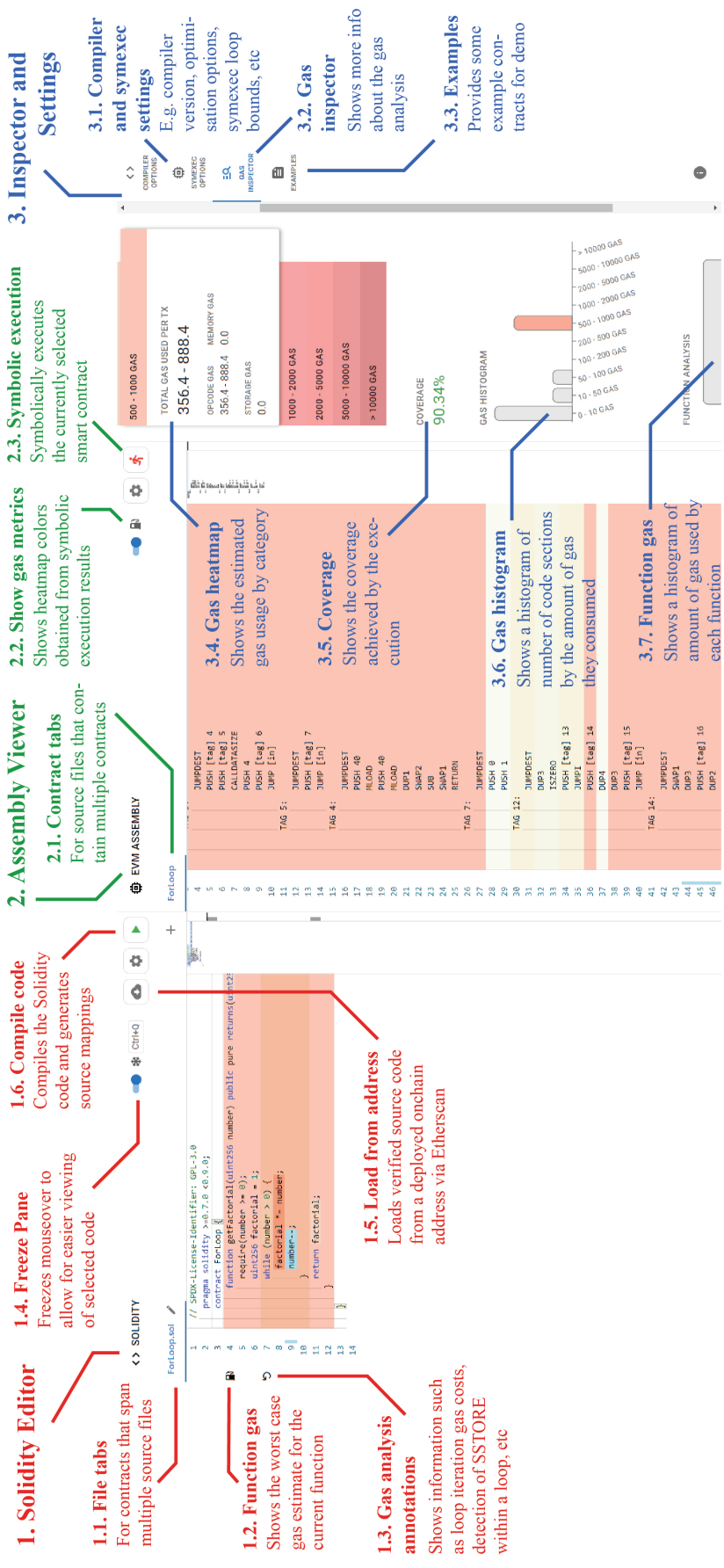
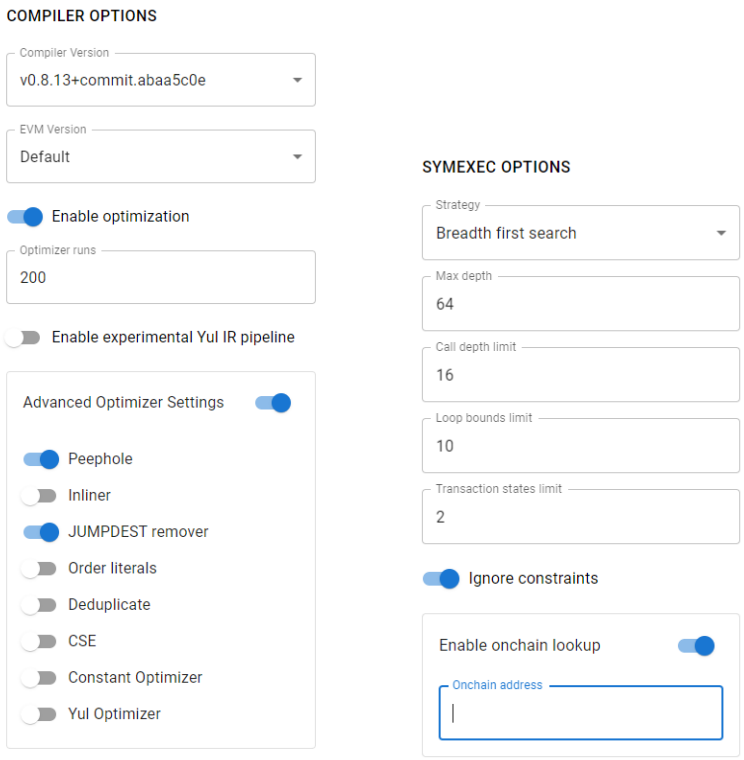


Figure 4.1: Summary of SOLBOLT features

view its corresponding mapped section on the other pane, which can be frozen via a keyboard shortcut.



(a) Compiler options

(b) Symbolic execution options

Figure 4.2: SOLBOLT options

4.1.2 Symbolic Execution

Once the source is compiled, users are able to run a symbolic execution using Item 2.3. The symbolic execution instance will try to execute each path of the contract bytecode, and derive gas estimates for each mapping generated by the compiler. This is indicated to the user as a coloured heatmap for quick identification of the gas-costly sections, with lighter yellow shades consuming fewer gas units, while the darker red shades consuming greater gas units.

Next, it will also calculate worst case gas estimates for each function it detects, as well as per-iteration gas costs for each loop, including hidden loops within compiler generated code. Finally, the symbolic executor can also detect common gas inefficient patterns, such as modifying a storage variable within a loop, which could possibly be fixed or refactored for additional gas savings.

Each symbolic execution is run with a timeout of 120 seconds, and users can customise some of its settings, such as the path traversal strategy, the maximum depth,

the call depth limit, the loop bounds limit, and the number of transactions to run, as seen in Figure 4.2b.

This may be important to tweak for different contracts to achieve the best results, depending on how complex they are, how many times a loop is expected to run, and so on. The user can also enable the Z3 SMT solver, which prunes paths that are not reachable as the constraints placed on them are not satisfiable. Users can also provide a on chain address for symbolic execution, and the engine will make use of the live concrete storage state of that address instead of a symbolic storage state.

4.1.3 Gas Inspector

Finally, the user is able to refer to the gas inspector tab on the right for a more detailed gas analysis report, as indicated by Item 3. Here, there is a gas heatmap legend indicating the total estimated gas used per transaction by the code section that is currently highlighted, as well as the gas used by the different categories, as seen with Item 3.4. It also shows the coverage percentage achieved by the symbolic execution, shown with Item 3.5.

The gas histogram, as seen with Item 3.6, shows a gas profile of the smart contract by plotting the total number of code sections that fall under a certain gas usage class. Lastly, Item 3.7 shows a histogram of worst case gas usage by each executed function.

4.2 Architecture Design

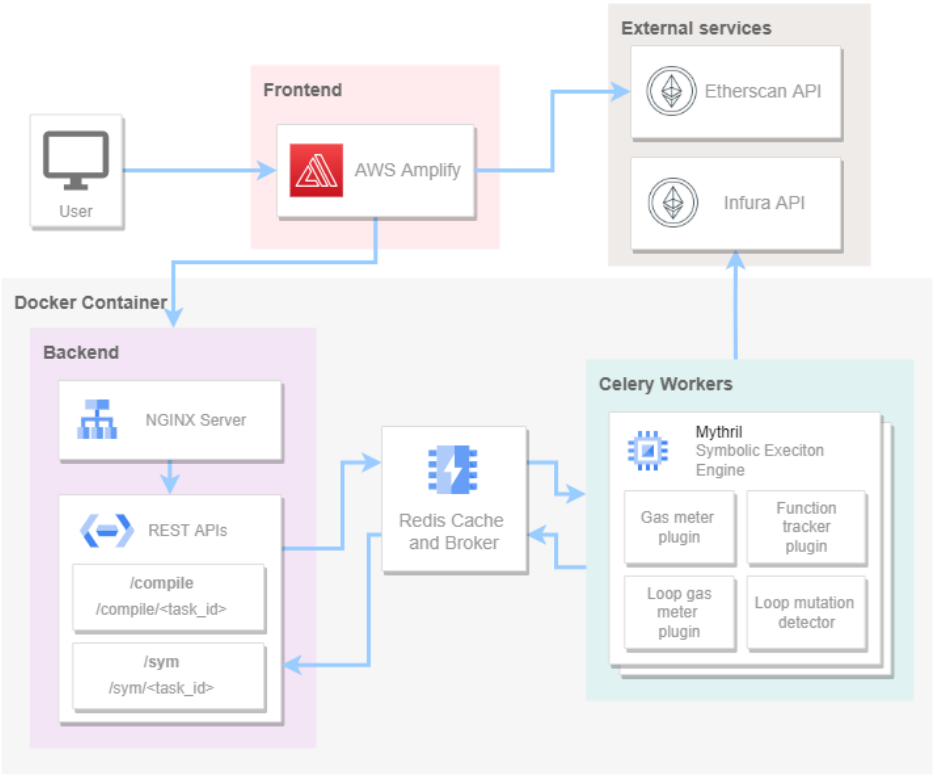


Figure 4.3: SOLBOLT architecture overview

4.2.1 UI Elements and Redux State Management

The SOLBOLT frontend is built from scratch using React (Typescript) 17.0.2 and Material UI 5.3.1. It is designed as a Single-Page Application (SPA), with three main panels for simple simple navigation, as seen in the screenshot in Figure 4.1. We chose to develop SOLBOLT fully in Typescript rather than Javascript, because although this required the definition of additional interfaces and typing of arguments, it also provided benefits such as code hints and error highlighting when the expected types of objects did not match.

A common challenge we faced when designing the frontend was having to pass and manage props deeply into nested components when using React. This was addressed by making extensive use of redux, `useContext` and hooks. For example, for managing the contract state, we created an `ContractsContext` that included the `ContractsState`, as well as functions to update the `ContractsState`, as seen in Listing 4.1.

```
1  const ContractsContext =
2    createContext<
3      [ContractsState | undefined, {
4        updateContract:
```

```
5           ((filename: string, name: string, contract: ContractJSON) => void) |
                undefined,
6         updateAllContracts:
7           ((contracts: {[fileName: string]: {[contractName: string]: ContractJSON}},
8             ast: {[name: string]: any}) => void) | undefined
9       }]
10    >([undefined, {
11          updateContract: undefined,
12          updateAllContracts: undefined
13      }]);
```

Listing 4.1: `ContractsContext` used for managing contract state

`ContractsContext` also has a `reducer`, which takes an update type and a payload. Depending on the type of the update, the `reducer` will modify the state accordingly, making use of relevant data in the payload, as seen in Listing 4.2

```
1   function reducer(state: ContractsState, { type, payload }: { type: UpdateTypes,
        payload: PayloadType | undefined}) {
2     switch (type) {
3       case UpdateTypes.UPDATE_CONTRACT: {
4         /* Perform update for one contract */
5         ...
6       }
7
8       case UpdateTypes.UPDATE_ALL_CONTRACTS: {
9         /* Perform update for ALL contracts */
10        ...
11      }
12
13      default: {
14        throw Error(`Unexpected action type in ContractsContext reducer: '${type}'.`)
15      }
16    }
17  }
```

Listing 4.2: Reducer used in `ContractsContext`

This `reducer` is called by the `Provider`, which lives near the root of the React DOM, above all of the components that wish to use the context. The class then exports hooks that can be easily used by any component that require the accessing or updating the contract state, such as `useContract`, `useUpdateContract`, and so on.

This method eliminates the need to pass props for shared information used frequently by many components across different routes, such as the application state, mappings state, highlighted class and contracts state. Updates are also handled automatically by React, because whenever the redux state is changed, the components dependent on it are also rerendered. This also makes it simple to store these states persistently in local browser storage, such that the user's session is restored after launching SOLBOLT again.

4.2.2 Monaco Editor

For the Solidity Editor and the EVM assembly viewer, we made use of the Monaco Editor, also used in Microsoft VSCode and the Godbolt compiler explorer. Although other similar libraries such as CodeMirror were lighter and had better mobile support, Monaco was the only editor that had native support for Solidity syntax highlighting, as well as a simpler system for handling code decorations and mouse handlers.

The current state of code decorations and mouse handlers are kept track of using a `useRef` variable, and destroyed when a new instance is being set to prevent memory leaks. Monaco also made use of a model and viewstate, which stores the content of an editor as well as the current view and undo stack. These can be saved and loaded as needed when switching tabs.

4.2.3 Etherscan Integration

The SOLBOLT frontend is directly integrated with the Etherscan API, which is a popular blockchain explorer tool and has a collection of over 160,000 verified contract source codes to date [53]. Etherscan now also supports multiple source files for verification, and as such, SOLBOLT also introduced tabs support for editing and compiling multiple files. The user simply needs to input the Ethereum address they wish to analyse, and SOLBOLT will automatically send a request and parse the Etherscan result, before updating the view with the retrieved source files.

4.2.4 Backend and REST API

Requests to the SOLBOLT backend are first routed through the NGINX reverse-proxy, which also acts as a rate limiter. These are then forwarded to the API endpoints, built using Flask and served with Gunicorn. Each request for initiating a compilation or symbolic execution task will generate a new task ID, which is returned in the response. This task request is also simultaneously published to a Redis broker, which are being listened to by workers. The client keeps track of the task ID, and will poll for the task result periodically, until the task is completed and they are made available.

There is no model or database used in this project, and no Solidity code or symbolic execution results are stored permanently, other than being cached by Redis.

4.2.5 Celery Task Queue

Since the published tasks take a significant amount of time to complete, much longer than a HTTP request typically remains alive for, we had to manage the task execution

on a separate task queue using Celery. Two Celery workers on different threads listen to tasks on the Redis broker, and pick them up as soon as they are ready and available. Each worker runs an instance of the extended Mythril symbolic execution engine or Solidity compiler, and once the job is complete, the results are stored within the Redis cache. These will be accessed and returned by the API endpoints upon the next client poll.

4.2.6 Deployment Pipeline

The frontend is built and served statically on AWS Amplify, which also utilises its global content distribution network for faster access. This is connected with the Github repository, and a new build is launched whenever a new commit is pushed.

The backend is instead hosted on a Hetzner Cloud instance with 3 AMD EPYC virtual cores and 4GB of RAM, running on Ubuntu 22.04 LTS. All backend services — the NGINX proxy, the Gunicorn instance, the Redis server, the Celery workers, and the SSL/TLS Certbot — are spun up at the same time using `docker-compose`, and run within a Docker container. Deploying a new version is not automated, but requires manual pulling of the new Github commit, and then rebuilding the Docker containers.

Best practices for security are also observed at all times, such as placing all API keys and production secrets within environment variables. The backend is also protected from malicious bots and denial-of-service attacks via custom jails with `fail2ban`.

4.3 Mythril Extensions

4.3.1 Gas Meter Plugin and Gas Hooks

Originally, Mythril already has some gas tracking capabilities, but it only kept track of the total gas used up at each program point, and did not store any information about which sections of code used up how much of that gas, therefore having little utility for gas analysis. In addition, the original machine state counted `SSTORE` and `SLOAD` storage opcode gas costs together with the other opcodes, and therefore only stored the opcode and memory gas usage. However, because storage costs often represent the majority of gas usage in most smart contracts, we felt it was informative to separate out the storage costs from the rest of the opcode costs.

We therefore built a custom gas meter plugin, whose purpose is to keep track of the average total amount of gas used per symbolic transaction by each Solidity code section. This will then be presented to the user in the form of a gas heatmap.

The plugin works by storing a global gas meter object that in turn, stores a map of

each code section to a struct containing the following information:

- `min_opcode_gas_used` and `max_opcode_gas_used` — stores the sum of the lower and upper bounds of the opcode gas estimated to be used
- `mem_gas_used` — stores the sum of the memory gas estimated to be used, as a result of memory expansion during execution
- `min_storage_gas_used` and `max_storage_gas_used` — stores the sum of the lower and upper bounds of the storage gas estimated to be used by `SSTORE` and `SLOAD` opcodes
- `num_invocations` — stores the total number of times that opcodes, which are mapped to this code section, are invoked throughout all transactions
- `num_tx` — stores the total number of transactions where opcodes mapped to this code section are executed

The plugin also uses a local gas meter object, which is stored in each global state as part of the gas meter annotation. This local gas meter contains the same information as above, but only for the current transaction trace.

Another challenge we faced while implementing the gas meter was the fact that none of the available hooks were suitable for calling the plugin. The pre-instruction hook was not suitable for this plugin, because the opcode has not been executed yet, and the gas is therefore not incremented. The instruction hooks are not suitable either, because they are called with the initial global state, rather than the new global states with updated gas costs. The post-instruction hooks are also not suitable, because although they had the updated gas costs after executing the opcode, the programme counter (PC) has also been incremented to the next instruction, and therefore we cannot attribute gas costs to the correct previous instruction that has used it.

As such, we had to design and implement a new custom hook for the Mythril engine, called the gas hook, which is called right after the machine state within the current global state is updated with the latest gas costs, but before the PC is incremented.

For every instruction, at the gas hook, the plugin runs the following pseudocode in Algorithm 1. The gas plugin first gets the current gas meter annotation for the current global state, and then tries to obtain the source mapping for the current programme counter. If no source mapping is found, then the current code is compiler generated, and the plugin will attribute the gas cost of the current code to the last seen source mapping. Then, the plugin gets the difference in gas between the current machine state, and the gas last seen by the annotation (i.e. at the previous opcode). The plugin then updates the annotation by attributing the current gas difference to the source mapping, and updates the last seen gas of the annotation to the current

gas.

Algorithm 1: Hooks implemented for gas meter plugin

```
1  Gashook InstrHook(state):
2      a ← getAnnotation(state)
3      PC ← state[instruction]_address
4      sourceMapping ← getSourceMapping(PC)
5      if contract.hasSource(sourceMapping_id) then
6          a_key ← parseMapping(sourceMapping)
7      if a_key ≠ None then
8          diff ← getGasDiff(a_gasMeter, state)
9          updateGasMeter(a_gasMeter, a_key, diff)
10         setLastSeenGas(a, diff)
11
12 Prehook StopHook(state):
13     a ← getAnnotation(state)
14     foreach k_i ∈ a_gasMeter.keys() do
15         merge(globalGasMeter[k_i], a_gasMeter[k_i])
```

When a transaction ending opcode is encountered, such as `REVERT`, `STOP` or `RETURN`, the local gas meter is merged into the global gas meter for each source mapping executed in the current ending transaction, which persists and stores this information across multiple transaction instances. After the symbolic execution is complete, statistics for each code section such as the mean are then calculated, and returned as part of the result to the client.

4.3.2 Function Tracker Plugin

Mythril also has some native capabilities for tracking the current function that the execution is in. However, one challenge we faced when using this is that it only tracks the current active function, and therefore if function X calls another function Y of the same contract, and the transaction reverts in function Y, then the active function will still be function Y, and the worst case gas accumulated so far will be incorrectly attributed to function Y instead of function X. Instead, we want to know the entry function that is called at the start of the transaction, and attribute all gas accumulated throughout all possible paths (no matter if other functions are called as well) to that entry function.

The function tracker plugin was therefore created to more accurately track the worst case transaction gas estimate for each function that can be called externally via a transaction. This is then displayed to the user via a glyph decoration within the

Solidity editor, next to each function detected by the plugin. The plugin also makes clever use of the pattern observed in the EVM dispatcher routine to detect which function we have entered.

Therefore, to understand how the plugin works, we first need to understand how the EVM dispatcher identifies and jumps to the correct function when executing a transaction call. Each smart contract call transaction must contain calldata, of which the starting 4 bytes indicate the signature (or hash) of the function that the transaction wishes to execute, as seen in Listing 4.3. At the entrypoint of the contract bytecode, the dispatcher first loads the calldata in line 2 of Listing 4.4, and then pushes a possible signature for a function in the smart contract, in line 5. Then, it compares the newly pushed signature with that of the calldata using `EQ`, as seen in line 7, afterwhich it pushes the destination that the function resides in line 8, and jumps into it if the signatures are equal using `JUMPI` in line 9. This is then repeated for each function within the smart contract. If there are no matching functions found by the dispatcher, the transaction is reverted.

```
1 Transaction raw calldata:
2 0xa9059cbb // Function signature
3 00000....3f7614377 // Argument one
4 00000....11d800000 // Argument two
```

Listing 4.3: Transaction calldata

```
1    ...
2    CALLDATALOAD
3    DIV
4    AND
5    PUSH4 6FDDE03
6    DUP2
7    EQ
8    PUSH [tag] 2
9    JUMPI
10   DUP1
11   PUSH4 95EA7B3
12   EQ
13   PUSH [tag] 3
14   JUMPI
15   DUP1
16   PUSH4 18160DDD
17   ...
```

Listing 4.4: EVM dispatcher routine

We can thus make use of this pattern to identify which function the current trace entered at the beginning. However, another challenge is that since the calldata is symbolic, we cannot simply check its value for the function signature called, as this value does not yet exist. Instead, we designed a new algorithm for the plugin as

outlined by the pseudocode in Algorithm 2.

Algorithm 2: Hooks implemented for function tracker plugin

```
1  Prehook Push4Hook(state):
2      if state is not ContractCreationTransaction then
3          a ← getAnnotation(state)
4          PushValue ← state[instruction]_argument
5          if a_func None then
6              sig ← parseSig(PushValue)
7              if sig ∈ contract.signatures ∧ state[code][state[pc] + 1]_opcode = EQ then
8                  a_candidateFunc ← sig
9
10 Posthook JumpIHook(state):
11     if state is not ContractCreationTransaction then
12         a ← getAnnotation(state)
13         if a_func = None ∧ a_candidateFunc ≠ None then
14             didJumpIntoFunc ← state[prevPC] + 1 ≠ state[pc]
15             if didJumpIntoFunc then
16                 a_func ← a_candidateFunc
17             a_candidateFunc ← None
18
19 Prehook StopHook(state):
20     a ← getAnnotation(state)
21     if a_func ≠ None then
22         prevMax ← globalFuncMeter[a_func]
23         globalFuncMeter[a_func] ← max(prevMax, state[gas]_max)
```

First, for both the `JUMPI` and `PUSH4` hooks, we check if the current transaction is the contract creation transaction. Since we only want to collect data for the runtime transactions, we do nothing if this is true. Otherwise, we first look for a `PUSH4` instruction with an argument that matches one of the function signatures in the contract. We also check if the next instruction is an `EQ` opcode. If this matches, then it is likely that we are in the dispatcher routine, and we set the current candidate function to be this function signature.

After that, we look for the upcoming `JUMPI` instruction, as per the routine. Since we are using the post hook for this, the global state provided would be the resulting state after the `JUMPI` has already been executed. Therefore, we can check if the state has jumped into the function (the true case), or simply moved on to the next instruction in the dispatcher routine (the false case), by checking if the current PC

is equal to the previous PC plus one. In the true case, we simply mark the current function in the annotation as the candidate function (which we just jumped into). In the false case, we set the current function back to `None`, and repeat the process again for the next function signatures.

Once the current function has been identified, the plugin will not for this dispatcher routine anymore. This is to ensure that the original entry function (the one that was called in a transaction) will not be overriden by another function that might be called within the entry function.

When a transaction ending opcode is encountered, we then compare the total gas accumulated in the current global state, with the maximum gas encountered so far by the global function gas meter, for the current function. We finally store the new maximum of both gas counts for that function, and repeat this over multiple transactions. The final maximum gas counts for each function is then returned as part of the result to the client.

4.3.3 Loop Gas Meter Plugin

Next, we developed a loop gas meter plugin, aimed to keep track of per-iteration loop gas costs for each loop that was detected. The Mythril engine already has a bounded loops strategy, which is able to detect the current number of loop iterations, but we would like to keep track of the gas cost associated with this as well.

To do this, we make use of the following pseudocode in Algorithm 3, as follows.

First, we set up a pre-instruction hook for all opcodes. This checks if the current transaction is not a contract creation transaction, and then gets the current PC and source mapping of the PC. If the source mapping is not none, then we update the current source mapping for the annotation. This essentially updates the annotation with the latest source mapping reached at every point of the execution.

Next, we have a pre-hook for all `JUMPDEST` opcodes. These opcodes mark the start of every valid jump destination, which the loop body would start with. We first append the annotation trace (which keeps a log of the PCs of all `JUMPDEST` opcodes seen so far) with the current PC, and then call the $FindLoop$ function to try to find the PC of the current loop head, as well as the current loop iteration gas cost. We will explain in more detail about how this function works in the next section. If no loop is found, the loop head retuned is `None`. If a loop is found, we then check if the current loop is a hidden one (where it is generated by the compiler, and the source mapping does not exist). Finally, all of this information is added to the local loop gas meter within

the current annotation.

Algorithm 3: Hooks implemented for loop gas meter plugin

```
1  Prehook InstrHook(state):
2      if state is not ContractCreationTransaction then
3          a ← getAnnotation(state)
4          PC ← state[instruction]_address
           /* Try to find a source map for that PC                    */
5          sourceMapping ← getSourceMapping(PC)
6          if sourceMapping ≠ None then
               /* Source map exists, so PC is not compiler generated  */
7              a_key ← sourceMapping
8
9  Prehook JumpdestHook(state):
10     a ← getAnnotation(state)
11     PC ← state_instruction
12     a_trace.append(PC, state[gas]_max)
13     (foundPC, loopGas) ← FindLoop(a_trace)
14     if found ≠ None then
15         loopSourceMap ← getSourceMapping(foundPC)
16         isHidden ← isGeneratedCode(loopSourceMap)
17         AddToGasMeter(a_key, foundPC, loopGas, isHidden)
18
19 Prehook StopHook(state):
20     a ← getAnnotation(state)
21     merge(globalGasMeter, a_gasMeter)
```

Lastly, we have a pre-hook for all terminating opcodes. Once these opcodes are reached, we get the current annotation (and hence local loop gas meter) from the global state, and merge it with the global loop gas meter stored in the plugin. The average iteration gas cost for each loop head found is then taken, and returned to the user as the result.

Now, we discuss more about how the *FindLoop* function works, which is the main challenge faced when implementing this plugin. Unlike other loop finding algorithms that work on linked lists, our case is unique in that this linked list of the execution trace is being constructed while we try to detect a loop. As such, methods such as the Floyd's Cycle Detection Algorithm will not work, because the "fast" pointer will not know what paths to proceed towards, as they have not been executed yet and added to the trace. Therefore, we have to make use of a pattern matching algorithm that starts from the end of the trace and proceeds backwards, in

order to detect a possible loop. An outline of its pseudocode is provided in Algorithm 4.

Algorithm 4: $FindLoop$ algorithm

Input: List of PCs jumped to before, in $trace$
Output: PC of start of detected loop, and the loop iteration gas

```
found ← None
i ← 0
l ← len(trace)
foreach trace item t_j ∈ trace do
    i ← j
    if t_i.pc = t_{l-1}.pc ∧ t_{i+1}.pc = t_{l-2}.pc then
        found ← t_i
        break
if found ≠ None then
    loopGas ← t_{l-2}.gas − found.gas
    /* Remove trace items to prevent this from finding it again    */
    foreach trace item t_k ∈ trace[i − 1 : l − 3] do
        delete t_k
    return found.pc, loopGas
else
    return None, 0
```

Essentially, the function works by finding a match of the latest two consecutive `JUMPDEST` instructions from within the PC history stored in the trace. If this is found, we set as former of the latest two consecutive instructions as the loop head. Overall, the algorithm is O(n) in time complexity since it needs to possibly traverse the entire trace, and O(n) in space complexity as well, since the entire trace has to be stored within the annotation.

This algorithm is not 100% accurate however, and we may find false positives. For example, if two sections of a function both call the same helper function, this may register as a loop even though the helper function is not recursive. However, we chose to use this heuristic over other static analysis methods (on the EVM bytecode or Solidity code) because it is able to not only capture possible "hidden" loops generated by the compiler, it is also able to capture any recursive loops seen during each symbolic execution. Also, from our testing, the rate of false positives is not high, and can be trivially ignored by the developer.

To illustrate this, let us examine Figure 4.4. Here, we begin at PC 241, which is the first PC to be added to the previously empty trace. We next jump to PC 388, which for this example is the loop head containing the break condition. This is also

appended to the trace. Next, we jump to PC 411, the loop body, and append it to the trace as well. This takes us back to PC 388, which is appended to the trace again. Currently, the trace contains $\{241, 388, 411, 388\}$, which suggests there might be a loop, but because no two consecutive PCs are found yet, the algorithm does not determine a loop exists so far. We now move to PC 411 again in the second iteration of the loop, and now the trace contains $\{241, 388, 411, 388, 411\}$. Here, the algorithm finally finds a match, and correctly determines a loop starting at PC 388. It then calculates the gas difference between the first and second instances of arriving at PC 388, which is the gas cost of one iteration, as seen in line 10 of the pseudocode.

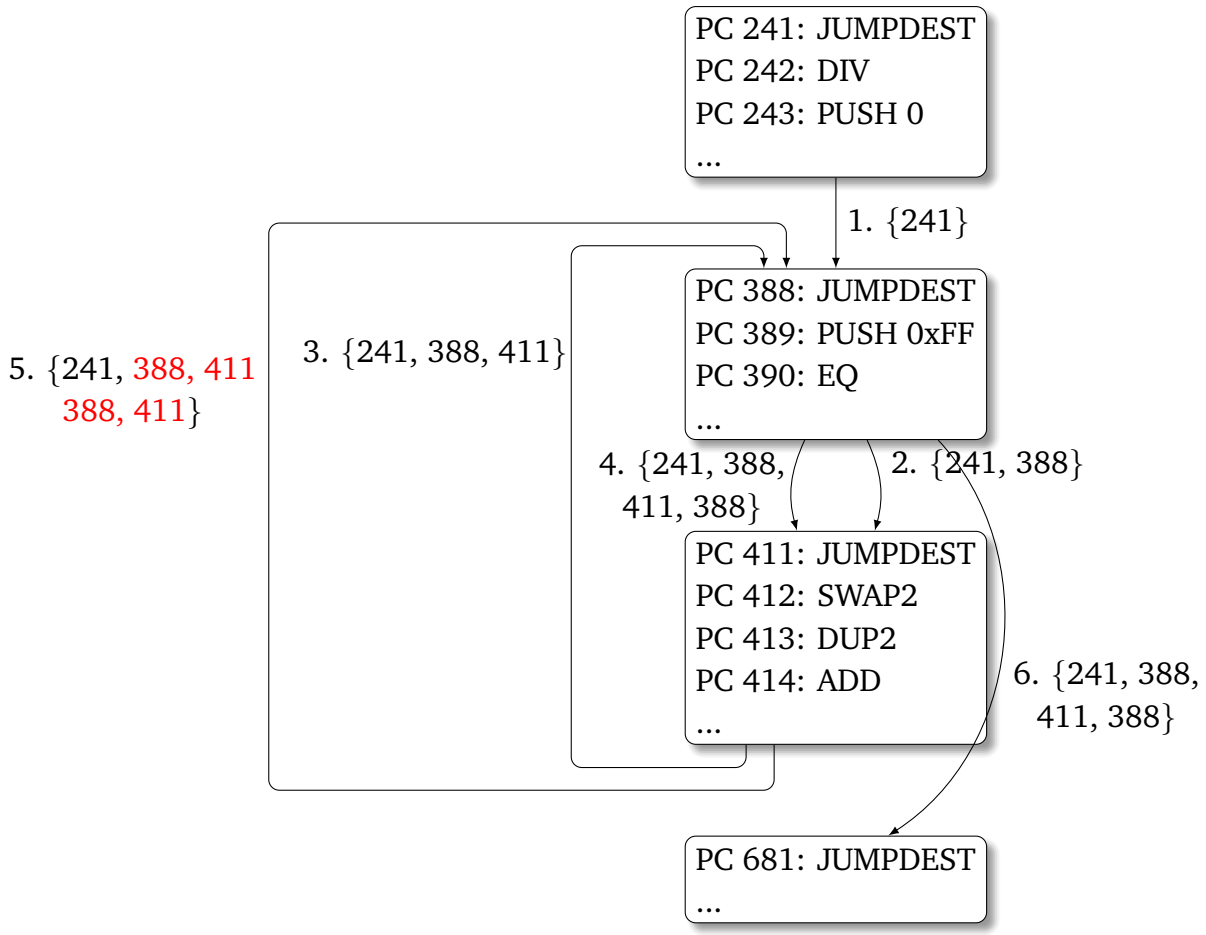


Figure 4.4: Example of loop finding algorithm

After this, we delete all trace items seen in the interval between the previous loop head and the current PC, as seen in lines 11 to 12. In our example, in step 6, we can see that the second occurence of $\{388, 411\}$ has been removed. This is to prevent any double counting of loop heads. For example, if no trace iterms were deleted, the trace at step 6 would instead be $\{241, 388, 411, 388, 411, 388\}$, and the algorithm would detect another loop at PC 411. However, this "loop" was already accounted for in the loop starting at PC 388. At the end of the algorithm, the PC of the loop head and the iteration gas cost are then returned in line 13, to be used within the

JUMPDEST hook.

4.3.4 Loop Mutation Detector Plugin

Finally, we also developed a loop mutation detector, which checks if there are any SLOAD or SSTORE opcodes found within a loop. This is because one of the common gas-hungry anti-patterns is unnecessarily modifying storage variables within the loop body, when you can instead use a temporary local copy of the storage variable within the loop, and then update the actual storage variable just once when the loop is complete. This example is evaluated in more detail in Section 5.2.

To implement this plugin, we set up an instruction hook for `SSTORE` and `SLOAD` instructions, and simply piggy-backed off the `JumpdestCountAnnotation` used by the bounded loops strategy te determine if we are in a loop or not. If we are, then the plugin looks up the source mapping for the current PC, and then adds this to the dictionary storing the set of offending source mappings.

The plugin is not always accurate, and discretion is required from the developer to determine whether the code can be refactored to fix this. For example, in the case of updating dynamically-sized variable types, such as `string` and `bytes`, the compiler automatically generates a routine that involves iterating through and updating the relevant storage slots in a loop. This will be flagged by the plugin, but cannot be fixed or refactored, firstly because the offending code is compiler generated, and secondly because this actually requires the use of a loop to iterate through the different slots.

4.3.5 Removal of Z3 SMT Constraints

While testing the performance of the symbolic execution engine, we noticed that the first few transactions were completed very quickly, but the next transactions took exponentially longer times to symbolically execute. This may be because the state space grows exponentially with the number of transactions, but we also suspected that because the list of constraints for each state also grows exponentially, this makes solving them much slower. In addition, we faced with the challenge where some states were not being executed despite being reachable, because the Z3 solver was timing out from solving constraints that were too complex, which resulted in it simply pruning the state altogether. This caused very poor performance for code coverage.

Security verification requires the fact that unreachable states are pruned and not executed, because they are focused on which paths are reachable and which are not. However, for our purpose of gas estimation, this is not as important when compared to faster execution times and greater coverage. Being able to quickly calculate the gas usage of a code section when it is reached is more important to us

than knowing exactly if that particular section is reachable or not. As such, although useful for pruning away states when the list of constraints is relatively small, the cost of running this Z3 solver quickly outweighs its benefits when the list of constraints starts to grow exponentially.

Therefore, we provide the user with a setting that disables the Z3 constraint check, and simply runs every possible execution path regardless of whether they are reachable or not. This is turned on by default, and in our testing, provides much better execution coverage with for the same execution time.

5

Evaluation

In order to evaluate the quality and effectiveness of the SOLBOLT, 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. **Accuracy**, 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

To achieve this, we performed a systematic test over a dataset of more than 1000 on-chain Ethereum smart contracts, collecting key metrics and comparing these with results from previous literature. We also performed a case study to illustrate the potential utility of this tool, as explored in the next few sections.

5.1 Study on coverage and estimation accuracy

The goal of this study is to provide some insight over points 1, 2 and 3 of our evaluation criteria — the speed, coverage and accuracy of our tool. To do this, we ran our extended Mythril symbolic execution engine over a dataset containing more than 160,000 verified smart contracts, obtained from Etherscan [53]. However, due to time constraints, we chose to analyse the top 1,392 smart contracts with the most number of transactions. This also allowed the analysed contracts to have a more statistically significant number of concrete transactions to compare our estimates against, which should allow for lower variance in our results.

5.1.1 Testing pipeline

First, we obtained the entire index of smart contracts [53], and sorted it by descending number of transactions. Next, for each smart contract, we obtained its verified source code using the Etherscan API, and compiled it using the same compiler settings as the deployed onchain code. We then ran a symbolic execution instance with the compilation result, using the default settings as shown in Appendix A. Each instance is also ran with a symbolic execution timeout of 120 seconds.

Next, after obtaining the symbolic execution results for each contract, we go through and compare the latest 10,000 transactions, or until 50 valid transactions are obtained, whichever is later. For each concrete transaction, we extract the function being called from the calldata, as well as the actual gas used by the transaction. We then, using the estimates from the symbolic execution, calculate the accuracy of the estimation by dividing it with the concrete gas cost. This means that an accuracy of 100% will mean the symbolic execution exactly predicted the concrete gas cost, an accuracy of 125% will mean the symbolic execution over-estimated by 25%, and an accuracy of 75% will mean the symbolic execution under-estimated by 25%. We also calculate the "gas class" of that transaction, which sorts the transaction into 9 classes based on its concrete gas cost. The accuracy for each transaction is then stored, together with the function it called and its gas class. Afterwards, this routine is repeated for each transaction, and for each smart contract. To prevent commonly called functions from skewing the test set, we also limit the maximum number of analysed transactions for each function to be 30, after which transactions for that function will be ignored.

Finally, the accuracy results are then collated and summarised, by calculating their sum, mean, median and counts. A summary is also calculated for each function and gas class. These gas estimation accuracy results, together with the coverage and symbolic execution results, are then stored within a text file, and will be analysed later. If there are any errors raised during the entire testing pipeline, it is also logged and stored within the text file.

5.1.2 Analysis methodology

After running the testing pipeline on a sufficient number of smart contracts, we then analysed the results using several methods. First, we took the mean gas estimation accuracy and coverage achieved for each contract, and plotted a violin graph representing the distribution of data across all smart contracts. We also plotted a histogram of all functions grouped by their their mean accuracy of predicted gas over concrete gas, and compared this with results from previous literature.

Next, for all smart contracts analysed, we again plotted a violin graph of mean gas estimation accuracy, but instead grouped them based on the coverage achieved, so

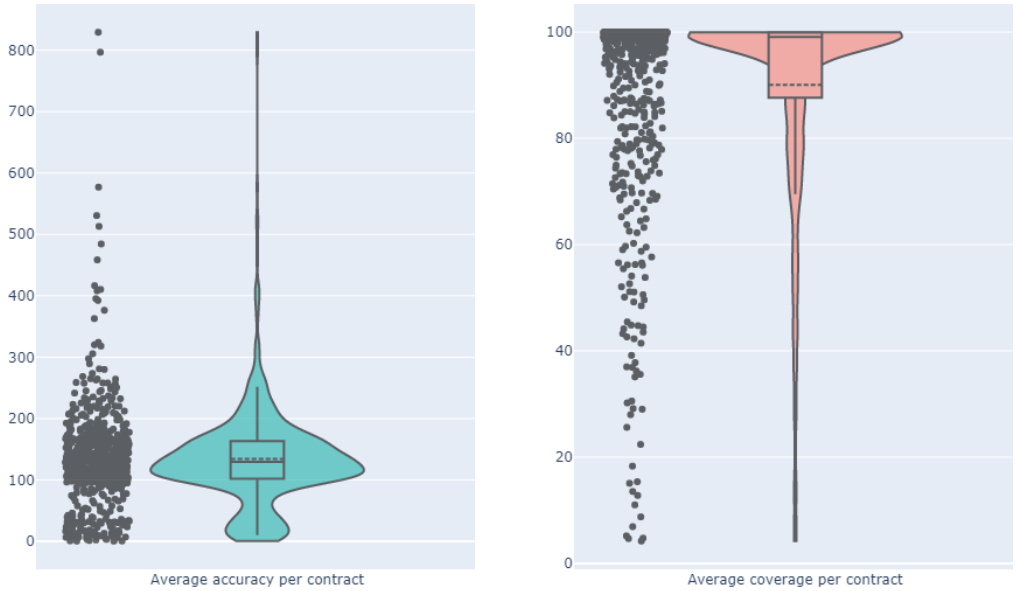
as to observe any correlations between the two. Similarly, this was also done for each transaction, plotting the mean gas estimation accuracy as a violin graph but grouping them based on their "gas class" as mentioned before.

Finally, we plotted a histogram of the number of counts of successful executions, together with the counts of various errors encountered, to understand the robustness of the tool under real-world usage.

5.1.3 Results

The entire evaluation pipeline was run over 51.5 hours on an Intel i7-9700K machine with 32GB DDR4 RAM, which was running Ubuntu on Windows Subsystem for Linux, and a total of 1,392 contracts were evaluated. Of which, 754 contracts were successfully symbolically executed and analysed with their concrete transactions, which evalutes to around 4 minutes for each contract. Out of these contracts, 698 contracts had enough valid concrete transactions for us to conduct our gas estimation accuracy analysis on, the results of which we shall present in this section.

Overall coverage and accuracy



(a) Overall gas estimation accuracy (%)

(b) Overall symbolic execution coverage (%)

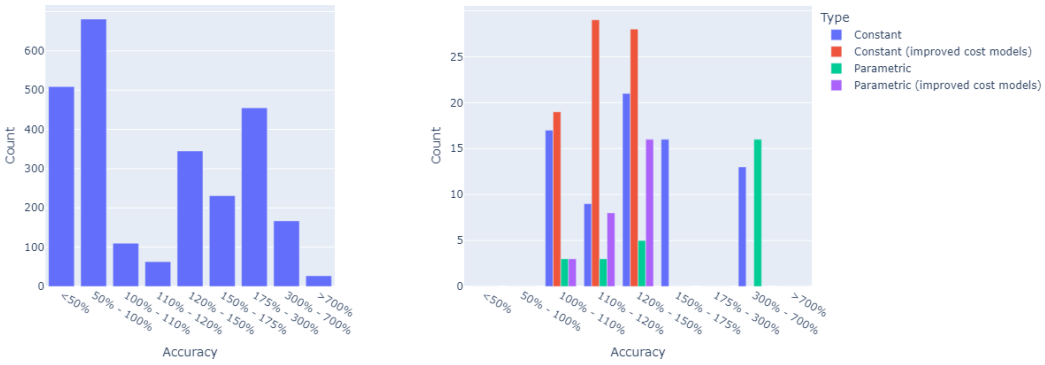
Figure 5.1: Overall results for coverage and accuracy

Overall, across all the analysed smart contracts, the symbolic executor managed to achieve a mean gas estimation accuracy of 134.4%, and a median gas estimation accuracy of 129.3%. This is expected, since we are estimating the worst case gas

cost of each function, which will generally be an over-estimate of the concrete gas cost. Examining Figure 5.1a, we see that the estimation accuracy mostly fall between the 80% to 200% range, with the 25th percentile being 102.1% and the 75th percentile being 163.5%. We also see very few contracts being severely over-estimated, although there is a significant density of contracts for which the symbolic executor heavily under-estimates worst case gas costs. This all suggests that the gas estimation of the symbolic executor is reasonably accurate, and we believe that the cause for the under-estimation of some contracts is likely due to the lack of coverage.

In addition, the symbolic executor also manages to achieve a mean coverage of 90.1%, and a median coverage of 99.1%. The 25th percentile achieved for this is 87.6%, while the 75th percentile is 99.9%. This is an impressive result that showcases the performance of our tool, since the symbolic executor is still able to cover most code paths despite facing a limit of 2 symbolic transactions and the execution timeout of 120 seconds. This is also much better than the 88% average and 94% median coverage achieved by VisualGas, which uses fuzzing and dynamic concrete executions instead of symbolic execution as mentioned in Section 3.2, while executing within a similar timeframe. These results also confirm the hypothesis that symbolic execution, which analyses a programme path-by-path instead of input-by-input, can be more efficient and achieve greater coverage for smart contracts, which generally have a large input statespace, but a relatively small executional statespace in terms of the range of executed opcodes.

Histogram of estimation accuracy per function



(a) Histogram of estimation accuracy for SOLBOLT

(b) Histogram of estimation accuracy for GASTAP

Figure 5.2: Estimation accuracy for each function

Next, for each function evaluated, we calculated the mean gas estimation accuracy across all transactions, and binned them into 9 different classes based on their accuracy. These classes are chosen to match those evaluated in GASTAP [48], which we

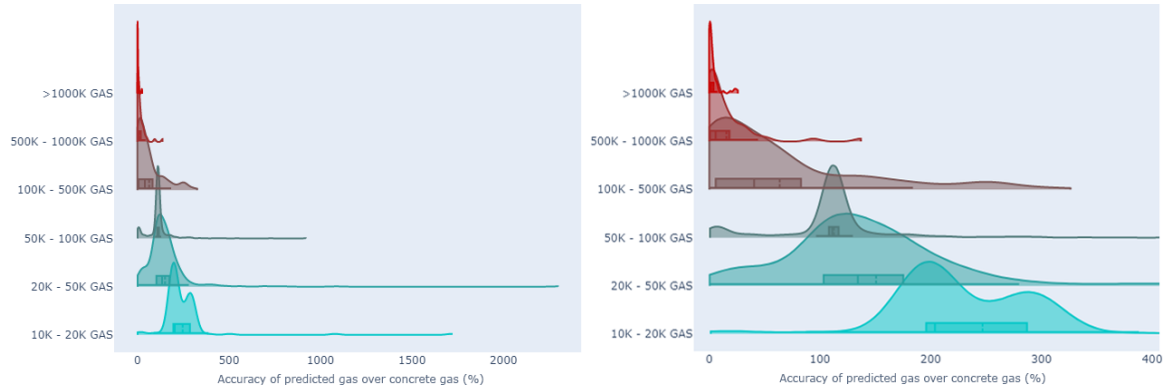
then compared our results against, as shown in Figure 5.2. Unfortunately, as GASTAP is closed-sourced and unavailable for public testing, we were unable to run the same tests under the same conditions as the ones we performed for SOLBOLT. As such, the published results from the paper were used for comparison instead, which we were also unable to verify for ourselves.

First, we notice that a far greater number of functions were tested for SOLBOLT (over 2500 functions) compared to GASTAP (only 125 functions). This may potentially lead to larger possible variability in results for GASTAP, especially if repeated on a similar larger set of contracts. Next, for a significant proportion of functions, SOLBOLT still underestimates the gas costs required for running them. This can be seen from the bars with estimation accuracy of less than 100%, as seen in Figure 5.2a. This is likely because of not having enough coverage for these functions, possibly due to the max depth being exceeded when executing the path, or hitting the execution timeout. On the other hand, GASTAP seemed to consistently provide a strict upper bound on gas costs, with zero of the tested functions recording a concrete gas usage greater than that which was estimated. This is to be expected, since GASTAP makes use of static analysis and reasoning over the generated rule-based representations of the bytecode, it is able to produce an over-estimate of the gas cost that is guaranteed to be greater than the worst case gas usage. In addition, the gas bounds generated by GASTAP can be parameterised by the arguments given, while SOLBOLT simply executes the contract until the max depth or loop bounds are hit. This means that for example, if a loop in a smart contract function runs 10 times, the GASTAP estimate can take this into account and calculate an updated worst case gas cost for this. However, if the loop bound limit set for analysing the same contract in SOLBOLT was only 5, then SOLBOLT will likely under-estimate the gas for this function.

For the functions that SOLBOLT and GASTAP over-estimate gas costs for, we see that in SOLBOLT, the functions are mostly over-estimated by 130% to 300%, while in GASTAP, the functions are mostly over-estimated from 100% to 150%, when comparing its improved cost models. This suggests that GASTAP still performs better at gas estimation than SOLBOLT, as it is able to provide a stricter and tighter gas upper bound on a transactional level. However, we believe that SOLBOLT still performed reasonably well even when compared to the state-of-the-art of gas bounds estimation. In addition, SOLBOLT is capable of tracking gas usage on a per code section basis, rather than simply on the per-function basis, which provides some additional insight for the developer that GASTAP does not.

Estimation accuracy by concrete gas usage

Next, we also binned each transaction into 6 “gas classes”, based on the concrete amount of gas they used in that transaction. We then plotted the estimation accuracy of each transaction, grouped by each gas class. This is shown in Figure 5.3a, with a



(a) Full plot **(b)** Zoomed in

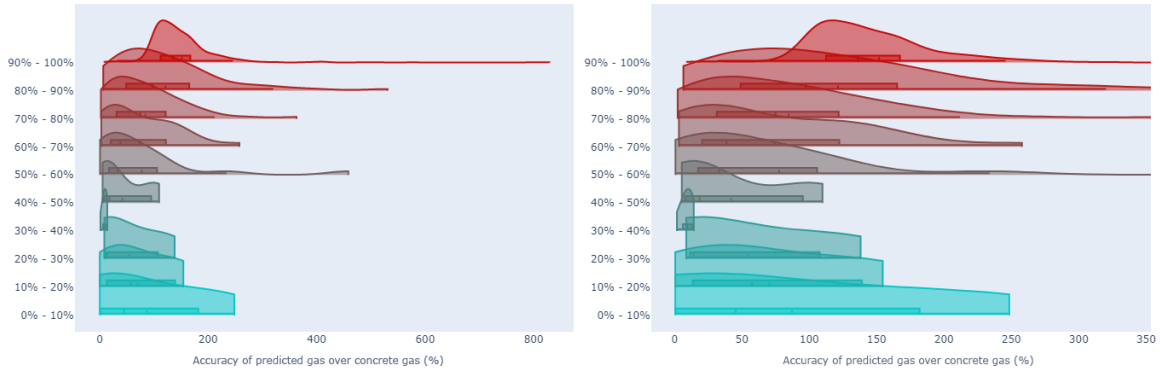
Figure 5.3: Estimation accuracy for each gas class

zoomed in view shown in Figure 5.3b. Here, we observe a trend where the higher the concrete gas cost is, the lower the percentage of the estimated gas cost over the concrete cost is. This may be directly because of the max depth and loop bounds limit imposed on the execution. For transactions that have very high gas usage, it is likely that they may be running many iterations of a loop, far greater than the loop bounds set in the symbolic execution. As such, the gas estimate generated by SOLBOLT will also likely be an underestimate. For transactions with very little gas usage, the inverse may be true. They may be taking paths through the function that require less gas, or possibly run very few iterations through a loop. This makes it likely for SOLBOLT to over-estimate the gas used.

We can clearly see that precisely estimating the concrete gas used by every transaction is not a trivial task, as each this depends on the exact state of the contract, as well as the parameters given. However, the results also show that the median gas estimation accuracy for gas classes ranging from 10,000 gas units to 500,000 gas units, which cover the vast majority (94.4%) of all transactions tested, are still relatively reasonable, ranging from an mean of 40.6% to 204.1% of the concrete gas costs. We believe that this makes the gas estimation from SOLBOLT still insightful for developers. Developers can also tweak the limits of the symbolic executor to match what is expected from their functions, which can potentially produce more accurate estimates.

Estimation accuracy by coverage

We again plotted a similar graph of estimation accuracy, this time grouped via 10 coverage classes instead, sorted based on the overall coverage achieved for each contract. The estimation accuracy is also taken from the mean estimation accuracy of all transactions for each contract, rather than the per transaction accuracy as before. The graph is shown in Figure 5.4a, with a zoomed in view shown in Figure



(a) Full plot **(b)** Zoomed in

Figure 5.4: Estimation accuracy for each coverage class

5.4b.

Here, we observe the trend that where the lower the coverage is, the lower the percentage of the estimated gas cost over the concrete cost is. This is to be expected, given that the gas estimation depends entirely on how much of the contract is executed. This also adds support to our theory that the lack of coverage is the main reason why SOLBOLT underestimates gas costs for certain contracts. This lack of coverage could occur for several reasons, such as if the max depth or call limit is hit when executing a certain path. If the Z3 solver was enabled, this could also happen because the solver timeout was reached, or if the solver determines a certain code path was unreachable.

We also see that for the class containing 90% to 100% coverage (which covers 73.7% of all contracts), the median gas estimation accuracy is 136.2%. This implies that SOLBOLT does tend to produce an over-estimate of the gas cost, if the contract is well-covered. For the classes containing 80% to 90% and 70% to 80% coverage (covering a total of 14.9% of all contracts), the median gas estimation accuracy drops to 97.5% and 74.9% respectively. This is likely because although SOLBOLT generally over-estimates the gas cost, the lower coverage offsets this effect, resulting in a slight under-estimate of gas costs instead. Similarly, this test was run with the same symbolic execution settings for all contracts, and users are encouraged to tweak the settings to the requirements of their own contracts.

Histogram of success and error rates

Finally, we plotted a graph of the final evaluation status — whether it was successful, or which error was caught while evaluating — for each of the 1,392 contracts tested, as shown in Figure 5.5. As shown in the graph, 698 contracts were successfully evaluated, while 694 contracts caught an error during the evaluation pipeline. Out of this, the vast majority of errors (487) were because the Solidity compiler ver-

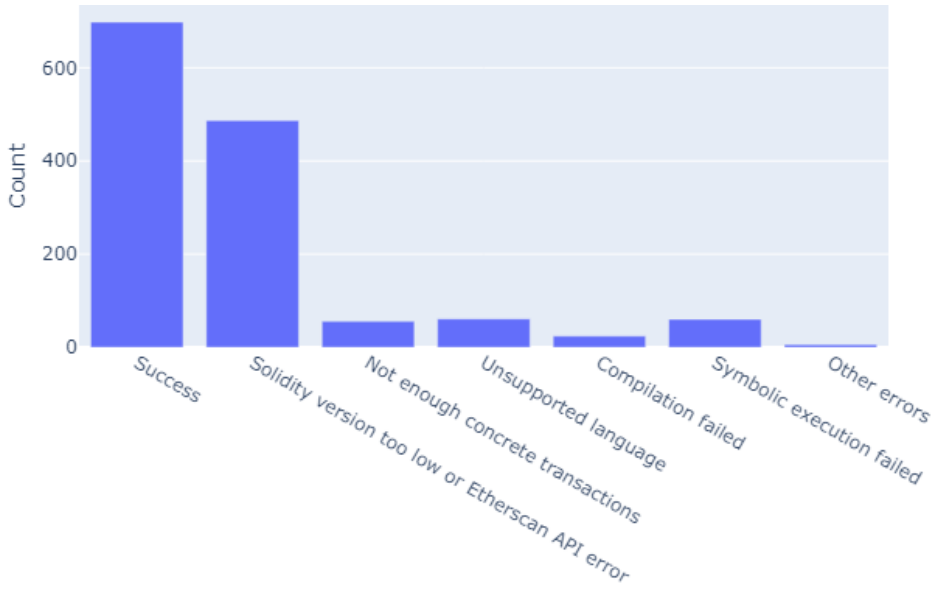


Figure 5.5: Evaluation status

sions of the smart contracts were lower than version 0.4.17. These compilers were not supported, because they still lacked the functionality to produce an abstract syntax tree, which was required for the symbolic execution. In addition, 56 contracts were discarded for having not enough concrete transactions (and hence datapoints) for evaluation, 61 contracts were discarded for not being written in Solidity, and 6 contracts were caught with other errors.

This left a remaining 24 contracts faced with compilation errors, and 60 contracts faced with a symbolic execution error. For the compilation errors, these were mainly due to some contracts manually linking certain libraries, which is not yet supported by SOLBOLT, or because of import errors. For the symbolic execution errors, these occured primarily due to the contract creation transaction not being able to deploy the contract and construct the initial open states. This could in turn be because the max depth was hit during the constructor of the contract, or the executor timing out.

Only counting the errors faced with symbolic execution and compilation, we have an overall evaluation success rate of 89.3%. While this certainly could be improved upon, we believe that the tool is still very reliable for general usage. In addition, many of the errors are from testing on older smart contracts, whereas the tool was designed with the newer features of more recent compiler versions in mind. As such, while the tool still mostly supports the wide range of compiler versions we have available, contracts written with more recent versions of Solidity are much less likely to face problems while using the tool.

5.2 Case Study: Gas-hungry loops

Next, we present a case study on three contracts, to investigate how a user might make use of SOLBOLT to optimise their code. The goal of this study is to provide some insight to the utility of this tool, and how a developer might use the available information to optimise a smart contract.

5.2.1 A really gas-hungry loop

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.7.0 <0.9.0;
3  contract GasHungryLoop {
4      mapping (uint256 => address) private tokenToOwner;
5      uint256 indexToMint;
6
7      function mint(uint256 numberToMint) external {
8        for (uint256 i = 0; i < numberToMint; i++) {
9          tokenToOwner[indexToMint++] = msg.sender;
10       }
11     }
12 }
```

Listing 5.1: A really gas-hungry loop contract

Let us examine the above smart contract, as shown in Listing 5.1. This contract is a stripped down version of a standard ERC-721 non-fungible token (NFT) contract, where only a mapping from the token ID to the owner address is stored. The other functions typically required for an ERC-721 token, such as `transferFrom`, `approve` and `tokenURI`, are not included here for brevity. Here, the mint function will "mint" a certain number of NFTs, by iterating through the loop `numberToMint` times, and each time setting the owner of the token ID to the sender of the transaction. After each iteration, `indexToMint` will be incremented and stored, so the contract knows what is the next token ID it should assign.

This contract is written in a highly concise way, similar to how a non-blockchain programme might be written. However, after running the symbolic execution (with settings listed in Appendix A), we notice a few problems, as shown in Figure 5.6.

Here, we see that the gas estimate for the `mint` function (assuming a loop bounds of 5) is 261,003 gas units, which is relatively high for a single transaction. We also see that the symbolic execution engine detected two loops from lines 8 to 10. The first loop, with a per-iteration gas cost of 52,143 gas units, is likely to be the `for` loop written within these lines. This is in line with the overall function gas estimate when multiplied by the loop bounds (five times of the per-iteration gas cost would be 260,715, slightly below the estimated function gas cost), as most of the gas would be consumed by the loop (which makes up majority of the function). The second loop, with a per-iteration gas cost of 51,209 gas units, is a hidden loop that is executed

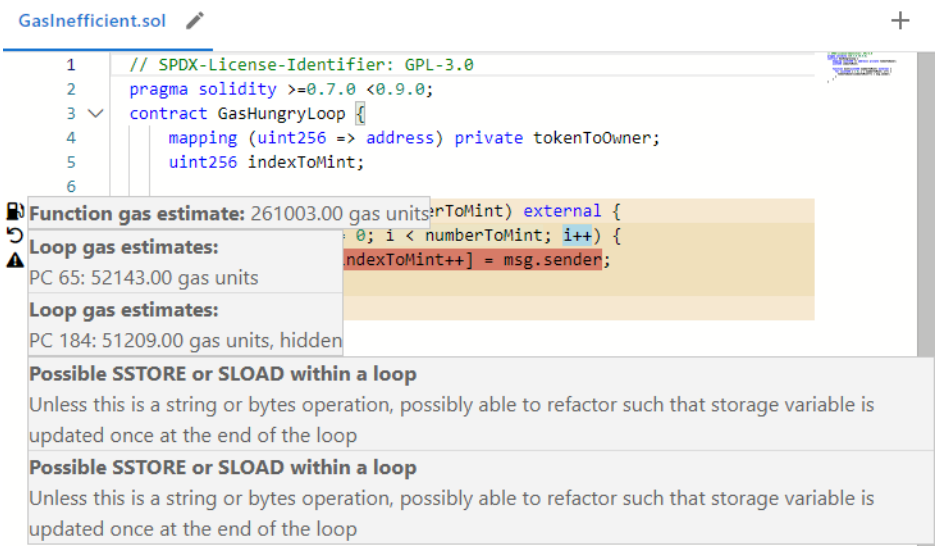


Figure 5.6: Gas estimates and issues detected by SOLBOLT

within the for loop, and is likely generated by the compiler because of the access of a mapping structure here.

We also see that there are two warnings for a possible SSTORE or SLOAD within a loop, on line 9. This suggests that it may be possible to refactor the code such that storage variables are not accessed on each iteration, but rather updated once at the end of the loop by making use of temporary local variables. This is precisely the case here, since we both read from and increment the `indexToMint` variable, which is a storage variable, at each loop iteration.

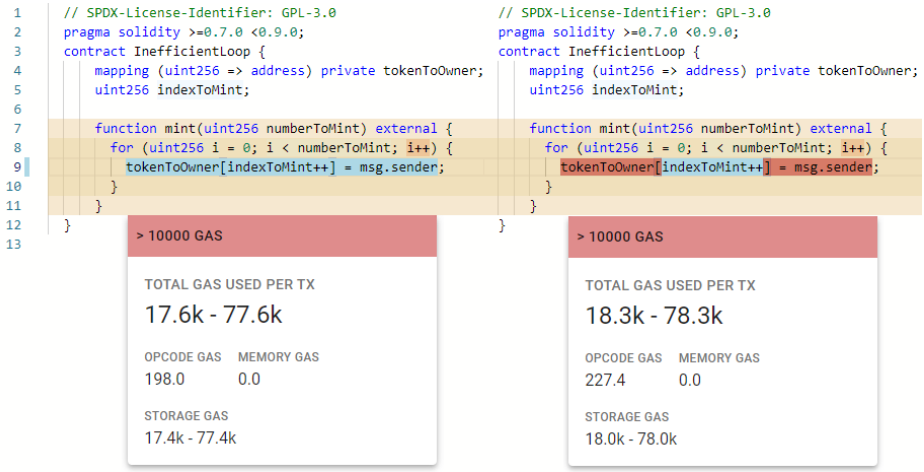


Figure 5.7: Two code sections with very high estimated gas costs

Hovering over them to see their precise gas estimates for each code section as seen in Figure 5.7, we also see that the two storage updates to the `tokenToOwner` mapping and `indexToMint` integer can take over 77k gas units, which makes up the majority

of all gas consumed in the loop. This suggests to the developer that optimising these storage updates could provide a significant gas saving.

Indeed, one idea could be to make use of the `i` local variable during each loop iteration for indexing the mapping, and then increment the `indexToMint` variable with `i` at the end. Since each transaction on the EVM is executed sequentially, there is no concern for race conditions with other contract interactions sent at the same time.

5.2.2 A more optimised loop

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.7.0 <0.9.0;
3  contract SlightlyBetterLoop {
4      mapping (uint256 => address) private tokenToOwner;
5      uint256 indexToMint;
6
7      function mint(uint256 numberToMint) external {
8        for (uint256 i = 0; i < numberToMint; i++) {
9          tokenToOwner[indexToMint + i] = msg.sender;
10       }
11       indexToMint += numberToMint;
12     }
13 }
```

Listing 5.2: A more optimised loop contract

Using the approach outlined before, we have this newer, more optimised contract, as shown in Listing 5.2. Running the symbolic execution again gives the following result, as shown in Figure 5.8.

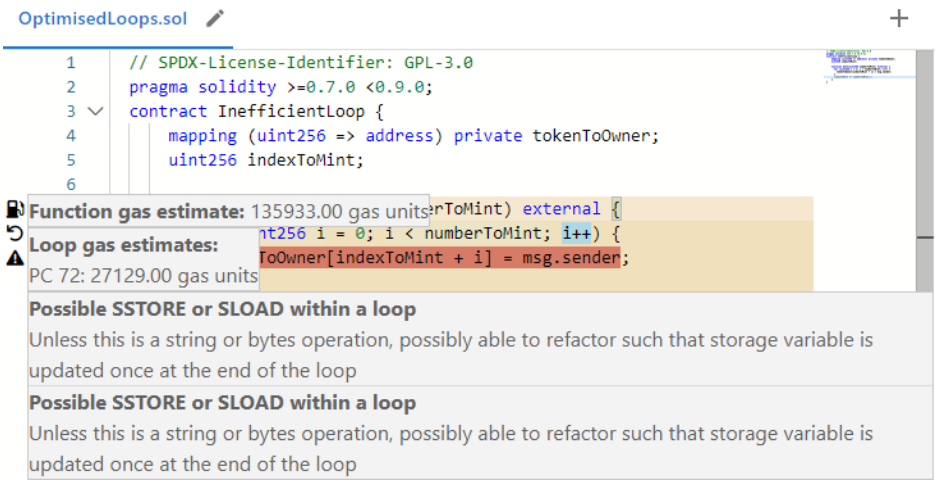


Figure 5.8: Gas estimates and issues for more optimised loop

Here, we see that now, the function gas estimate has dropped to just 135,933 gas units, which is a improvement of 47.9% over the non-optimised contract. The per-iteration gas cost has also improved by a similar amount to just 27,129 gas units.

Examining the gas estimates of the same two costly sections identified previously, as displayed in Figure 5.9, we see that the gas cost for calculating the index of the mapping has reduced greatly to just 266 gas units. The gas estimate for updating the mapping itself has not changed, as it cannot be refactored, due to it being a necessary operation of this function.

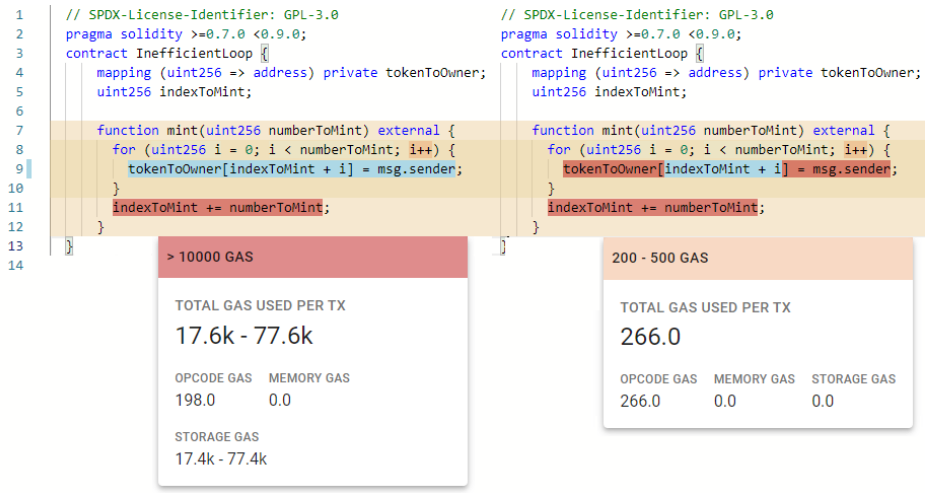


Figure 5.9: Optimised costs for the two gas-hungry code sections identified

However, we still see that there are two warnings for the use of `SSTORE` or `SLOAD` within the loop. This suggests that there may still be further gas savings achievable. Indeed, we are still reading the `indexToMint` storage variable within the loop, albeit no longer updating it.

The most optimised solution to this would therefore be to store a temporary copy of the `indexToMint` variable, and then simply working with this within the loop, before finally updating the actual storage variable at the end of the loop, as seen in Listing 5.3. However, this gas saving comes at a cost of verbosity and readability, which the developer needs to carefully balance while creating a contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract InefficientLoop {
    mapping (uint256 => address) private tokenToOwner;
    uint256 indexToMint;

    function mint(uint256 numberToMint) external {
      uint256 i = indexToMint;
      uint256 limit = i + numberToMint;
      for (; i < limit; i++) {
        tokenToOwner[i] = msg.sender;
      }
      indexToMint += numberToMint;
    }
}
```

Listing 5.3: The most optimised loop contract

5.2.3 Other examples

Through this case study, we have successfully demonstrated how in this particular case, a developer can make use of the gas insights provided to optimise a gas-inefficient loop pattern. In addition, we also provide 3 other examples for users to explore, illustrating issues such as the ordering of storage variables, the use of recursion, and the effect of public functions on the dispatcher routine. These are all available within the "Examples" tab on the live website.

We believe that these are a useful starting point for developers learning to use SOLBOLT, allowing them to better make use of the detailed gas cost breakdown and potential issues provided by SOLBOLT to identify problems in their own smart contracts.

6

Conclusion

As the DeFi and NFT space continues to grow in the coming years, concerns over high gas prices required to perform any transactions within this space will likely continue. While scaling decentralised blockchains such as Ethereum remains an open problem being worked on, developers today can help partially address this by writing better, more gas-efficient smart contracts.

As such, SOLBOLT is designed particularly for this purpose, to help developers better pinpoint exactly which parts of their code is using up the most amount of gas, and to more easily identify gas-inefficient patterns.

6.1 Summary of Achievements

Currently, the majority of Solidity analysis tools available on the market are focused on security verification of smart contracts, and often neglect another important part of the development pipeline — gas optimisation. With SOLBOLT, the compiler-explorer and gas analysis tool we have built, we believe that we have successfully achieved our objective of creating a tool that helps developers better understand the compiled bytecode and gas profile of their contract, and addressed some of the market gaps within this part of the Solidity development ecosystem, as summarised:

1. **Intuitive and powerful user interface of compiled EVM bytecode and their source maps**

   We have successfully built a simple-to-use and intuitive interface for viewing the compiled EVM bytecode and their source maps, inspired by the tool Godbolt. The frontend is built using React Typescript and served using AWS Amplify, while the backend made use of a Celery task queue for the dispatching of jobs, as well as Flask for serving the API endpoints. The hover function makes it easy to identify which code sections relate to which EVM bytecodes, and

the Etherscan API integration allows for easy analysis of any verified Ethereum smart contracts. In addition, the tool allows for the compilation of multi-file, multi-contract source codes, from a wide range of supported Solidity compilers. Finally, the customisable compiler settings also exposes a detailed range of optimisation options.

To date, we know of no other tool that provides a similar user experience when analysing the EVM bytecode. We believe that this function can be very useful for researchers and developers, who are looking to more closely examine their deployed bytecode, or wish to tweak some of the more obscure optimisation options.

2. **Accurate and detailed analysis of gas consumption**

   Our extended Mythril symbolic execution engine can accurately estimate the gas consumption of most contracts within reasonable margins, given sufficient coverage. This can be performed at the code section level, using the gas meter and gas hooks which we have built specifically for this task. Gas estimation can also be done at the functional level and the per loop iteration level, by developing a function tracker plugin which made use of the EVM dispatcher routine to infer the function that was taken by each symbolic execution path, as well as a loop gas meter plugin which stores the current executional trace and is able to detect if we are currently in a loop. These features provide detailed insight about the gas profile of the smart contract and where possible savings may be, which to our knowledge, no other tool is able to generate with the same level of accuracy and fineness.

   The project also proves the efficiency of symbolic execution for not just security analysis, but also gas estimation.

3. **Automatic detection of potential issues for further gas savings**

   SOLBOLT is also able to automatically detect common gas-inefficient patterns, such as modifying storage variables within a loop, to allow developers to quickly pinpoint where potential gas savings can be achieved. This has shown to be useful and accurate in the case study we have examined.

6.2 Limitations

While we believe that SOLBOLT is ready for general development use, we also acknowledge some of the limitations that SOLBOLT currently has. First, when compared to some state-of-that-art gas estimation tools such as GASTAP, SOLBOLT still is not able to guarantee a worst-case gas bound, and is unable to generate paramteric bounds depending on function arguments. SOLBOLT may also at times fail to

work when using advanced features such as manual linking of libraries, or the use of oracles.

In addition, while the original scope of the work included compilation and mapping of the Yul intermediate representation, this has been sidelined by the creation of more detailed plugins for analysis on the EVM bytecode instead. However, we believe that once Yul matures from its experimental state, such a feature could be very useful in the future. This is especially so given the planned usage of EWASM over EVM for the future release of Ethereum, where Yul will continue to act as an intermediate language.

6.3 Future work

SOLBOLT is still a work in progress and improvements to the user interface and the symbolic engine can definitely still be made. Some possible ideas that come to mind are laid out as follows:

1. **Support for Yul IR code mappings** The code mappings for the Yul IR can be added to the SOLBOLT pipeline, by first storing the Solidity code maps generated for the Yul IR, and then compiling the Yul IR into the EVM bytecode and storing the corresponding Yul code maps. The Yul code maps would then need to be merged with the Solidity code maps in order to produce a global code mapping. This would however only be supported for more recent versions of the Solidity compiler that support Yul compilation.
2. **Integration with block explorers for other EVM-compatible chains** Next, in addition to the Etherscan API, we would like to possibly integrate other block explorers to analyse contracts across all EVM-compatible chains, such as BSCScan for Binance Smart Chain, Snowtrace for the Avalanche C-Chain, and so on.
3. **More plugins for detection of gas inefficient patterns** Currently, SOLBOLT is only able to detect storage variable access within loops as a gas inefficient pattern. However, many other gas inefficient patterns also exist, such as dead code, opaque predicates, potentially fusible loops, and repeated computation of the same result [18] [54]. These can possibly be implemented as additional plugins for Mythril, and be used to quickly detect potentialy avoidable gas usage in a smart contract.

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

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

[8] Defi llama: Ethereum tvl. URL https://defillama.com/chain/Ethereum. pages 2

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

[10] 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-2 pages 2

[11] 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

[12] 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

[13] 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

[14] *Solidity Documentation Release 0.8.12*, 2021. URL https://buildmedia.readthedocs.org/media/pdf/solidity/develop/solidity.pdf. pages 2, 10

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

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

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

[18] 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, 55

[19] 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

[20] Liam Wright. Bayc otherside sales cost $100m in gas fees unnecessarily due to badly optimized code. 2022. URL https://cryptoslate.com/100m-in-bayc-otherside-gas-fees-wasted-due-to-badly-optimized-code. pages 3

[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 3

[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 3

[23] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Dec 2021. URL https://ethereum.github.io/yellowpaper/paper.pdf. pages 4, 6, 7, 8, 9

[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 6

[25] Ethereum. Proof-of-work (pow), . URL https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/. pages 6

[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 6

[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 7

[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 7

[29] Ethereum. Ethereum virtual machine (evm), . URL https://ethereum.org/en/developers/docs/evm/. pages 7

[30] Solidity. *Introduction to Smart Contracts*, 2021. URL https://docs.soliditylang.org/en/latest/introduction-to-smart-contracts.html. pages 7, 8

[31] Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014. URL http://arxiv.org/abs/1407.3561. pages 8

[32] Samuel Williams, William Jones. Arweave lightpaper, Apr. 2018. URL https://www.arweave.org/files/arweave-lightpaper.pdf. pages 8

[33] Ethereum. Gas and fees, . URL https://ethereum.org/en/developers/docs/gas/. pages 8

[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 8

[35] Ethereum. Design rationale, . URL https://eth.wiki/en/fundamentals/design-rationale. pages 9

[36] Ethereum. Smart contract languages, . URL https://ethereum.org/en/developers/docs/smart-contracts/languages/. pages 10

[37] Fabian Vogelsteller, Vitalik Buterin. Eip-20: Token standard. URL https://eips.ethereum.org/EIPS/eip-20. pages 10

[38] William Entriken, Dieter Shirley, Jacob Evans, Nastassia Sachs. Eip-721: Non-fungible token standard. URL https://eips.ethereum.org/EIPS/eip-721. pages 10

[39] OpenZeppelin. Openzeppelin contracts. URL https://github.com/OpenZeppelin/openzeppelin-contracts. pages 10

[40] Solidity. Expressions and control structures, . URL https://docs.soliditylang.org/en/latest/control-structures.html. pages 10

[41] Solidity. Yul, . URL https://docs.soliditylang.org/en/latest/yul.html. pages 12

[42] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19 (7):385–394, jul 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL https://doi.org/10.1145/360248.360252. pages 13

[43] 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 13, 19

[44] 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, 17

[45] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. *CoRR*, abs/1907.03890, 2019. URL http://arxiv.org/abs/1907.03890. pages 13, 17

[46] Bernhard Mueller. Smashing ethereum contracts for fun and real profit. *HITBSecConf*, 2018. URL `https://github.com/b-mueller/smashing-smart-contracts/`. pages 13, 17

[47] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Analyzing the out-of-gas world of smart contracts. *Commun. ACM*, 63(10):87–95, sep 2020. ISSN 0001-0782. doi: 10.1145/3416262. URL `https://doi.org/10.1145/3416262`. pages 17

[48] 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 17, 43

[49] 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 18

[50] 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 19

[51] Christopher Signer. Timedcrowdsale. URL `https://github.com/ABBDVD/TimedCrowdsale`. pages 20

[52] Richard Xiong. Solbolt. URL `https://www.solbolt.com`. pages 22

[53] Martin Ortner and Shayan Eskandari. Smart contract sanctuary. URL `https://github.com/tintinweb/smart-contract-sanctuary`. pages 28, 40, 41

[54] Lodovica Marchesi, Michele Marchesi, Giuseppe Destefanis, Giulio Barabino, and Danilo Tigano. Design patterns for gas optimization in ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–15, 2020. doi: 10.1109/IWBOSE50093.2020.9050163. pages 55

Appendix A

Default settings used in evaluation

max_depth	64
call_depth_limit	16
strategy	bfs
loop_bound	10
transaction_count	2
ignore_constraints	True

Table A.1: Default symbolic execution settings