

School of Electronic Engineering  
and Computer Science

**Final Report**

**Programme of study:**

BSc FT Computer Science 3 year

**Project Title:**

**Optimising Smart  
Contract Gas  
Consumption on  
Ethereum Rollups**

**Supervisor:**

Dr. Raymond Hu

**Student Name:**

Benno Price

Final Year  
Undergraduate Project 2023/24



Date: 29/04/2024

# Abstract

This project will explore the design and implementation of a gas efficient Solidity Smart Contract compiler for Ethereum Rollups in order to reduce costs for end users. In Ethereum, executing transactions via its stack based virtual machine (a.k.a., EVM) incurs an associated transaction cost which compensates miners/validators for the computation performed as well as inclusion of the transaction within a block. Existing Solidity Smart Contract compilers as well as prior research has focused on gas optimizations at the bytecode level since gas-efficient bytecode leads to cheaper blockchain interactions.

However, despite various gas optimizations, transactions costs have continued to steadily increase with Ethereum's growing popularity, making interactions with the network prohibitively expensive for many thus stifling Ethereum mass adoption. Layer 2 scaling solutions (a.k.a., Rollups) attempt to address these bottlenecks by moving transaction execution away from Ethereum to separate blockchains which periodically post their state to Ethereum along with proof that this state was reached legitimately (i.e., abiding by the laws of the EVM). This proof is verified by Smart Contracts on Ethereum which affords users on the Rollup the same security guarantees they would otherwise have on base layer, at an orders-of-magnitude lower cost.

Whilst periodically posting Rollup state to Ethereum is what allows Rollups to inherit Ethereum's security, it is also the leading cost contributor – the larger the data posted by the Rollup to Ethereum, the larger the associated transaction cost, which is paid by users on the Rollup. A large portion of this data is something known as "*calldata*", which is the user input to Smart Contracts they interact with (i.e., call into). Reducing the size of *calldata* is the focus of this project since it results in less data posted to Ethereum leading to lower costs for Rollup end users.

Existing Smart Contract compilers implement the Ethereum ABI-encoding specification, which is extremely inefficient as all arguments, which comprise *calldata*, are padded to 32-bytes regardless of their actual size, which is often much less. This project, instead, implements a compact ABI-encoding such that compiled methods expect parameters packed together rather than padded. This leads to further reduced cost on Layer 2's in hopes of reducing the blockchain barrier of entry, that is expensive transaction costs.

# C contents

---

Chapter 1: Introduction.....	4
1.1 Background.....	4
1.2 Ethereum Rollups .....	5
1.3 Problem Statement .....	6
1.4 Aim.....	7
1.5 Objectives .....	7
1.6 Considerations.....	8
Chapter 2: Literature Review.....	9
Chapter 3: EVM Execution Environment .....	11
3.1 Dependencies .....	12
Chapter 4: Smart Contract ABI Encoding .....	13
4.1 Regular ABI Specification .....	13
4.2 Compact ABI Specification .....	14
4.3 Backwards Compatibility.....	16
Chapter 5: Featherweight Solidity .....	17
Chapter 6: Compiler .....	18
6.1 Example Smart Contract.....	19
6.2 Demonstration.....	20
6.3 Dependencies .....	21
Chapter 7: Evaluation.....	22
Chapter 8: Conclusion.....	25
Chapter 9: Further Work.....	26
References .....	27
Appendix – Optimism Testnet Transactions.....	29

# Chapter 1: Introduction

## 1.1 Background

Ethereum is a blockchain platform founded by Vitalik Buterin and launched in 2015 facilitating the creation of decentralized applications (a.k.a., “dApps”). Bitcoin first introduced the underlying blockchain technology as a tool for consensus which records transactions across a network of machines in a tamper-resistant manner. Each block contains a list of transactions, which, once added to the chain are difficult to alter as they are secured by computational work or monetary stake. Bitcoin was primarily designed as a peer-to-peer trust less digital currency eliminating the dependence on a central authority. Whilst Bitcoin also provides a primitive scripting language for its transactions, it is limited in functionality. Ethereum extends this idea by providing a fully-fledged Turing-complete virtual machine which executes programs defining arbitrary computation known as Smart Contracts (*Buterin, 2014*).

With the launch of Ethereum, a first version of a high-level object-oriented smart contract programming language called Solidity was released, which is compiled using the “solc” compiler (*Heintel, 2020*). Notably, this was influenced by several programming languages such as C++, Python, and JavaScript but introduces some of its own Ethereum specific concepts (*Solidity Documentation Release 0.8.21 Ethereum, 2023*). Vyper is another programming language for developing Ethereum smart contracts and designed with simplicity and security in mind by intentionally avoiding some of the complex features found in Solidity (*Vyper Team, 2023*). However, as Solidity is the more popular and established language it is the focus of this project.

The stack based Ethereum Virtual Machine (a.k.a., *EVM*) defines its own instruction set with each instruction having an associated cost measured in gas units (i.e., “ether”) in addition to other expenses such as state transitions and persistent storage. This is somewhat analogous to clock cycles used by a CPU to execute a single instruction. The gas consumed by a given transaction is multiplied by a user specified gas price value to arrive at the total transaction cost, which is subsequently deducted from the users Ethereum balance. The transaction cost rewards network participants for the computation performed and for including the transaction in the associated block produced. Higher gas prices incentivise miners/validators to include more profitable transactions first, which organically results in an auction mechanism leading to rising gas prices during network congestion (*Buterin, 2014*).

However, Ethereum’s growing transaction costs are simply prohibitively expensive for many retail users and pose a serious barrier to its eventual mass adoption. Whilst there have been many great efforts to optimize smart contract gas consumption at the compiler level, these resulted in, at most, marginal savings, but failed to reduce costs

by the multiple orders of magnitude required in order to onboard the masses. Consider the following hypothetical example:

*A well optimised Ethereum smart contract may use 50% less gas than its non-optimised counterparty bringing the cost of interacting with this smart contract from \$80 all the way down to \$40. However, the cost remains a major deterrent for users from alternate blockchains where transactions costs are fractions of a penny and especially for crypto non-natives who don't expect to have to pay at all.*

The desired cost reduction is simply not achievable through compiler optimizations alone. The underlying problem is that the Ethereum base layer has trouble scaling since all computation is repeated across its almost one million validating nodes in order to reach consensus (*beaconcha.in, n.d.*). Whilst this provides exceptional security, Ethereum can only scale so far vertically.

## 1.2 Ethereum Rollups

In response to scalability issues, Layer 2 horizontal scaling solutions, which move execution away from Ethereum, gained momentum. Rollups are a promising type of Layer 2 scaling solution which act as separate blockchains performing their own transaction execution whilst periodically posting aggregate transaction data to Ethereum (a.k.a., base layer) for verification. In October 2020, Vitalik Buterin announced that “the Ethereum ecosystem is likely to be all-in rollups ... as a scaling strategy for the near and mid-term future.” (*Buterin, 2020*).

There exist two types of Rollups: *Optimistic Rollups* and *ZK Rollups*. Optimistic Rollups allow users to dispute the Rollup state on Ethereum within a dispute window – once this window passes, the rollup state is considered final. ZK Rollups on the other hand, compute a zero-knowledge mathematical proof that the Rollup state is valid, which can be verified by a Smart Contract on Ethereum **without having to re-execute all transactions**. In either case, Rollup users can always *ensure*, in the case of Optimistic Rollups, or *be sure* in the case of ZK Rollups that the state is legitimate. This is why Rollups are said to inherit the security from Ethereum. That is, as an Ethereum-native user, I need not make any additional trust assumptions when migrating to a Rollup.

Both Optimistic- and ZK Rollups reduce transactions costs by orders of magnitude whilst promising the same security guarantees as Ethereum. Whilst periodically posting transaction data to Ethereum for verification is what affords Rollup users these security guarantees, it is also the leading cost contributor, as the Rollup incurs expensive Ethereum fees in doing so. The larger the data posted to Ethereum, the larger the associated transaction cost, which must be paid by users on the Rollup.

## 1.3 Problem Statement

Since Rollup costs proportionally increase with the amount of data posted to the expensive Ethereum base layer, it is prudent that the size of the data be kept minimal. The majority of this data is something known as “*calldata*”, which is the users input to every smart contract on the Rollup they interact with. Consider the following:

*Bob wants to transfer 50 APPLE tokens to Alice on the Rollup*

This would result in Bob calling the *APPLE* smart contract on the Rollup with the following *calldata*:

1. **Function selector** (e.g., *transfer*)
2. **Number of tokens** (e.g., *50*)
3. **Recipient of tokens** (e.g., *Alice*)

Whilst this transaction is executed on the Rollup, all the *calldata* is eventually posted to Ethereum. In other words, gas consumed on Ethereum increases linearly with the *calldata* size on the Rollup.

The encoding of *calldata* is defined by the Contract ABI Specification which is relatively simple and handles both static and dynamically sized arguments (*The Solidity Authors, 2023*). The problem, however, is its size inefficiency, as each argument is allocated its own 32-byte slot. In other words, each argument is padded to 32-bytes regardless of its actual size, which is often much less. This results in much larger than necessary *calldata*, leading to higher transaction costs for Rollup end users. Consider the following illustrative example:

*Imagine a method which accepts eight Boolean parameters. The calldata for this method could include all eight Booleans in a single byte. Standard ABI encoding, however, will pad each individual Boolean to 32 bytes. Thus, it introduces an overhead of  $8 \times 32 - 1 = 255$  bytes; 256 times larger than necessary!*

Whilst the above example may seem far-fetched, it is not all too unrealistic in practice. In fact, Ethereum improvement proposals such as ERC-2098 (*Moore and Johnson, 2019*) work around this limitation by manually packing cryptographic signature arguments together to take up two slots rather than three (i.e., saving 32-bytes). However, doing this for all arguments is not feasible as it requires significant changes to existing smart contracts and subsequently audits to ensure no vulnerabilities are introduced. Often, developers incorrectly assume that *calldata* is packed and choose parameter types to save space accordingly (e.g., *uint16* rather than *uint32*) just to have them all padded to 32-bytes regardless.

## 1.4 Aim

Given the importance of optimizing *calldata* size, the aim of this project is to build a Solidity compiler implementing a custom compact ABI encoding specification with the goal of reducing transaction costs for Rollup users. In essence, arguments will be packed together tightly, rather than padded, and compiler generated prologue is responsible for decoding these packed arguments into their respective 32-byte slots at runtime. Whilst this does introduce additional runtime costs, they are negligible since Rollup execution is extremely cheap. For this same reason, the compiler also need not implement any additional bytecode optimizations and can focus solely on optimising *calldata* size.

To narrow the scope of the compiler, it will implement a Featherweight Solidity, inspired by Featherweight Java (*Igarashi, Pierce & Wadler, 2002*). Whilst not as feature rich as Solidity, it allows for compilation of many basic smart contracts currently deployed on Ethereum. In order to maintain backwards compatibility, smart contracts must opt into having *calldata* packed via a new “*packed*” keyword which can be specified within the function signature – more on this later.

## 1.5 Objectives

To aid in development, debugging, and benchmarking of the compiler, the first objective is the development of a local Ethereum Virtual Machine (i.e., EVM) execution environment which allows bytecode to be executed locally against current Ethereum state without having to deploy it to Ethereum. This framework enables end-to-end continuous integration testing and asserts the validity of the compiler generated bytecode.

The second objective is designing a compact ABI encoding specification which packs arguments tightly rather than padding them with the goal of significantly reducing *calldata* size and thus decreasing transaction costs for users on Rollups. Ideally this should also be backwards compatible as to support existing smart contracts allowing developers to gradually adopt and opt-into the optimization. More on this in later sections.

The aforementioned objectives act as stepping stones towards to main objective which is the compiler itself. Again, more on this later.

## 1.6 Considerations

It is worth noting that Rollups such as *Arbitrum* already perform compression on the aggregate transaction data posted to Ethereum by using the general-purpose *brotli* compression algorithm developed by Google on its highest compression setting (*Offchain Labs, 2024*). Whilst, on one hand, this may reduce the benefits of packing *calldata* together, the two optimizations likely complement each other. Due to the general-purpose nature of *brotli*, it lacks the context specific information required to entirely omit padding but, by combining the two, compression can be performed on packed *calldata*, which is likely to perform better.

It is also worth acknowledging that optimising *calldata* size may become less effective if not redundant in the future with various cheaper data availability (a.k.a., DA) solutions on the horizon. Whilst posting Rollup transaction data to Ethereum is expensive, there exist specialised data availability networks, such as Celestia, which provide much cheaper storage. These can be used in combination with Ethereum but do present additional trust assumptions. Interestingly, Ethereum itself aims to offer cheap data availability to rollups in the future via protocol upgrades such as “Danksharding” (*Buterin et al., 2022*). This would certainly reduce if not nullify the impact of *calldata* size optimisations, but as it stands, is still several years away (*ethereum.org, 2024*).



## Chapter 2: Literature Review

This literature review aims to briefly discuss gas saving design patterns which are implemented by Solidity smart contract developers manually as well as looking at smart contract benchmarking/testing techniques. Below are a few snippets.

*“Pack the variables. When declaring storage variables, the packable ones, with the same data type, should be declared consecutively. In this way, the packing is done automatically by the Solidity compiler.” (Marchesi et al., 2020).*

This is interesting since clearly the existing Solidity compiler (i.e., *Solc*) is able to pack variables but only does so for storage variables (i.e., contract attributes) presumably since this doesn’t require any special ABI encoding. This, however, is only the case for consecutive variables of the same type which implies that the compiler is not willing to rearrange variables for optimal packing since this likely violates the developer’s intent. In a less constrained setting, reordering arguments for optimal packing could be considered desirable.

*“Pack Booleans in a single uint256 variable. To this purpose, create functions that pack and unpack the Booleans into and from a single variable. The cost of running these functions is cheaper than the cost of extra Storage.” (Marchesi et al., 2020).*

This is essentially exactly what we would like the compiler to do automatically with support for all data types, not just Booleans. This particular statement is referring to packing Booleans so that they can be written to storage but also applies to *calldata* which is written to Ethereum by the Rollup operator. The cost of running the functions being cheaper than the cost of extra storage is especially true for Ethereum Rollups where execution costs are negligible – the paper wasn’t written with Rollups in mind but there are quite a few parallels.

Clearly the idea of packing data is not new, but thus far had not been implemented natively by a compiler as far as I can tell. It is worth noting that Rollups only gained significant traction in very recent years, so this paper likely hadn’t yet considered the benefits of also packing arguments to reduce *calldata* size and consequently reducing Rollup costs.

An immediate challenge in building a Solidity compiler is being able to test the generated smart contract bytecode without having to deploy it each time. Huang et al. (2022) states that:

“To verify the effectiveness, most of the methods deploy a private chain to make verification. However, a more reasonable way is to employ the real transactions on Ethereum to trigger the contracts before and after optimization, and then compare the Gas consumption. To achieve this goal, we proposed a method, GOV, to estimate the Gas consumption of the optimized contract by using the real transactions on Ethereum. Our method enables the optimized contract to follow the execution path of the contract before optimization, thus solving the problem of inconsistent execution paths before and after optimization.”

Whilst the above snippet is mostly concerned with benchmarking rather than verifying correctness, the two are closely related. This project takes a somewhat similar approach by using a public RPC node provider (e.g., Alchemy) and forking network state at a desired block. We can then deploy any smart contracts on this forked network and interact with it to ensure it exhibits the desired behaviour. Once complete, we can simply revert state back to the original block, discarding all changes and repeat the same process for another smart contract. This eliminates any invariance between testing and allows smart contracts to interact with global network state (e.g., other smart contracts). This can be considered a middle ground between the two approaches mentioned in the extract above. It is not a private chain since it has access to all other chain state and doesn't require any real transactions on the underlying blockchain (e.g., Ethereum).

## Chapter 3: EVM Execution Environment

**Languages:** TypeScript

**GitHub Repository:** <https://github.com/bennoprice/evm-execution-environment>

**Command-line Usage:**

```
yarn hardhat run scripts/run.ts
```

The execution environment is developed in TypeScript using the Hardhat framework and provides two important features during development:

1. Forking current Ethereum network state and executing compiled smart contracts (runtime bytecode) against it. This avoids having to deploy the smart contracts to Ethereum every time which would be prohibitively expensive. Forking network state rather than just running the EVM locally offers access to all the existing Ethereum state, including other, already existing, smart contracts.
2. Deploying and benchmarking smart contracts on the Optimism Testnet once they have been tested on forked network state. Once the smart contract is deployed, a transaction is executed against it, to evaluate how much gas was used, and its state is observed before and after the transaction to ensure correctness (i.e., the state changed as expected).

It is worth noting that the execution environment is currently not very user friendly and designed with testing/benchmarking purposes in mind. Further work can certainly be done in making it more modular.

## 3.1 Dependencies

### Command-line Installation:

```
yarn install
```

Package	Version
<b>ethers</b>	<b>6.7.1</b>
<b>hardhat</b>	<b>2.17.3</b>
<b>ts-node</b>	<b>10.9.1</b>
<b>typescript</b>	<b>5.2.2</b>
<b>dotenv</b>	<b>16.4.5</b>
<b>@nomicfoundation/hardhat-ethers</b>	<b>3.0.4</b>

# Chapter 4: Smart Contract ABI Encoding

## 4.1 Regular ABI Specification

Smart contracts accept user-input via *calldata* which is analogous to programs accepting command-line inputs. This data is a series of bytes and is not required to conform to any specific structure; the user may pass arbitrary data and the smart contract may interpret this data however it chooses. In most cases, however, it is beneficial to follow a universally agreed upon encoding standard such that Ethereum libraries, which build *calldata* and smart contracts, which consume it, are compatible with one another. The Contract Application Binary Interface (a.k.a., ABI) is exactly that and defines a simple argument encoding scheme which supports both static- and dynamically sized data (*The Solidity Authors, 2023*). This standard is implemented by all major smart contract programming languages (e.g., Solidity, Vyper) and their respective compilers as well as Ethereum libraries (e.g., ethers, web3js) used by dApps, wallets, and other tooling.

To understand how exactly the encoding works, consider the following Solidity smart contract that exposes a single method called “*bar*”, which accepts a Boolean value:

```
contract Foo {
    function bar(bool value) external {
        ...
    }
}
```

Assuming the above smart contract is compiled by the standard *solc* compiler, we must generate *calldata* that adheres to the standard contract ABI encoding specification. Thus, in order to call the method “*bar*” with the Boolean value “*true*”, we arrive at the following *calldata*:

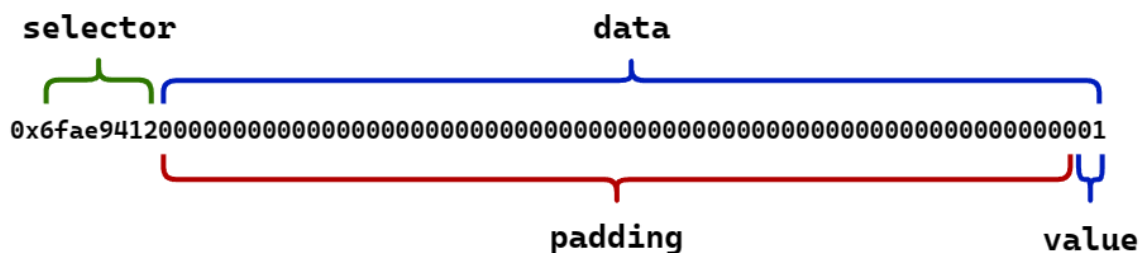


Figure i

The first four bytes contain the “*function selector*” which is a unique identifier that specifies which method we would like to call. This is computed by taking the *keccak256* hash of the function signature and discarding everything but the first four bytes. The function signature is simply the name of the function along with its parameter types comma separated and nested in parenthesis. For example, the function signature of “bar” is the following:

```
bar(bool)
```

The *keccak256* hash of the above is:

```
0x6fae9412f1ca9151f51ad8bc5e4d39869b1d4f8026734172f80601d9427db875
```

Notice that the first four bytes match the function selector from *Figure i*. The remaining data after the function selector contains the encoded function arguments, which are each allocated their own 32-byte slot. This means that each individual argument is effectively padded to 32-bytes regardless of its actual size, which is often much less. The data portion of the *calldata* from *Figure i*, for example, only contains a single byte<sup>1</sup> of useful data whilst the remaining 31-bytes are just padding. In other words, the data portion is 32 times larger than necessary which creates significant overhead resulting in increased transaction costs for Rollup users. The problem only compounds with additional arguments and nested data structures in which all fields are also padded to 32-bytes irrespective of their size.

## 4.2 Compact ABI Specification

This project defines a custom compact ABI specification, which aims to remove padding entirely and instead packs all arguments tightly together to appreciably reduce the overall *calldata* size. The packed arguments can be extracted into their corresponding slots at runtime. The specification is very straightforward and best illustrated with an example – again, consider the same smart contract this time with a minor modification.

---

<sup>1</sup> Note that a Boolean is just a single bit but is considered a byte in this instance for simplicity's sake.

```
contract Foo {
    function bar(bool value) external packed {
        ...
    }
}
```

Note the presence of a new “*packed*” keyword within the function signature, indicating that arguments are to be packed together. This keyword is not standard Solidity syntax, but introduced by this project, and allows for backwards compatibility – more on this in the next section. Assuming the above smart contract is compiled by the compiler from this project, we must generate *calldata* that adheres to the compact contract ABI encoding specification. Thus, in order to, once again, call the method “bar” with the Boolean value “true”, we arrive at the following, much smaller, *calldata*:



**selector**  
  
 0x5b971da501  
  
**value**

Figure ii

As before, the first four bytes contain the “*function selector*” which is now different in order to avoid collisions with non-packed variants of the same function – again, more on this in the next section. The function selector is computed in much the same way as before, but the function signature is appended the keyword “packed” in order to make it distinguishable. For example, the function signature of “bar” is now the following:

```
bar(bool) packed
```

The data following the function selector contains the encoded arguments which, in this case, consist of our single Boolean value. Note that the data no longer contains any padding, and the Boolean value only consumes a single byte of data, drastically reducing the overall *calldata* size. Since the *EVM* operates on 32-byte slots, packed functions must perform additional work at runtime to unpack the data into individual slots. This, however, is completely feasible and doesn’t introduce significant expenses since Rollup execution costs are negligible.

## 4.3 Backwards Compatibility

To preserve backwards compatibility, smart contract methods must opt-in to have their arguments packed by specifying a new “packed” keyword in their function signature. Smart contracts which do not include this new modifier, will continue to use the regular non-compact ABI specification by default. Since, packed functions have a unique function selector to distinguish them from their non-packed counterparts, smart contracts can implement a packed and non-packed variant of the same function in order to support both padded and packed encodings at the same time. This allows legacy smart contracts and libraries to continue calling the legacy padded methods whilst newer implementations can take advantage of the more size efficient packed methods. The main caveat that remains with the new encoding, is its incompatibility with the standard ABI specification and all tooling which implements it. However, supporting both encodings at once, somewhat eases this friction whilst existing tooling is updated or forked to support the compact ABI specification.



## Chapter 5: Featherweight Solidity

Below is the featherweight Solidity syntax implemented by the compiler. This syntax is by no means feature complete nor is it designed to be. Whilst certain syntax is missing, much can be implemented in terms of what is available. For example, whilst **public** variables are not supported, their functionality can be implemented via **private** variables with explicitly defined getter methods. Constructors are also not supported but can be implemented via a one-time callable **init** method.

```

program ::= contract
contract ::= contract TYPE { feature + }
feature ::= attribute | method
attribute ::= TYPE private ID;
method ::= function ID ( [params] ) accessibility [returns] { expr+; }
params ::= TYPE ID [ , TYPE ID ]*
accessibility ::= internal | public packed
returns ::= returns ( TYPE )
expr ::= ID ( expr [ , expr ]* )
        | expr * expr
        | expr / expr
        | expr + expr
        | expr − expr
        | expr ≤ expr
        | expr < expr
        | expr == expr
        | ! expr
        | expr ? expr : expr
        | ID = expr
        | TYPE ID = expr
        | ( expr )
        | return expr
        | ID
        | integer
        | true
        | false

```

## Chapter 6: Compiler

Languages: Java

GitHub Repository: <https://github.com/bennoprice/SolidityCompiler>

Command-line Usage:

1. Navigate to project root:

```
cd ...
```

2. Build the compiler:

```
./buildme
```

3. Navigate to build folder:

```
cd ./build
```

4. Run the compiler, passing source code path as first argument:

```
./compile "{PROJECT_ROOT_PATH}/examples/hello_world.sol"
```

This generates two output files in the `./out` directory:

1. **Runtime bytecode** (.runtime) which contains the deployed smart contract bytecode as stored on the blockchain.
2. **Creation bytecode** (.creation) which is the code responsible for deploying the aforementioned runtime bytecode.

## 6.1 Example Smart Contract

The following Solidity smart contract demonstrates various language features implemented by the compiler such as reading/writing various variable types (e.g., locals, parameters, attributes) as well as arithmetic and function calls. This smart contract will be referred to in subsequent sections as “*HelloWorld.sol*”. Other examples are also included in the repository demonstrating other syntax such as ternary expressions, casting, and conditionals.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.23;

contract HelloWorld {
    uint256 private attr;

    // accepts eight single-byte values
    // sums up the values and calls "mul"
    // passes the result to "bar"
    function foo(
        uint8 arg1, uint8 arg2, uint8 arg3, uint8 arg4,
        uint8 arg5, uint8 arg6, uint8 arg7, uint8 arg8
    ) public packed {
        uint256 local =
            arg1 + arg2 + arg3 + arg4 + arg5 + arg6 + arg7 + arg8;

        local = mul(local, 2);
        bar(local);

        // single line comment

        /*
        this is a...
        multiline comment
        */
    }

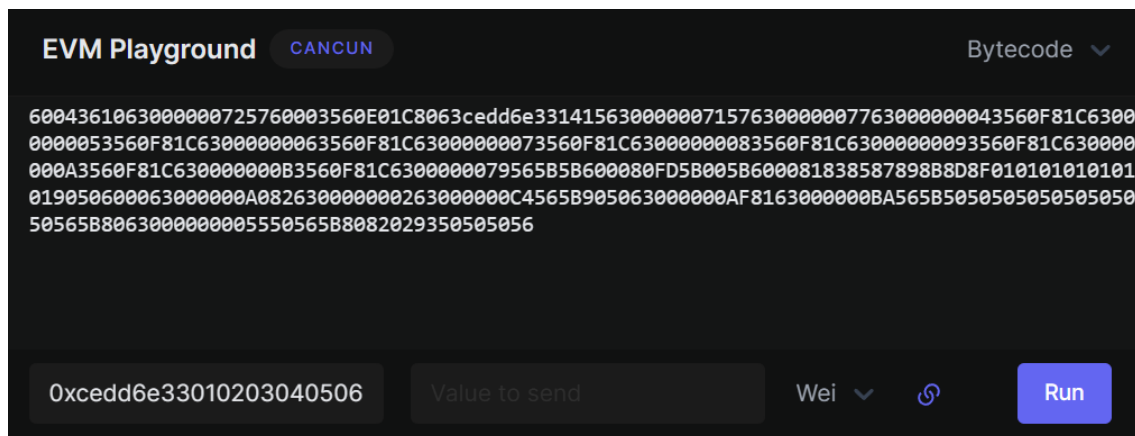
    // assigns value to an attribute (i.e., writes it to storage)
    function bar(uint256 x) internal {
        attr = x;
    }

    // returns the product of x * scalar
    function mul(uint256 x, uint256 scalar) internal returns (uint256) {
        return x * scalar;
    }
}
```

## 6.2 Demonstration

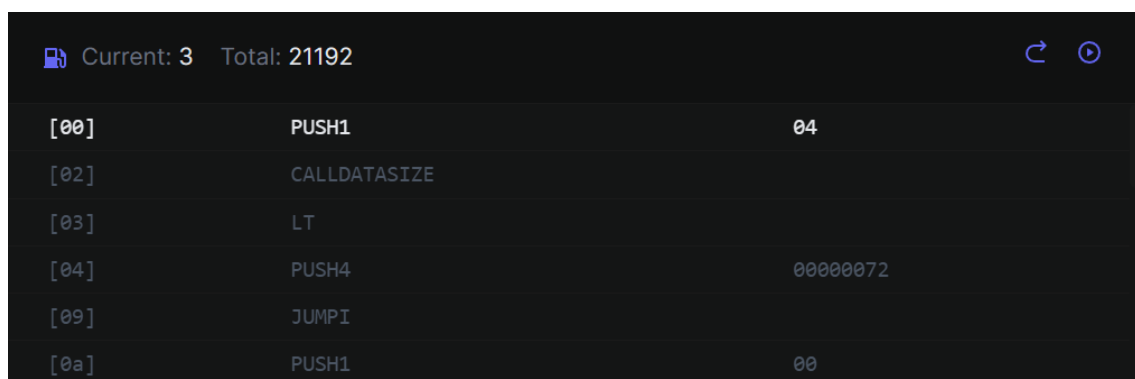
The compiled smart contract in the output directory can either be deployed to an EVM-compatible network by including its creation bytecode in an EVM transaction or by executing the runtime bytecode locally via an online EVM interpreter such as [evmcodes playground](#). The latter is nice for demonstration purposes since it allows stepping through the code and doesn't require contract deployment each time.

The [following link](#) redirects to an interactive *evmcodes* session containing the compiled **HelloWorld.sol** smart contract runtime bytecode as seen below:



Below the bytecode, on the far left of the run button, is the transaction *calldata*. Recall, that the first four bytes comprise the function selector, in this case specifying the **foo** method, and the remaining bytes are the function arguments, in this case the numbers one through eight (single-byte values).

To execute the bytecode, hit the “Run” button. The window to the right, highlights the current instruction being executed as shown in the figure below:



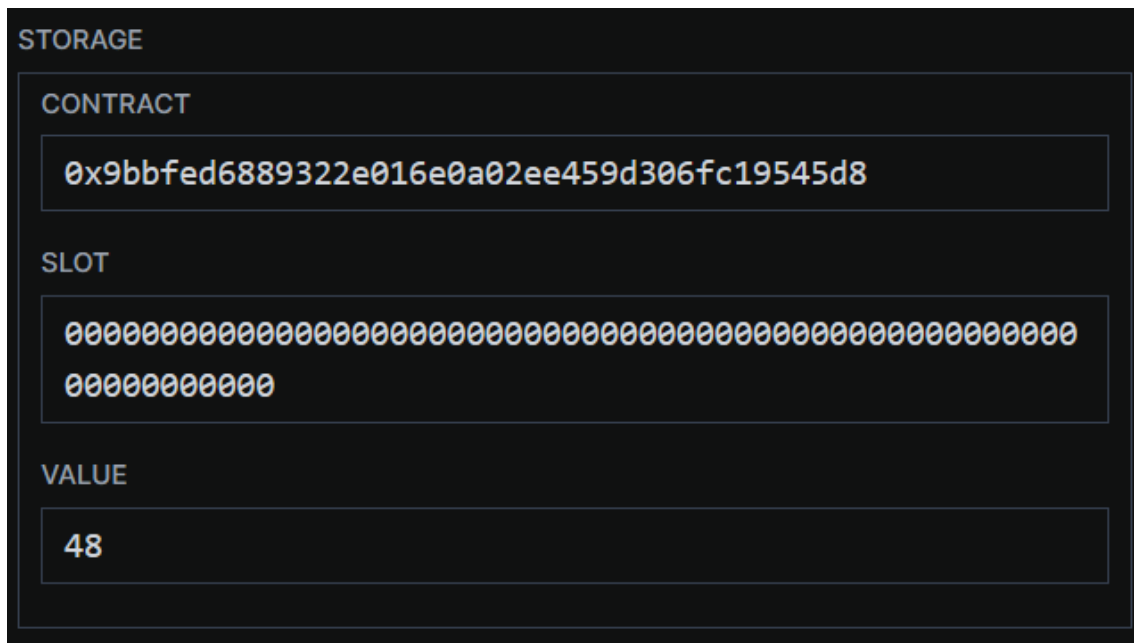
The two buttons at the top-right of the above figure allow...

- Stepping forward a single instruction (button on the left)
- Continuing execution until a breakpoint is hit or until termination if none are set (button on the right)

Once the smart contract has finished executing, we can observe its state (e.g., stack, storage, memory) to determine what outputs it computed. In this example, the smart contract sums all of its inputs and doubles the result. Thus, the output we expect is the following:

$$output = 2(1 + 2 + 3 + 4 + 5 + 6 + 7 + 8) = 72 = 0x48$$

Indeed, this is the output computed by the smart contract as seen in the figure below:



## 6.3 Dependencies

These are included in the repository and do not need to be installed manually.

Package	Version
<b>antlr-4.6-complete</b>	4.6
<b>bcprov-jdk18on-1.78.1</b>	1.78.1

## Chapter 7: Evaluation

To recap, the main objective is to reduce costs for Ethereum Rollup (i.e., Layer 2) end users by building a compiler implementing a custom compact ABI encoding to reduce smart contract *calldata* size. This way, users calling smart contracts on Rollups will post less data to Ethereum, which accounts for the majority of the cost. The assumptions here are that:

1. Compact ABI encoding reduces *calldata* size.
2. Compact ABI encoding is feasible to implement in practice.
3. Reducing *calldata* size reduces the cost associated with posting data to Ethereum (i.e., less gas is consumed by the Rollup data availability layer).

Evaluating the above criteria consists of compiling the ***HelloWorld.sol*** smart contract both with the reference compiler (i.e., *Solc*) and with the compiler developed in this project, herein referred to as “*MySolc*”. Both compiled smart contracts are deployed on Optimism Testnet<sup>2</sup>, which is a popular Ethereum Rollup, and then called with the same inputs (i.e., both calling the same “*foo*” method with the same arguments). We then compare:

1. *Calldata* size.
2. L1 (Ethereum) gas consumed.
3. Smart contract output – the outputs/state changes/behaviour of the smart contracts should match. This determines whether Solidity was faithfully implemented by *MySolc*.

This process of compiling the smart contract with both compilers, deploying it on Optimism Testnet, and calling it is repeated seven more times – each time removing an argument from the *foo* method. This allows us to better observe the effects of packing a various number of arguments.

Before presenting the evaluation results, let’s first visualise the *calldata* produced for the following (pseudocode) contract call below. We pass eight single-byte values to *foo*. Recall, *Solc* will pad each individual argument to 32-bytes whereas *MySolc* will pack them together tightly.

```
HelloWorld.foo(1, 2, 3, 4, 5, 6, 7, 8);
```

---

<sup>2</sup> Optimism Testnet is used since it exhibits the same behaviour as Optimism Mainnet whilst not charging real Ethereum native tokens (a.k.a., ETH), which reduces development costs.

For the *Solc* compiled ***HelloWorld.sol*** variant, we generate the following *calldata*:

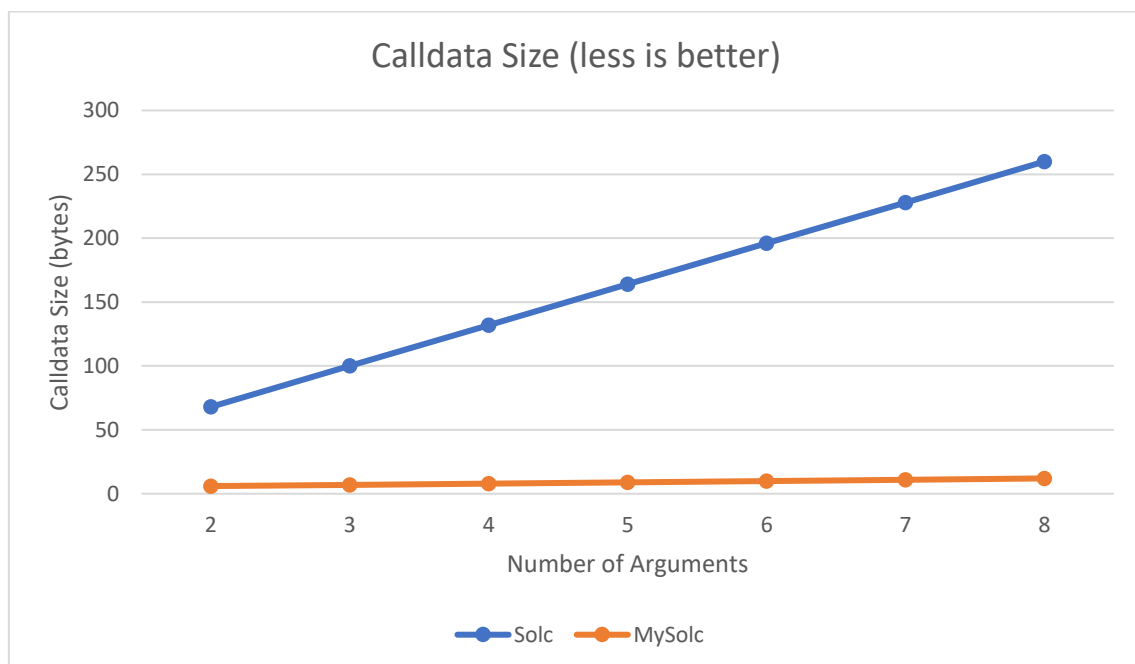
[illegible]

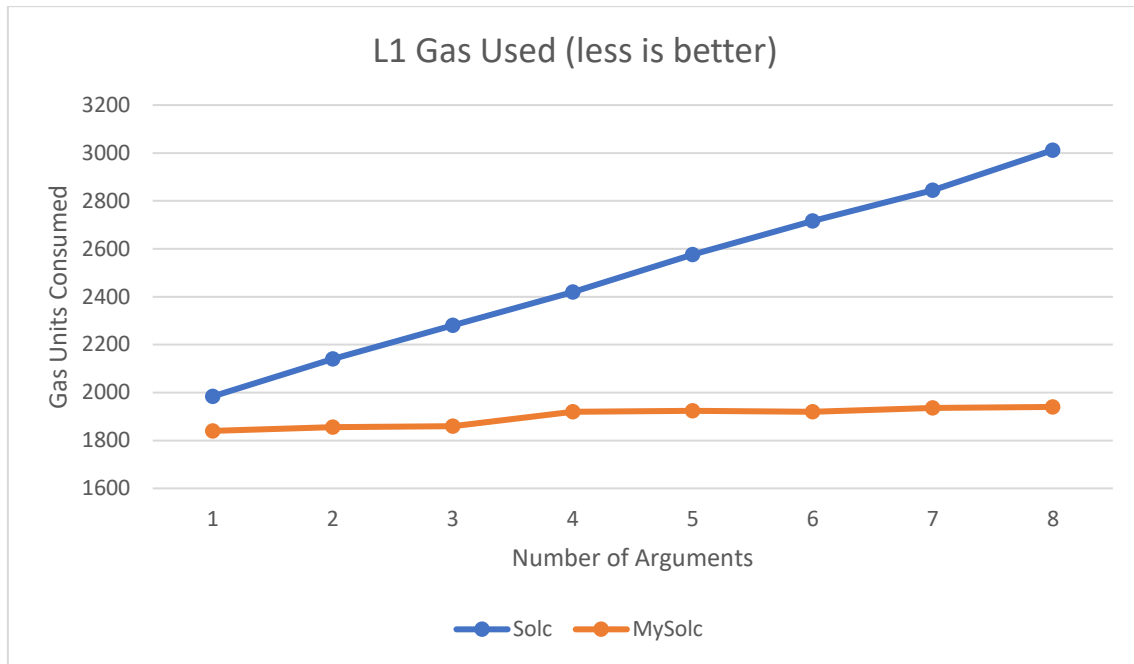
For the *MySolc* compiled ***HelloWorld.sol*** variant, we generate the following *calldata*:

```
0xcedd6e330102030405060708
```

In both instances, the first four bytes contains the function selector. Note, the selector is different since *MySolc* introduces the *packed* keyword to preserve backwards compatibility. Following the function selector are the function arguments which are much more compact in the *MySolc* variant (32-times smaller).

This demonstrates that indeed, packing arguments tightly significantly reduces *calldata* size. The remaining question is whether this actually reduces L1 gas consumed and leads to any real cost savings – below are the results.





The first chart shows the increase in *calldata* size with the number of function arguments for the **HelloWorld.sol** smart contract compiled with *Solc* and *MySolc* respectively. As expected, given the above *calldata* breakdown, the *MySolc* compiled smart contract requires much less *calldata*. In both cases, *calldata* size increases linearly with argument count, but much more rapidly for *Solc* compiled smart contracts.

The second chart shows much the same as the first, but with L1 gas consumed on the y-axis instead of *calldata* size. As hypothesized, there is a direct correlation between *calldata* size and L1 gas used. The variability in the second chart can be attributed to fluctuating L1 (i.e., Ethereum) gas prices, which contribute towards how much L1 gas the Rollup consumes. It is worth noting that, even with just a single function argument, the *Solc* compiled smart contract used more L1 gas than the *MySolc* compiled smart contract with eight function arguments. The L1 gas consumed by *MySolc* compiled smart contracts almost appears to remain constant despite an increase in arguments.

It is also worth noting at this point that *MySolc* doesn't unconditionally reduce *calldata* size. Since it simply packs arguments tightly it reduces *calldata* size by however much smaller the data type is than 32-bytes (which is what *Solc* pads to). Thus, if all arguments are 32-byte, this optimization won't yield any *calldata*/L1 gas reduction. However, this means that in the worst case, our bytecode is as efficient as *Solc*, and in the best case, much more *calldata* size efficient (up to a factor of 32). In the above example, all arguments are single-byte types which represents the best-case scenario.



## Chapter 8: Conclusion

As it stands the compiler is very much a proof-of-concept and not feature rich by any means. However, it clearly demonstrates the feasibility and applicability of reducing *calldata* size via a compact ABI encoding, consequently reducing Rollup L1 gas usage, which represents the majority of the cost. In the worst case, where all arguments are 32-bytes, it is just as efficient as the reference *Solc* compiler but, in the best case, reduces *calldata* size by up to a factor of 32. Since it achieves this whilst preserving full backwards compatibility, it seems prudent for existing Solidity compilers to offer this *packed* encoding as a feature especially for, but not limited to, smart contracts deployed on Ethereum Rollups. Developers can then choose whether or not to opt-in circumstantially. The main hurdle, as I see it, is adoption since this new compact ABI encoding specification must be implemented by existing tooling. However, this doesn't seem so unlikely as the majority of Ethereum libraries are open source and the spec could be proposed and/or implemented via a pull request from a community member to the respective repository as has often been the case in the past.

It is worth acknowledging that reducing *calldata* size may be made somewhat redundant with the rise of alternative data availability solution which can store Rollup data at a fraction of the cost at which point optimizing *calldata* size yields diminishing returns. Ethereum itself plans to provide much cheaper data availability for Rollups via an upgrade called *Danksharding*. In any case however, there, again, don't seem to be any real caveats to supporting this *packed* encoding regardless, other than the work required in implementing it.

## Chapter 9: Further Work

There are many avenues of further work worth exploring. The most obvious perhaps is fleshing out the compiler to support the entirety of Solidity rather than just a featherweight adaptation. This was beyond the scope of this project but is necessary for adoption since developers have come to expect various language features from other compilers. Alternatively, the compact ABI specification outlined in this paper could be implemented by existing Solidity compilers (e.g., *Solc*) – this may be more sensible than implementing it from scratch since these are already maintained by active teams and will receive frequent updates with new language features. An added benefit is that *Solc* already performs optimizations at the bytecode level which, whilst less relevant for Ethereum Rollups where execution is extremely cheap, are still nice to have especially if deploying the smart contract to Ethereum directly.

More work can also be done on the compact ABI encoding specification since currently only fixed size primitive types are supported but no dynamically sized data (e.g., byte arrays) or structs. Whilst slightly more complex to implement these, it is entirely possible. Structs will likely see the most *calldata* size savings as all fields are padded to 32-bytes in the regular ABI specification. Whilst functions can only accept up to eight arguments (due to EVM constraints), structs can have many more than eight fields which can all be packed together.

It would also be interesting to explore other more sophisticated encoding techniques. Simply packing the arguments together is nice because extracting them again incurs very little runtime cost but could potentially be further optimised. Compression algorithms could be explored and may significantly reduce size especially for low entropy *calldata*. The question remains whether it is worth doing this at the expense of having to recover the original data at runtime. Presumably there is an optimal balance between compression algorithm complexity and compression ratio, but this is only speculation and would have to be researched further.

Another important task is implementing the compact ABI encoding specification in existing Ethereum libraries and tooling either via pull requests or forks. This could be achieved by posting a refined specification on platforms such as *Ethereum Improvement Proposals* and/or *Ethereum Research* at which point the community may get involved assuming it gains traction.

## References

- Buterin, V. (2014). Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. [online] Available at: [https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum\\_Whitepaper\\_-\\_Buterin\\_2014.pdf](https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf).
- Ethereum. (2023). Solidity Documentation Release 0.8.21. [online] Available at: [https://docs.soliditylang.org/\\_downloads/en/latest/pdf/](https://docs.soliditylang.org/_downloads/en/latest/pdf/).
- Heintel, F. (2020). Solidity v0.1.0 turns 5! A walk down memory lane... [online] Available at: <https://soliditylang.org/blog/2020/07/08/solidity-turns-5/>.
- Vyper Team (originally created by Vitalik Buterin). (2023). Vyper Documentation. [online] Available at: [https://vyper.readthedocs.io/\\_downloads/en/stable/pdf/](https://vyper.readthedocs.io/_downloads/en/stable/pdf/).
- Igarashi, A., Pierce, B., Wadler, P. (2002). Featherweight Java: A Minimal Core Calculus for Java and GJ. [online] Available at: <https://www.cis.upenn.edu/~bcpierce/papers/fj-toplas.pdf>
- Buterin, V. (2020). A rollup-centric Ethereum roadmap. [online] Available at: <https://ethereum-magicians.org/t/a-rollup-centric-ethereum-roadmap/4698>.
- beaconcha.in. (n.d.). Validators Chart - Open Source Ethereum Blockchain Explorer - beaconcha.in - 2024. [online] Available at: <https://beaconcha.in/charts/validators> [Accessed 18 Mar. 2024].
- L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino and D. Tigano, "Design Patterns for Gas Optimization in Ethereum," 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), London, ON, Canada, 2020, pp. 9-15, doi: 10.1109/IWBOSE50093.2020.9050163.
- Y. Huang, R. Wang, X. Chen, X. Zhou and Z. Wang, "GOV: A Verification Method for Smart Contract Gas-Optimization," 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), Guangzhou, China, 2022, pp. 473-479, doi: 10.1109/QRS57517.2022.00055.
- Buterin, V. (2020). A rollup-centric ethereum roadmap. [online] Fellowship of Ethereum Magicians. Available at: <https://ethereum-magicians.org/t/a-rollup-centric-ethereum-roadmap/4698> [Accessed 18 Mar. 2024].
- The Solidity Authors (2023). Contract ABI Specification — Solidity 0.8.26 documentation. [online] docs.soliditylang.org. Available at: <https://docs.soliditylang.org/en/develop/abi-spec.html> [Accessed 18 Mar. 2024].

Moore, R. and Johnson, N. (2019). ERC-2098: Compact Signature Representation. [online] Ethereum Improvement Proposals. Available at: <https://eips.ethereum.org/EIPS/eip-2098> [Accessed 18 Mar. 2024].

Offchain Labs (2024). Inside Arbitrum Nitro | Arbitrum Docs. [online] docs.arbitrum.io. Available at: <https://docs.arbitrum.io/inside-arbitrum-nitro/> [Accessed 18 Mar. 2024].

Buterin, V., Feist, D., Loerakker, D., Kadianakis, G., Garnett, M., Taiwo, M. and Dietrichs, A. (2022). EIP-4844: Shard Blob Transactions. [online] Ethereum Improvement Proposals. Available at: <https://eips.ethereum.org/EIPS/eip-4844> [Accessed 18 Mar. 2024].

ethereum.org (2024). Danksharding. [online] ethereum.org. Available at: <https://ethereum.org/en/roadmap/danksharding/> [Accessed 18 Mar. 2024]

www.evm.codes. (n.d.). *EVM Codes*. [online] Available at: <https://www.evm.codes/>.

Ethereum Research. (n.d.). *Ethereum Research*. [online] Available at: <https://ethresear.ch/>.

Ethereum Improvement Proposals. (n.d.). *Home*. [online] Available at: <https://eips.ethereum.org/>.

## Appendix – Optimism Testnet Transactions

Solc (Reference Compiler)			
Num. Args	Calldata Size	L1 Gas Used	Tx Hash
1	36	1,984	<a href="#">0x79c...12c</a>
2	68	2,140	<a href="#">0x279...fbb</a>
3	100	2,280	<a href="#">0x11f...b45</a>
4	132	2,420	<a href="#">0xa40...9b5</a>
5	164	2,576	<a href="#">0x719...9b3</a>
6	196	2,716	<a href="#">0x6a1...9e9</a>
7	228	2,844	<a href="#">0x569...bd6</a>
8	260	3,012	<a href="#">0x786...939</a>

MySolc (My Compiler)			
Num. Args	Calldata Size	L1 Gas Used	Tx Hash
1	5	1,840	<a href="#">0x14b...057</a>
2	6	1,856	<a href="#">0xa9e...0a0</a>
3	7	1,860	<a href="#">0x6d0...993</a>
4	8	1,920	<a href="#">0x279...580</a>
5	9	1,924	<a href="#">0x4e2...f0a</a>
6	10	1,920	<a href="#">0xd51...332</a>
7	11	1,936	<a href="#">0xf9b...339</a>
8	12	1,940	<a href="#">0xc7e...eea</a>