NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Macaron - A tool for examining solidity smart contract transactions on the ethereum blockchain

Ioannis A. Cheilaris

Supervisors: **Neville Grech,** Lecturer
Yannis Smaragdakis, Professor

ATHENS

DECEMBER 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Macaron - Ένα εργαλείο εξέτασης συναλλαγών έξυπνων συμβολαίων στην αλυσίδα-μπλόκ του Ethereum

Ιωάννης Α. Χείλαρης

Επιβλέποντες: **Νέβιλ Γκρέτς,** Λέκτορας
Γιάννης Σμαραγδάκης, Καθηγητής

ΑΘΗΝΑ

ΔΕΚΕΜΒΡΙΟΣ 2020

BSc THESIS

Macaron - A tool for examining solidity smart contract transactions on the ethereum blockchain

Ioannis A. Cheilaris

S.N.: 1115201500176

SUPERVISORS: **Neville Grech,** Lecturer
Yannis Smaragdakis, Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Macaron - Ένα εργαλείο εξέτασης συναλλαγών έξυπνων συμβολαίων στην αλυσίδα-μπλόκ του Ethereum

Ιωάννης Α. Χείλαρης

Α.Μ.: 1115201500176

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Νέβιλ Γκρέτς,** Λέκτορας
Γιάννης Σμαραγδάκης, Καθηγητής

ABSTRACT

In recent years, blockchain technologies have seen continuous expansion. One such case is Ethereum, with its smart contracts finding widespread use. However, there is a lack of development toolsets for the inspection and interpretation of smart contracts. Our prototype aims to provide something new in this field, offering multi-level examination - allowing the user to take in the greater picture of a transaction whilst still being able to focus on the actual source code that was executed. Our tool makes extended use of the solidity compiler, using its Abstract Syntax Tree representation of a contract's source code to facilitate a translation between the bytecode executed and the source code related to said bytecode. Finally, as part of out case study we have included in our report the a comparison between our tool and similar ones already developed, an analysis of its accuracy, and our findings after running our tool in the wild, on a part of the Ethereum blockchain.

SUBJECT AREA: Ethereum Smart Contracts

KEYWORDS: Smart Contracts, Ethereum, Solidity, Blockchain, Visualization

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια, οι τεχνολογίες αλυσίδων μπλόκ έχουν δει συνεχή ανάπτυξη. Μια τέτοια περίπτωση είναι το Ethereum, του οποίου τα έξυπνα συμβόλαια έχουν γίνει αρκετά διαδεδομένα. Βέβαια, υπάρχει μια έλλειψη απο εργαλεία που θα έκαναν την ανάλυση και ερμήνευση αυτών των συμβολαίων πιο εύκολη για τους χρήστες. Το πρόγραμμά μας έχει ως στόχο να βοηθήσει με αυτό το πρόβλημα, προσφέροντας πολυ-επίπεδη εξέταση των έξυπνων συμβολαίων αφήνοντας στον χρήστη να κατανοήσει την ευρύτερη εικόνα μιας συναλλαγής ενώ ταυτόχρονα επιτρέπει την δυνατότητα να εστιάσει στον κώδικα του συμβολαίου που εκτελέστηκε. Κάναμε εκτεταμένη χρήση του μεταγλωττιστή της γλώσσας solidity στην προσπάθειά μας, χρησιμοποιώντας την ενδιάμεση αναπαράσταση του κώδικα σε μορφή Abstract Syntax Tree για να επιτύχουμε μια έγκυρη μετάφραση μεταξύ του bytecode που εκτελέστηκε και του κώδικα που συσχετίζεται με αυτό το bytecode. Τέλος, ως μέρος της έρευνάς μας προσθέσαμε στην πτυχιακή αυτή μια σύγκριση μεταξύ του εργαλείου μας και άλλων που ήδη υπάρχουν, μια ανάλυση της ευστοχίας του και τα πορίσματά μας αφότου εκτελέσαμε το εργαλείο μας πάνω σε μέρος της αλυσίδας μπλόκ του Ethereum.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Έξυπνα συμβόλαια στο Ethereum

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Έξυπνα Συμβόλαια, Ethereum, Solidity, Αλυσίδα Μπλόκ, Οπτικοποίηση

To my faithful companion, Charlemagne...

ACKNOWLEDGEMENTS

I am grateful to my professor, Prof. Yannis Smaragdakis for giving me the chance to work in an intriguing field and for his stellar work ethic of putting the student first.

I would also like to extend my warmest and most sincere thanks to my supervisor, Neville Grech, whose continuous support, guidance, and patience proved fundamental to the completion of this work.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This report is my bachelor thesis for the conclusion of my undergraduate studies at the Department of Informatics & Telecommunications of the National and Kapodistrian University of Athens. It was developed using python3.9 and the solidity compiler, and serves as an extensible infrastructure for future work on inspecting, interpreting and visualizing the Ethereum blockchain.

1. INTRODUCTION

During the last decade, the number of digital monetary exchanges has risen dramatically. This in turn was facilitated by the founding of the world's first cryptocurrency; Bitcoin [20]. In subsequent years, numerous other cryptocurrencies - estimated to be more than 1600 - have been deployed and enjoy a share of the market, with the primary contender being Bitcoin, and the secondary challenger being Ethereum [19].

Specifically, Ethereum is a cryptocurrency founded on blockchain technology; a decentralized mechanism with the sole purpose of consensus-based transaction validation. Ethereum though, is a much larger beast. Instead of only providing a network for transactions to occur in, it also boasts an impressive tool; smart contracts. A smart contract is a piece of computer code that resides on the blockchain and performs a specific function or functions. Currently, the main contract-oriented language used for writing smart contracts is Solidity.

This thesis presents *Macaron*, a tool designed to make exploring the execution of ethereum transactions on solidity contracts easier. Since the transaction execution trace data are in low-level bytecode form, this prototype creates an improved visual output by mapping said bytecode instructions to the Solidity source code they were generated from and showcasing changes in peristent variables.

2. RELATED WORK

Similar tools already exist, each with their own feature set. Bitquery offers a high-level view transaction trace display, giving emphasis on the function calls that occured, without showing what transpired between solidity instructions[2]. Other such high-level navigators are the Oko contract explorer, which instead shows calls between contracts[8], and Bloxy, the most verbose of the three, since it not only exhibits calls between contracts and function calls, but also the calldata of the calls[3].

A different approach is followed by the Truffle Suite, which boasts an impressive arsenal, being able to debug transactions, though it is designed for use while developing a contract and cannot examine all the transactions on the ethereum blockchain[18].

Great strides have been made with instruments that delve into the source code of a contract and focus on showing which high-level instructions were executed, what results they had, and what changes they made to the peristent layer of a contract. A stellar example is Remix, which gives both low-level information (stack and memory alterations) but also presents the user with the respective source code that made said alterations occur[9]. Perhaps the most ambitious undertaking is Auditless, which offers all of the above in addition to indicating value assignments on a per variable basis[1].

3. BACKGROUND

3.1 Cryptocurrencies and the blockchain technology

A cryptocurrency is an instrument that allows its users to provide payment for goods and services; a virtual monetary system free from a central trusted authority. Instead, a peer-to-peer networking system is implemented to assure and safeguard the validity of all transactions by using a consensus-based protocol. The first such currency was Bitcoin, deployed in 2009 by an unknown developer with the pseudonym ”Satoshi Nakamoto” and soon after that, many more cryptocurrencies emerged[20].

Since the point of such systems is decentralized control, validation, and security, an appropriate foundation is necessary. Thus, the idea of the blockchain was conceived[20] - a distributed ledger that is curated and certified by its userbase[22].

3.2 Introduction to smart contracts

There is more potential to be found with regard to the blockchain. So far, we have described a sophisticated accounting system, but the versatility of the blockchain’s architecture allows for so much more complex behaviour. Any data or asset can be digitized and embedded in a blockchain, being offered for trade. This idea can be realized with the use of computer code, otherwise referred to as a ’smart contract’[19]. A very important first iteration of this notion was the extension of the bitcoin protocol, which utilized Script - a simple stack-based non-Turing complete language[10]. Perhaps the most advanced and famous blockchain for its smart contract integration is Ethereum;[19] offering a robust bytecode instruction execution unit, the Ethereum Virtual Machine (EVM)[23].

3.3 The Ethereum Virtual Machine

The EVM has a simple stack-based architecture with a word size of 256-bits, to underpin the Keccak-256 hash scheme and elliptic-curve calculations. It offers a word-addressed byte array volatile memory model and an independent persistent storage. Its stack has a maximum size of 1024 entries. Rather than having program code stored in an accessible area, it is placed in a virtual ROM interactable only through a specialized instruction [23].

3.4 Solidity

One of the most notable languages used to write smart contracts on the ethereum blockchain is Solidity; an object-oriented, high-level language. It is statically-typed and supports inheritance, libraries and complex user-defined types[16]. Solidity code is compiled, optimized and deployed on the Ethereum network in EVM bytecode form.

3.4.1 Solidity internals

Of great interest to this thesis are the inner workings of the Solidity language concerning how the storage is modelled and how function calls are implemented.

Storage is persistent, meaning that its values are retained on the blockchain. Its structure is a mapping of 32-byte addresses to 32-byte values. Statically-sized variables are laid out contiguously in storage, beginning from address **0**. In the case of multiple contiguous items that need less than 32 bytes, these are placed into a single storage slot if possible, to conserve size, according to the following:

- The first item in a storage slot is stored lower-order aligned.
- Basic types such as boolean, int, etc, use as many bytes as are necessary to store them.
- If a basic type cannot fit in the remaining part of a storage slot, it is moved to the next slot.
- Struct and array data always occupy a new whole slot.

Contracts that use inheritance have the ordering of state variables be determined by the C3-linearized order of contracts starting with the most base-ward contract. If allowed by the rules above, state variables from different contracts can share the same storage slot. In addition, the elements of structs and arrays are stored one after the other.

Dynamically-sized objects are a different case however. Mappings and dynamic arrays use a Keccak-256 hash calculation to locate the starting position of the value or the array data, due to their unpredictable size. The mapping or the dynamic array itself occupies a slot in storage at some position p in accordance to the rules above. In the case of dynamic arrays, that slot stores the number of elements of said array. For mappings, the slot is unused. Array data are located at `keccak256(p)`, in successive slots and the value of a key k is located at `keccak256(k . p)`, where `.` denotes concatenation.

`Bytes` and `string` are encoded identically. If possible, byte arrays store their data in the same slot where their length is stored. Specifically, if the data is less than 32 bytes, it is then stored in the higher-order bytes (left aligned) and the lowest-order byte contains `length * 2`. In the case of the data being 32 or more bytes long, the main slot stores `length * 2 + 1` and the data is stored in `keccak256(slot)`[11].

On the subject of function calls, they can be split in two types: internal ones that do not create an actual EVM call and external ones that do[14]. When solidity code is compiled into EVM bytecode, internal calls are achieved by simple `JUMPs` between `JUMPDESTs`, while external calls use the `CALL` opcode to refer to other contracts.

3.4.2 Compiler code transformations

Since code executed on the blockchain incurs costs to the transactor and space is of limited quantity, one often encounters optimized contracts. The optimizer itself operates on assembly. It starts by splitting the sequence of instructions into basic blocks at `JUMPs` and `JUMPDESTs`. Afterwards, these blocks are analyzed, their behaviour (stack, memory or storage changes) is recorded, and common expressions found are recursively eliminated, thus shortening the code length. After this process, a control flow graph is built by using the

extracted knowledge and the code in each block is re-generated. Moreover, a dependency graph is generated which is used to drop dangling operations[15].

3.4.3 Compiler output used

Of great importance and interest to this thesis are three output options of the solidity compiler, namely the code's abstract syntax tree (AST) representation, the source map, and the contract metadata. The AST of a given contract is a tree where each node is a language construct abstraction and each node's children are part of said abstraction. Each node also contains a `src` field that shows to which part of the source code the node refers to - also known as a source mapping[12].

These source mappings comes in the form of `s:l:f`, where `s` is the byte-offset to the start of the mapping area in the source file `f`, and `l` is the mapping's length. The source map is a list of contiguous source mappings that adhere to the more complex notation `s:l:f:j:m` and are seperated by `;`. Each of those mappings corresponds to a specific EVM instruction with the `s, l ,f` fields being the same, the `j` takes one of the values `i, o, -`, signifying whether a jump instruction goes into a function, returns from a function or is a regular jump as part of a loop, branch, etc. The last field, `m`, is an integer that indicates the "modifier depth". This depth is increased whenever a placeholder statement `_` is entered in a modifier and decreased when it is left again. However, the solidity optimizer can prune and merge identical basic blocks, making the source mapping information inaccurate at times.

To avoid unnecessary waste of space, the source map is compressed using the following rules:

- If a field is empty, the last element's value is used.
- If a `:` is missing, then the following fields are considered to be empty.

For example, the following two source maps represent the same information:

```
1:2:1;1:9:1;2:1:2;2:1:2;2:1:2

1:2:1;:9;2:1:2;;
```

[17]

Finally, the contract metadata is a `JSON` file that contains information about a compiled contract. This file can be used to extract the compiler version, the used source files, the contract's ABI, and the compilation settings amongst other properties. The hash of this file - the type of which depends on the compiler version used - is also encoded and appended at the end of the deployed contract's bytecode[13]. Our focus shall remain on the ABI however, which is used to extricate a contract's function signature and arguments, to be used in the calldata visualization process.

3.5 The Ethereum JSON-RPC Protocol API

JSON-RPC is a stateless, light-weight remote procedure call protocol which uses JSON as its data format and defines certain data structures and the rules about their processing[7].

Since this thesis was developed using the Go Ethereum (GETH) client in mind, we had a few rather usefull rpc calls at our disposal: namely `eth_getTransactionByHash` and `debug_TraceTransaction`. The former returns information about a transaction, such as its gas, the address of the receiver, etc, by supplying the call with said transaction's hash[5]. The latter is a call that returns the bytecode that was executed during a transaction and the state of the EVM (storage, stack, memory) throughout the whole process[6].

3.6 Contract-Library

Contract-Library (`https://contract-library.com`) is a free service aimed towards offering decompiled versions of all contracts on the Ethereum blockchain. Praised by the Ethereum community, it remains an invaluable tool for security analysts and receives several unique visitors per day[21]. We have used this instrument to get fast and reliable access to the source code of the transactions that our prototype aims to inspect.

4. TECHNICAL ASPECTS

Our prototype concentrates on transactions that have occurred on the Ethereum blockchain. It is not a tool made for debugging contracts, but one for exploring flows of execution and the history of the blockchain. Currently, it supports the inspection of contracts compiled with versions of the solidity compiler greater than **v0.4.10** for the minimum features. Contract storage access visualization is supported for compiler versions **v0.5.13** and above.

4.1 Features

The features are the following:

- Basic navigation through a transaction's flow of execution, split in basic blocks.
- Display of solidity source code that was executed in each block.
- Visualization of any accesses made to the contract's storage.
- Print the contents of an expression related to contract storage.
- Presentation of the calldata of any external call that was made.
- High-level view that briefly shows which calls were made, in what order, and how they were terminated.
- Low-level display of EVM instructions executed in each basic block.

4.2 Phases of processing

In this section, we will explain how the prototype produces its results by examining each phase of the assembly line, from the initial acquisition of the input data, to the end of processing.

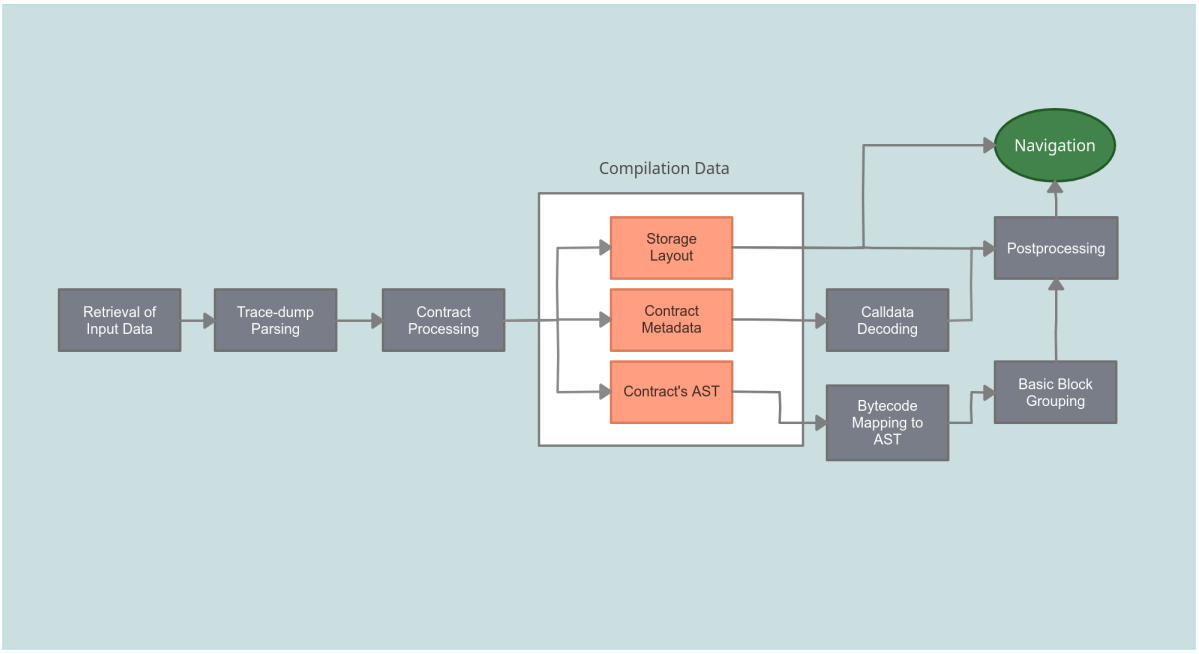

Figure 4.1: Macaron's phases of processing

4.2.1 Retrieval of input data

Firstly, it is necessary to obtain the transaction trace-dump, which lists what opcodes were executed and in which order, along with other data such as gas costs, EVM data, call depth, etc. This is done by performing the `debug_traceTransaction` JSON-RPC call on a Geth node of our choice. It is necessary to request both the stack, memory, and storage data, in order to properly extract and display the calldata and storage accesses. After being served with a response, to actually examine a transaction, we require the source code of all the contracts that said transaction pertains to, the compiler version that was used to produce the deployed bytecode, and the optimization options used (if any). The relevant compilation data will be querried by establishing a connection to Contract-Library's mysql database after the following section.

```
1  pragma solidity ^0.7.4;
2
3  contract Foo {
4      mapping(address => uint256) balances;
5
6      function deposit() external payable
7      {
8          balances[msg.sender] += msg.value;
9      }
10
11     function getBalance() view external returns(uint256)
12     {
13         return balances[msg.sender];
14     }
15 }
```

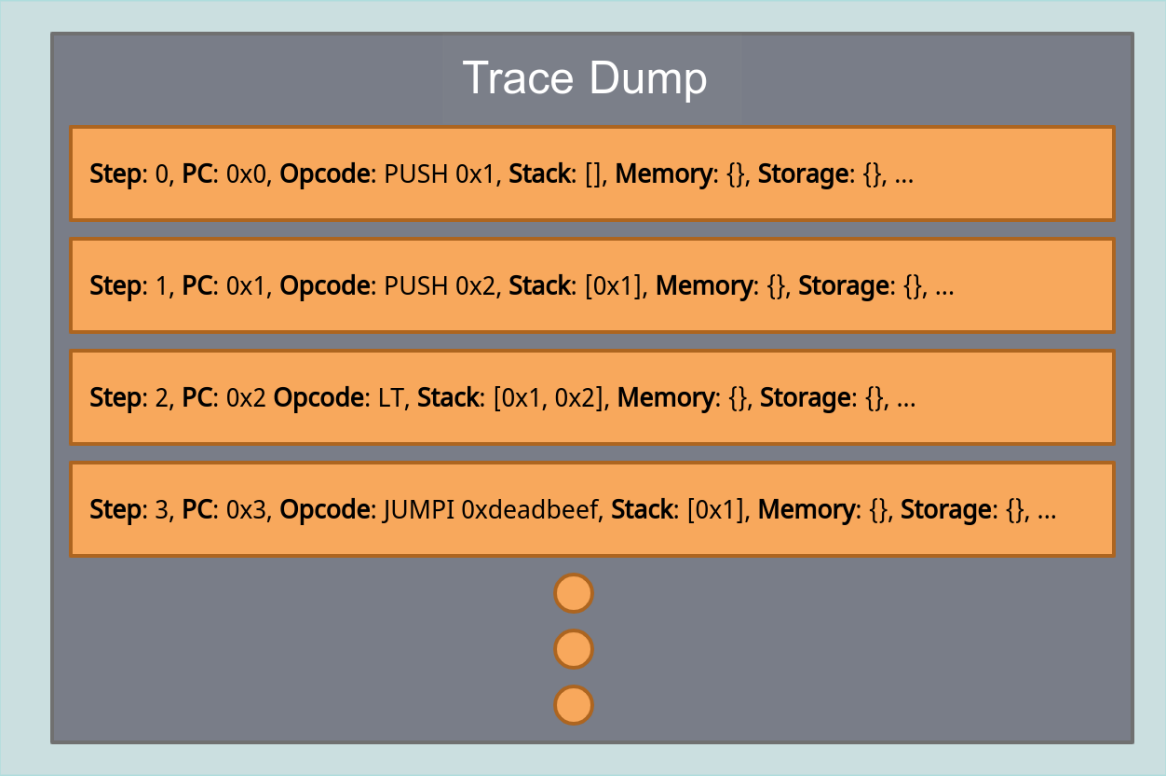

Figure 4.2: Sample input data

4.2.2 Trace-dump parsing

The initial processing of the trace-dump begins by populating a stack machine with entries. This machine contains three lists; one simulating the actual call-stack, one modelling the memory of each call during execution, and one storing the whole trace history. The latter is what we will use for our visualization calculations.

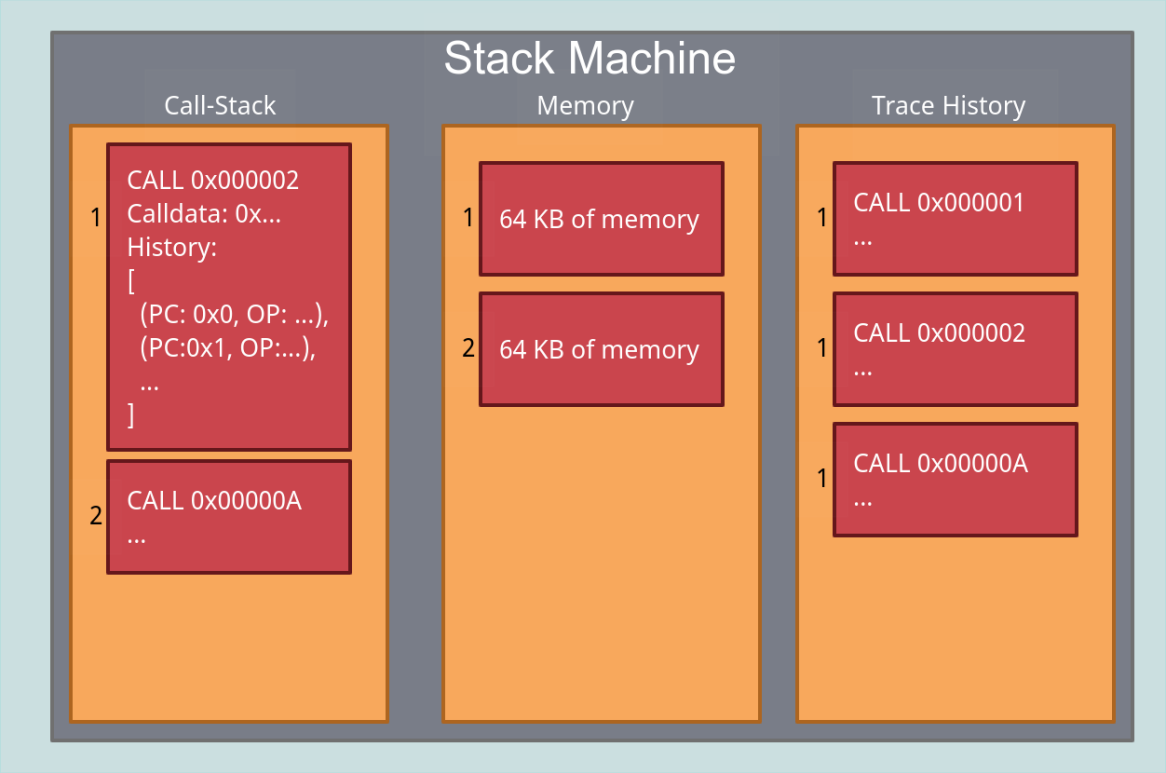

Figure 4.3: Stack machine in the midst of parsing

More precisely, the program creates a starting call-stack entry and begins iterating through the trace-dump, appending the encountered EVM bytecode and system snapshots to the entry's history. Upon encountering instructions that relate to memory access, contract code copying, or to the calldata, (such as `MSTORE`, `MSTORE8`, `CALLDATACOPY`, `CODECOPY`), their side effects are recorded and kept track of. Since we have not yet extraced which contracts were called, the calldata remain in their raw hexadecimal form, awaiting further processing. In addition, upon discovering an opcode associated with calling, (for instace `CALL`, `CALLCODE`, `STATICCALL`, `DELEGATECALL`, `CREATE` or `CREATE2`), if the call is to a contract and not to an external - user - address, which is how the solidity `<address>.transfer` function is implemented; facilitating the movement of eth between addresses, then a new stack entry is created and all new history changes will be appended there; this is recorded in the trace history too. Finally, when the recorded contract call depth of the current trace-dump entry is reduced, the call-stack's last entry is popped and the trace-history stack makes a record of why. In the end, we have the full call history ready to be further used.

4.2.3 Contract processing

Succeeding the trace parsing, recorded in the trace-history stack are all the contracts that will be reached by the current transaction, directly or otherwise. Thus, we start by retrieving the source code and compilation data of all the contracts and then compiling them. If an error is encountered, that will be reflected on the program's output. The contract compilation output is also cached locally, to avoid unnecessary and time-costly network activity. An invaluable addition to the compiler output is the *storage layout*, a storage variable index that records information like their type, slot number, and other useful data. We use

this to both evaluate any expressions given by the user and translate storage addresses of statically-sized variables. It is available for solidity compiler versions **v.0.5.13** and on. Furthermore, the metadata of each contract are used to find the signatures and parameter data of all its functions, to be utilized in the calldata decoding process. A function's signature is the canonical expression of the basic prototype without a data location specifier, meaning the function name with the parenthesised list of parameter types, split by a single comma. We will save this data for easy access later and refer to each function by its selector, i.e. the first (left, high-order in big-endian) four bytes of the `keccak256` hash of the function signature. The data used for this process are found in the ABI of the contract[4].

4.2.4 Calldata decoding

Every entry in the trace-history stack refers to a contract call (or a return from a call), the calldata of which have already been extracted by our stack machine. Since we have in our possession the called function's argument data, we can start combing through the calldata and generate a string of the form `function_name(arg1=val1, arg2=val2, ...)`.

4.2.5 Bytecode mapping to the AST

The whole foundation of this project is the actual procedure of mapping EVM bytecode instructions to the source code they were generated by. To begin with, we start iterating through the trace-history and decompressing each source map using the rules we mentioned in the background section. Afterwards, we proceed by examining the whole bytecode and the decompressed source map in parallel, recording each instruction's mapping. Following that, we run through the instructions encountered during the transaction and after making sure that they have a valid mapping,i.e. related to code generated by the compiler, we apply a DFS search on the contract's AST, trying to find nodes whose source field matches to the instruction's mapping. Special attention is given to inline assembly instructions, which are not minutely modelled on the AST. Instead of invalidating them, we create our own custom AST nodes which offers a degree of flexibility. Eventually, if no match is found, the user will be notified later. This can happen if newer solidity versions alter the AST's structure or if the AST isn't that detailed, like in the case of inline assembly.

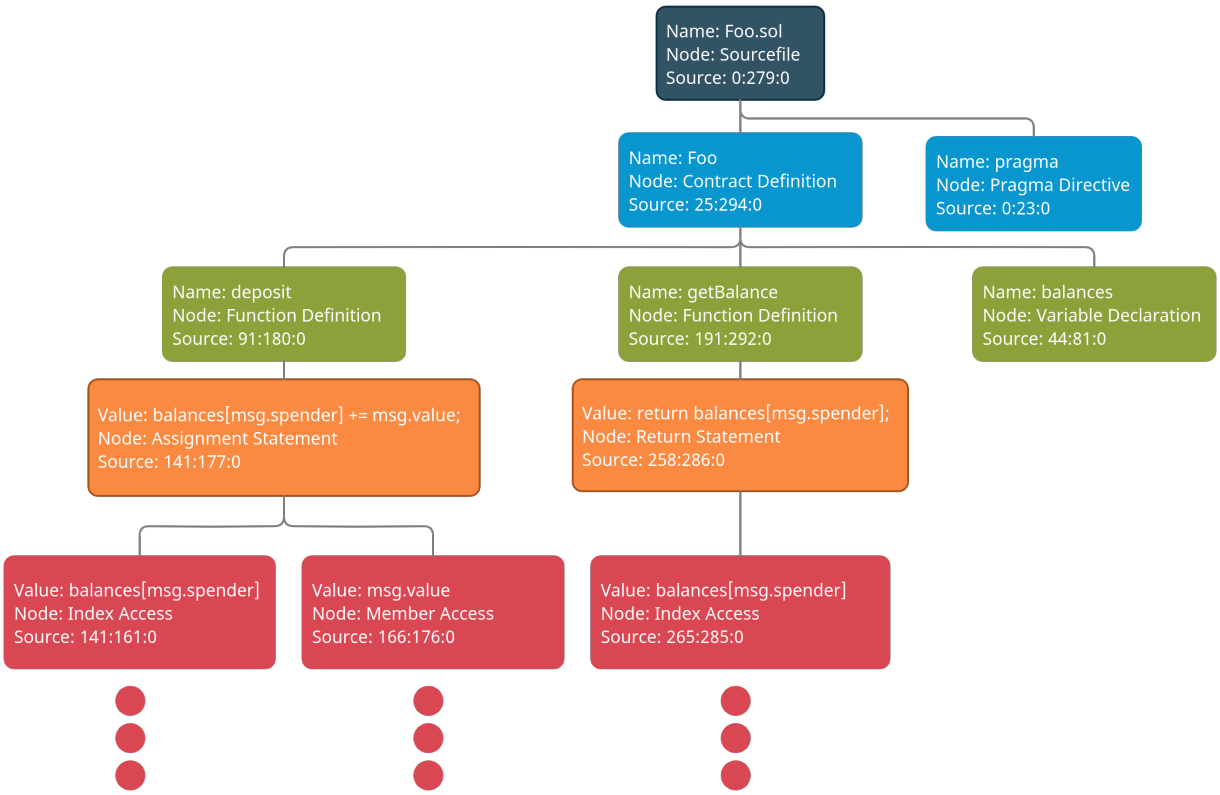

Figure 4.4: Sample AST

4.2.6 Group instructions by basic block

After the bytecode to AST mapping has been completed, we continue by splitting each call's instructions into groups of basic blocks. A set is kept, which gradually increases with the addition of new AST nodes of the instructions we encounter. Upon encountering a `JUMPDEST` instruction, we append our set, along with which instructions were executed and in which order, plus the latest EVM snapshot (stack, memory, and storage contents at a given point), to our output and empty the set. In addition, in each entry appended we also include the node of the function to which all these instructions belong to.

4.2.7 Postprocessing

What remains now, is to choose what to display to the user.

So far, we have kept the EVM snapshot information, which is quite bloated and unnecessary. A much more thoughtful approach is to indicate which storage changes have been made between basic block executions. Therefore, we make an annotation whenever two successive storage snapshots are different and highlight the previous and changed value of the storage address that was accessed.

Moving on, we examine the node set that each basic block has been mapped to and decide which nodes' source code is valuable. For example, there exist nodes like `FunctionDefinition`, whose source code is a whole function. Such nodes provide useless information and are not used in the highlighting process. Mappings to these nodes are usually made by `JUMPDEST` and `JUMPs` that are generated from solidity loops or branches. We opted not to automatically remove these kind of mappings in the instruction grouping process; instead,

the user can alter which nodes should be highlighted and which should be discarded by changing the `macaron_utils.py` python file used by the program. We utilize the remaining nodes' source index to highlight the executed source code.

In the end, we try to translate any storage addresses to their related variable name. Since `keccak256` is not a reversible process, we can only lookup statically-sized variables whose position in storage can be inferred by the storage-layout module. Dynamically-sized variables like mappings, strings and dynamic arrays cannot be translated.

4.2.8 The Navigator

The navigation module has been engineered with simplicity in mind. The user can move back and forth on the trace's history, easilly detect storage changes, and move between the source code view and the contract call view. The user can save certain transactions with names of their choosing to make the inspection easier. It is also possible to write expressions and have their evaluation printed. A state machine consumes the input expression and formulates a set of calculations that should be applied in order to determine the storage address it points to. Those calculations are produced using the rules about how variables are stored in storage[11] and the storage-layout module.

5. EVALUATION

5.1 Evaluation method

The evaluation was performed with both quantitative and qualitative measures in mind. In order to test the accuracy and usefulness of *Macaron*, we asked the following questions:

- How well does Macaron facilitate the translation of a transaction trace?
- How do these translated transaction traces facilitate the understanding of the execution of a transaction?
- How robust is Macaron on real-world transactions

5.2 Accuracy of the translation

Our case study starts with being able to validate the accuracy of our tool. In this example, we will take a look at the following test contract:

```
pragma solidity ^0.5.13;
contract Fib
{
    modifier is_valid_request(uint n)
    {
        require(n >= 0);
        require(msg.value == n);
        _;
    }

    function call_fib(uint n, bool call_rec) is_valid_request(n) public payable returns(uint)
    {
        if (call_rec)
            return fib_rec(n);
        else
            return fib_iter(n);
    }

    function fib_rec(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;
        else
            return fib_rec(n - 1) + fib_rec(n - 2);
    }

    function fib_iter(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;

        uint prevfib = 0;
        uint prevprevfib = 1;
        uint fib;

        for(uint i = 2; i <= n; i++)
        {
            fib = prevfib + prevprevfib;
            prevprevfib = prevfib;
            prevfib = fib;
        }

        return fib;
    }
}
```

We will examine how unerringly the call `Fib.call_fib(4, true)` with a value of `4 wei` is translated through the following step-by-step figures.

The first part of the transaction relating to choosing which function will be called is not shown, since it is not generated from the source code itself. We can see that the modifier `is_valid_request` gets called first.

```
################################################################################
EVM is running code at 0288277a9d6459fefe64b43aaaf01a69b8783719. Reason: ENTRY
Calldata: call_fib.value(0.000000 ETH)(n = 3, call_rec = true)
No bytecode was found for this contract
step 0:
line: 4 : modifier is_valid_request(uint n)
    {
        require(n >= 0);
        require(msg.value == n);
        _;
    }
```

Figure 5.1: Step 0

Afterwards, `call_fib` is entered, and after a condition check `fib_rec` is called.

```
step 1:
line: 11 : function call_fib(uint n, bool call_rec) is_valid_request(n) public payable returns(uint)
    {
        if (call_rec)
            return fib_rec(n);
        else
            return fib_iter(n);
    }
```

Figure 5.2: Step 1

First call of `fib_rec`:

```
step 2:
line: 19 : function fib_rec(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;
        else
            return fib_rec(n - 1) + fib_rec(n - 2);
    }
```

```
step 3:
line: 19 : function fib_rec(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;
        else
            return fib_rec(n - 1) + fib_rec(n - 2);
    }
```

```
step 4:
line: 19 : function fib_rec(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;
        else
            return fib_rec(n - 1) + fib_rec(n - 2);
    }
```

Figure 5.3: Steps 2 - 4

Second call of `fib_rec`:

```
step 5:
line: 19 : function fib_rec(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;
        else
            return fib_rec(n - 1) + fib_rec(n - 2);
    }
```

Figure 5.4: Step 5

Return from recursive call and calling `fib_rec`:

```
step 6:
line: 19 : function fib_rec(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;
        else
            return fib_rec(n - 1) + fib_rec(n - 2);
    }
```

Figure 5.5: Step 6

Second call of `fib_rec`:

```
step 7:
line: 19 : function fib_rec(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;
        else
            return fib_rec(n - 1) + fib_rec(n - 2);
    }
```

```
step 8:
line: 19 : function fib_rec(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;
        else
            return fib_rec(n - 1) + fib_rec(n - 2);
    }
```

```
step 9:
line: 19 : function fib_rec(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;
        else
            return fib_rec(n - 1) + fib_rec(n - 2);
    }
```

Figure 5.6: Steps 7 - 9

Third call of `fib_rec`:

```
step 10:
line: 19 : function fib_rec(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;
        else
            return fib_rec(n - 1) + fib_rec(n - 2);
    }
```

```
step 11:
line: 19 : function fib_rec(uint n) private returns(uint)
    {
        if (n == 1)
            return 1;
        else if (n == 0)
            return 0;
        else
            return fib_rec(n - 1) + fib_rec(n - 2);
    }
```

Figure 5.7: Steps 10 - 11

The trace continues similarly, as seen in the following condensed diagram.

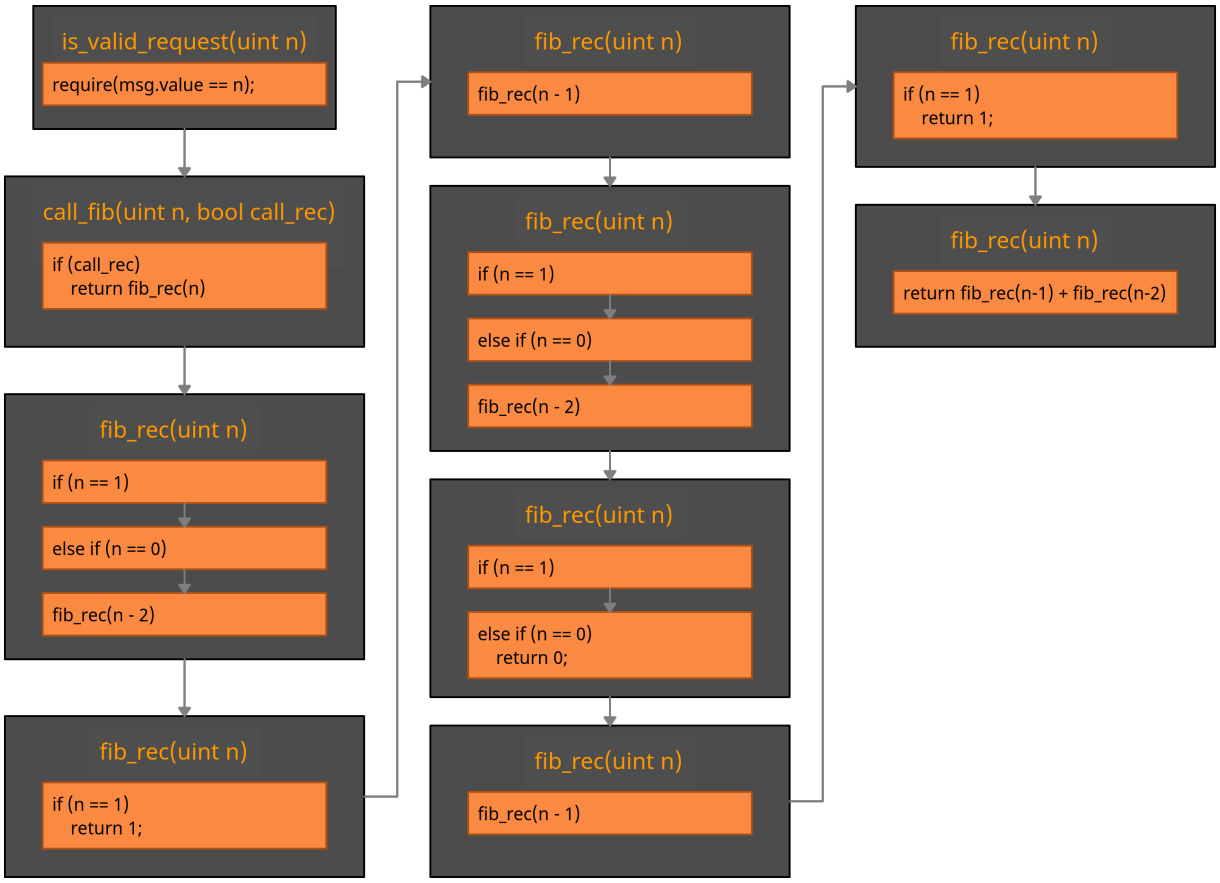

Figure 5.8: Trace Diagram

From simple test cases such as this, we can perceive that our tool is positively accurate.

5.3 Intuitiveness of the translation

Accuracy by itself means nothing if the user is not able to understand and interpret the program output. For that, we present the following two transaction visualization comparisons, between our tool and other known tools like *Bloxy* and the *Oko contract explorer*.

Below, we will examine the transaction with the hash:

0x4af8741a3db3e9dc2f657f701247a291c2cbd720e04b83f442dbe37be016cefa

At first glance, *Macaron* performs quite similarly to *Bloxy*. The high-level view is approximately the same, barring the exception of event logs accompanying the execution trace in *Bloxy*.

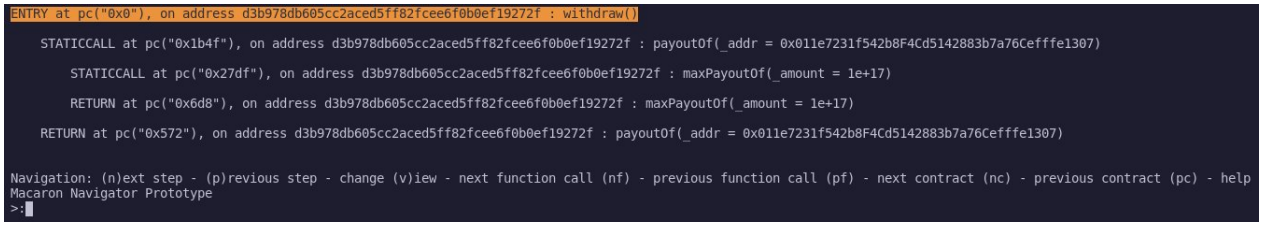

Figure 5.9: Macaron high level view

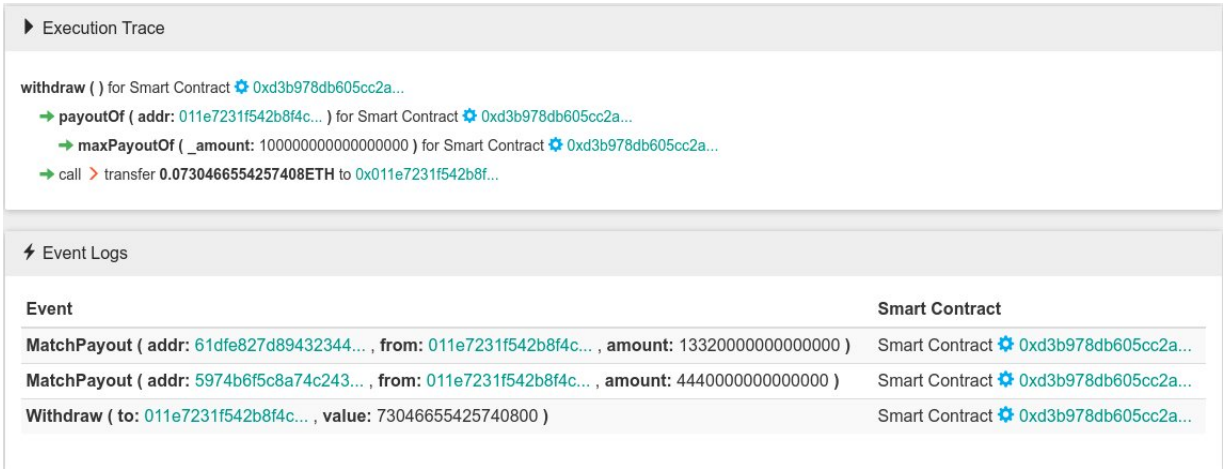

Figure 5.10: Bloxy View

Nevertheless, we offer something that *Bloxy* does not; finer granularity in our execution trace. Specifically, as witnessed in the previous section, we can jump back and forth in the transaction trace, observing which parts of the code were executed and which were not. Fluidity and ease of use were our primary concerns, so we tried to make easy transitioning from one view to the next. In the next example, we can see the user moving From the targeted `payoutOf` function in the high-level view, to the function's solidity code.

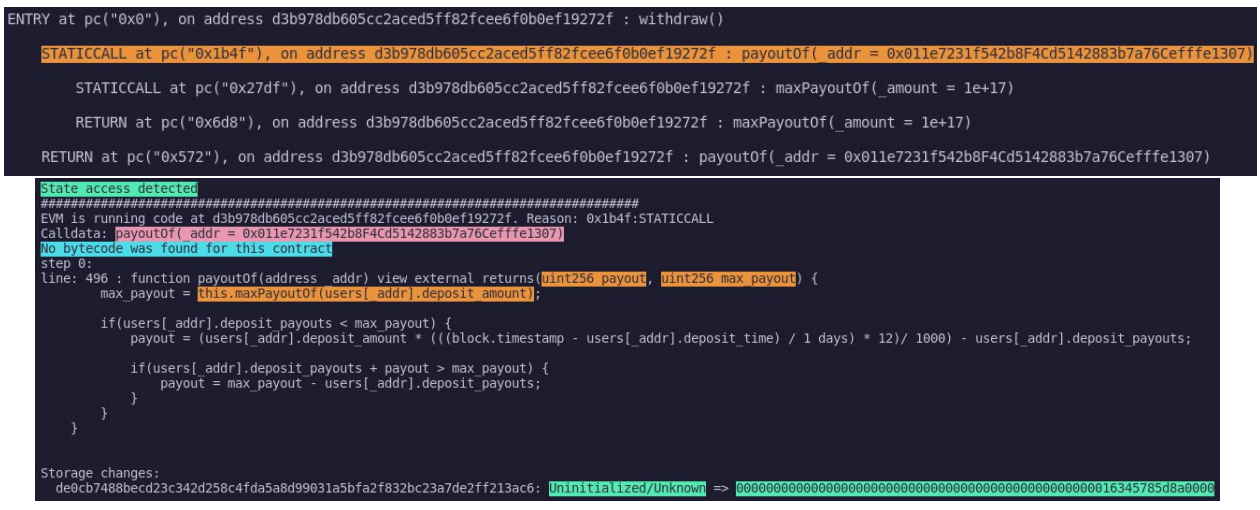

Figure 5.11: Moving between views in Macaron

In the next example, we will focus on our instrument's storage display capabilities versus the *Oko contract explorer's*. We'll take a look at the transaction with the hash:

0xf6ad2c27c7c71dc93b5f2c333e2e878d4f42c35160b9e7a8617e8cf9e0baa1fc

Figure 5.12: Oko Contract Explorer Output

As is evident by the figure above, the information displayed that concerns the inner state of the contract is rather convoluted and illegible. For instance, the *Oko contract explorer* is refering to numbered storage slots as `storN`, where `N` the number of the slot in question, instead of translating the storage address to a variable name. *Macaron* comes to rectify that.

The following figure was taken during the middle of navigation.

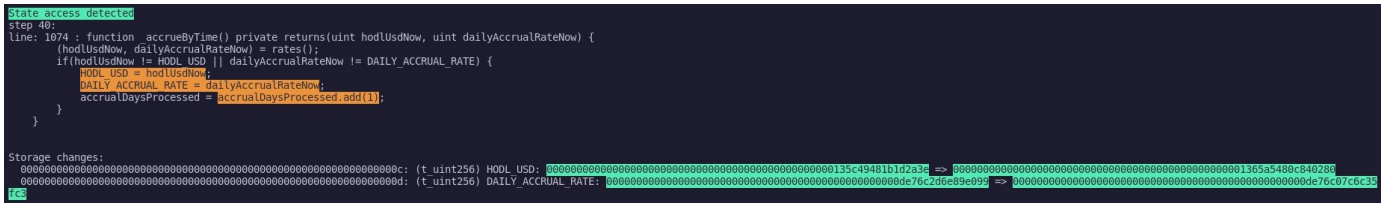

Figure 5.13: Macaron storage display

We can see from the code being executed that not only are storage accesses and changes registered, but that the relevant variable names are attached to the storage addresses as well. Where the *Oko contract explorer* outputs the value of storage address 12 (`stor12`) to be $1,395070559 * 10^{18}$, *Macaron* can infer that storage address 12 is where the contents of

the variable `HODL_USD` (with type `t_uint256`) reside. In the case of dynamically allocated variables, that is not possible, but we have a feature that can help in this instance: printing.

In the next two figures we see the user navigating through a basic block that accesses and then alters the contract's storage, but the storage address has no variable name attached. We can surmise by reading the code that the storage access happened by reading the conctents of `userStruct[msg.sender]`. We can verify that by typing

```
print userStruct[0xA2ED0Fd7A7A1781770D3aF711484f22B3F33b94C].balanceEth
```

which outputs the same value as the storage access notification. This way, a user can read the contents of any storage variable referenced in the transaction trace, at any point in time.

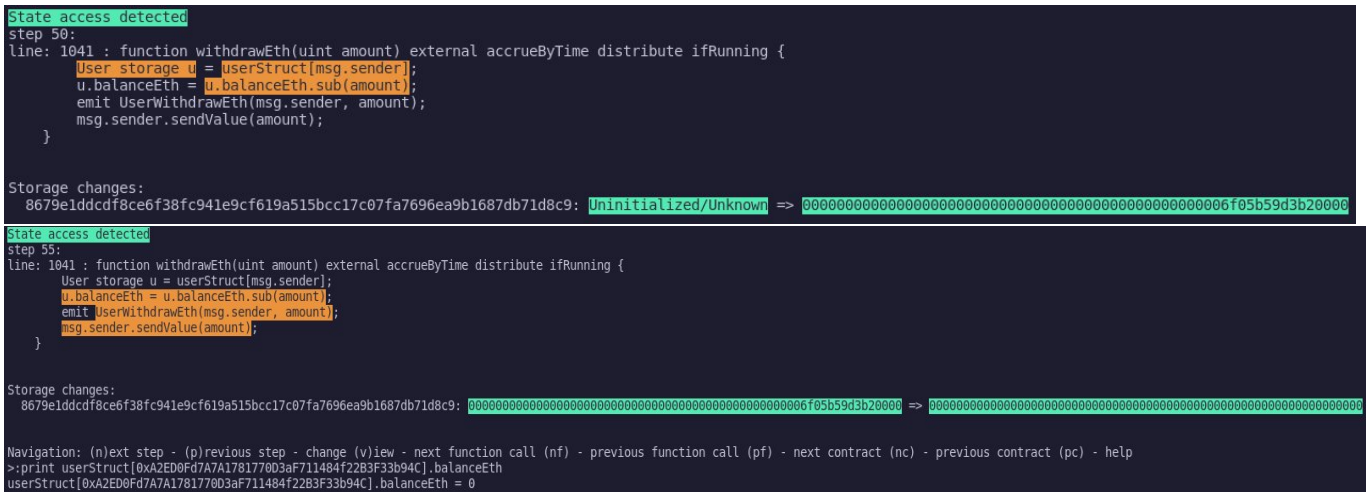

Figure 5.14: Printing in Macaron

5.4 En-masse application of Macaron on the Ethereum Blockchain

In this final section, we will take a look at the results we gathered after running *Macaron* in the wild, on a small random portion of the Ethereum blockchain. We applied our tool on 20 sequential blocks, making sure to filter out contract creation transactions and transactions towards wallets. The total amount of transactions that remained were **2332**; of those, approximately **.02%** encountered a contract with a solidity version lower than **v0.5.0** during their execution, making them unable to be processed. Around **13%** encountered compilation errors due to unknown solidity import directives. Thus, overall we achieved an **87%** success rate.

Table 5.1: En-masse application statistics

Number of Transactions	Incompatible AST	Compilation Errors	Success Rate
2332	.02%	13%	87%

6. CONCLUSIONS AND FUTURE WORK

6.1 Limitations

As seen so far, our design has some deficiencies. To start with, it does not support the examination of contracts compiled with a solidity version lower than **v0.5.0**, making it unnable to work on a large part of the blockchain. Secondly, the inspection of contracts that use the solidity `import` directive, often fail. Thirdly, certain features like stepping over or out of a function, or skipping the execution of loops can't be implemented when the solidity optimizer is activated, since it merges basic blocks and contributes to information loss. Finally, we haven't taken advantage nor added support for solidity's new Yul optimizer.

However, there also are conceptual limitations. For instance, our toolset can only be utilized on the blockchain's history, and cannot be used to help during smart contract development. In addition, *Macaron* relies on external data of great size - namely the trace-dump - which could be otherwise avoided if it simulated the transaction instead of parsing it. Last but not least, our prototype is using a command-line interface, making the selective visualization and expansion of data highly difficult.

6.2 Conclusion

The smart contract world is still in its infancy, lacking the tools that would greatly help with both its development and interpretation by a more general audience. We have sought to create an instrument that will make the examination of smart contracts easier and more accessible to the user. We emphasized on versatility, providing both a microscopic and a macroscopic way to examine smart contracts, and our case study outlines the accuracy of our project. Furthermore, we illustrated how our tool can contend with the others currently available, and presented a promising image of its execution on a small part of the Ethereum blockchain.

Whilst there are certain limitations with our design, we believe that it can serve as a sound foundation for further expansion. Added support for the inspection of non-storage variables would be a great first start, adding finer granularity to the display process. In addition, there is potential for integration of security auditors, to flag and visualize transaction loophole breaches, such as reentrancy, flash-loan attacks, etc.

ABBREVIATIONS - ACRONYMS

ABI	Application Binary Interface
API	Application Programming Interface
AST	Abstract Syntax Tree
CBOR	Concise Binary Object Representation
DFS	Depth First Search
EVM	Ethereum Virtual Machine
GETH	Go Ethereum
JSON	JavaScript Object Notation
RPC	Remote Procedure Call

BIBLIOGRAPHY

[1] Auditless. https://www.auditless.com/.

[2] Bitquery. https://explorer.bitquery.io/.

[3] Blockchain about page. https://bloxy.info/about/show.

[4] Contract abi specification. https://docs.soliditylang.org/en/v0.7.5/abi-spec.html.

[5] Ethereum json-rpc api. https://eth.wiki/json-rpc/API.

[6] Geth debug trace transaction. https://rdrr.io/cran/gethr/man/debug_traceTransaction.html.

[7] Json-rpc specification. https://www.jsonrpc.org/specification.

[8] Oko contract explorer. https://oko.palkeo.com/.

[9] Remix. https://remix-ide.readthedocs.io/en/latest/#.

[10] Script. https://en.bitcoin.it/wiki/Script.

[11] Solidity - layout of state variables in storage. https://docs.soliditylang.org/en/v0.7.5/internals/layout_in_storage.html.

[12] Solidity compiler output description. https://docs.soliditylang.org/en/v0.7.5/using-the-compiler.html#output-description.

[13] Solidity contract metadata. https://docs.soliditylang.org/en/v0.7.5/metadata.html.

[14] Solidity contracts. https://docs.soliditylang.org/en/v0.7.5/contracts.html.

[15] Solidity optimizer. https://docs.soliditylang.org/en/v0.7.5/internals/optimiser.html.

[16] Solidity readthedocs page. https://docs.soliditylang.org/en/v0.7.5/.

[17] Solidity source mappings. https://docs.soliditylang.org/en/v0.7.5/internals/source_mappings.html.

[18] Truffle suite. https://www.trufflesuite.com/truffle.

[19] Pierluigi Cuccuru. Beyond bitcoin: an early overview on smart contracts. *International Journal of Law and Information Technology*, 25(3):179–195, 2017.

[20] Ryan Farell. An analysis of the cryptocurrency industry. 2015.

[21] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186. IEEE, 2019.

[22] Michael Nofer, Peter Gomber, Oliver Hinz, and Dirk Schiereck. Blockchain. *Business & Information Systems Engineering*, 59(3):183–187, 2017.

[23] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.