City University of London

BSC(Hons) Computer Science

IN3007 Project Report

Decompiling Ethereum EVM Bytecode for Static Analysis.

By Gera Jahja

2021-2022 V1.4

Decompiling Ethereum EVM Bytecode for Static Analysis.

Prepared by: Gera Jahja, Computer Science BSC

Prepared for: City UOL, Department of Computer Science

Email: gera.jahja@city.ac.uk

Consultant: Martin Nyx Brain

Academic Client: Martin Nyx Brain and Michał Król

Version Control:

- ⇒ V1.0: 16th Feb 2022: Introduction and Project Definition sections added
- ⇒ V1.2: 20th March 2022: changed layout, added chapters 2-5 and drafted first three/four chapters
- ⇒ V1.3:30th March 2022 documented chapters 2-5
- ⇒ V1.4: 16th May 2022, finalised report

This project is an academic proposal supervised by Martin Nyx Brain and Michał Król. The project will be to write a decompiler so that it can convert EVM byte code into C that can be handled by the CPROVER tools.

Acknowledgements

I would like to thank Mr Martin Nyx Brain for his unbiased support over the three years of my degree as a personal tutor, and the aid he provided me with towards completing this project. Without his help this project would not have gone as smoothly as it has, and I am great full to have worked with him in looking into this interesting problem. I would also like to thank Mr Michał Król for advising me on the scope and development of the project. Finally I would like to thank my Family, for their understanding and encouragement during the development of this project. I would like to especially thank my Mother Ola for always being there to listen to talk about this project, even if she didn't understand what I was talking about, and always reassuring me.

Table of contents

Chapter 1

Introduction

Ethereum is one of the most exciting block chain technologies as it is not just a cryptocurrency but also supports smart contracts. These are programs that are written in a variety of programming languages and compiled to EVM, a byte-code format like JVM, before they are run on the block chain. The security of these contracts is vital as they can control significant amounts of cryptocurrency.

1.1.1 Description of the problem

This project requires me to translate EVM byte code to C code. The purpose of this translation is to see whether we can detect bugs using CPROVER tools. If we can successfully use tools used to detect C code bugs on EVM this means we can add verification to EVM (as well as aiding our understanding of the behaviour of the smart contracts) and ensure that the Ethereum currency that is associated with the byte code is protected and less viable to hacking.

Decompiling is a part of reverse engineering, I will be using this approach to convert EVM byte code to op code, and then generating C code from this.

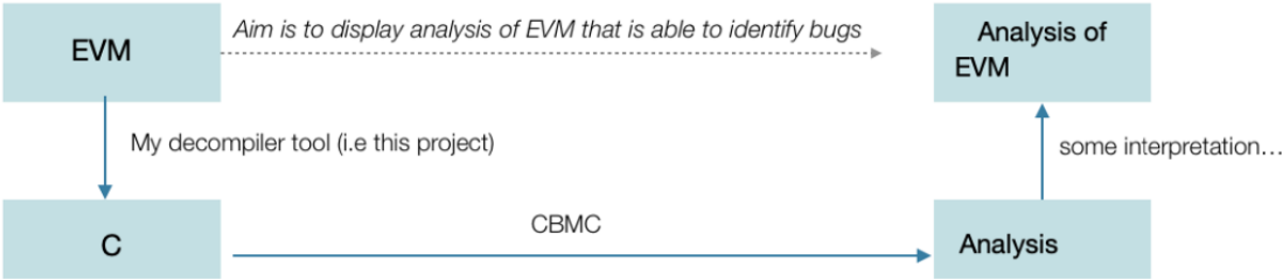
When looking to verify whether the software works, I will be specifically be looking at answering:

Will the program crash? Can it be hacked? Can we do it without running the original program? (Static Analysis, i.e., using CPROVER)

1.1.2 Project Objectives:

- Produce a subset of EVM byte code that has been decompiled to C code.
- Develop a program that displays this decomplication
- Ensure this subset (of EVM byte code) can be run through the CPROVER tool CBMC
- Investigate to what degree do these tools allow us to analyse the EVM contracts, i.e. is it able to aid us in detecting bugs?

These objectives will follow this work plan:



1.1.3 Sub Objectives

In order to meet the requirements for the main objective , The sub objectives of the problem were…

- Thoroughly understand each EVM bytecode and their respective Opcode.
- Learn how to approach Decompilation and understand reverse engineering.
- Learn the connection between high-level programs and byte code.
- Learn about the EVM in deep detail.
- Understand how the EVM processes Bytecode
- Understand the Semantics of each Opcode.
- Develop methods to sort bytecode into these individual Opcodes.
- Learn how to program in C.
- Using Knowledge of programming in C, develop equivalent methods for each EVM bytecode.
- Using Knowledge of Java programming, develop software to decompile the EVM byte code, and produce the relevant C code as an output.
- Produce analysis of each opcode working.
- Produce analysis of the C code being executable.

1.1.3 Project Beneficiaries

Beneficiaries of the project include:
My clients and I:

- Martin is quite interested in the verification end of the problem we are trying to solve, while Michael is interested in the Ethereum cryptocurrency side of things. My aim is to pave a connection between the two.

Academics:

- The findings of this development may be of interests to academics in similar fields of work some examples include :
  Cyber Security, Decomplication EVM bytecode, reverse engineering and more.

People using blockchain:

- If we can validate smart contracts, it can prevent huge financial losses for people that are part of the Ethereum blockchain.
- In the past there have been times where hacking into the currency has led to cryptocurrency being destroyed. In 2017, “$300m of cryptocurrency was lost after a series of bugs in a popular digital wallet service led one curious developer to accidentally take control of and then lock up the funds” Locating these bugs can prevent similar cases from occurring, adding more security to the currency and more protection for people using block chain.

In general, building tools that are useful is the main benefit of this project. Identifying and ideally preventing bugs with evidence that it works would be the ideal outcome of this project.

1.1.4 Work performed

In order to achieve the objectives and sub-objectives, a lot of time was spent developing the appropriate C code for each instruction. I had no previous experience with C or EVM, so this

was all new to me. Every development was discussed and tested with the consultant, and advice was given to produce two possible outputs, one that is executable in c , and one that is analysable with CProver. The difference between the two is that the EVM processes its values as 256 bits per word. This is not compatible for executable c , so an executable product must use a 64-bit integer instead. However an analysable program (one that is checked with Cprover) can have a 256-bit integer . The project has a switch between an executable and analysable, i.e. it gives the user an option to make runnable code or Cprover code.

The project was split into two sections, disassembling opcodes and decompiling the disassembled opcodes. These sections were developed using the iteration development cycle , since the disassembly is closely linked to the Decompilation. Due to this link, the iteration development cycle was followed, since each stage of this project built upon a previous requirement. The emphasis of this project is the design of the c code that is decompiled from EVM bytecode, and its analysis. Finding bugs in the decompiled code means there are errors in the bytecode itself, which can be useful in developing the security of the Ethereum blockchain. Analysis of all the tests conducted on each opcode allow for further development on this project if it is found to be relevant to locating bugs in EVM smart contracts.

Since accurate decomplication is very complex and considering the time it took to develop c code for each instruction, the scope of the project was slightly limited when decompiling certain opcodes. Decompilation to the exact previous high-level code (for example, bytecode can be produced when compiling a high-level programming language like Solidity) is near impossible. This is because when high-level programs turn into bytecode, they are simplified greatly and omit a lot of detail. This project displays every opcode's gas consumption, and their disassembly. However certain opcodes do not have equivalent C methods. For example, some op codes require access to the EVM's storage ,accounts on the Ethereum blockchain and memory, which are not accessible in C as these are environmental variables. Programming these in C would require mapping several systems together , creating a blockchain, etc. Therefore 11% of the opcodes only have limited C code, but a big portion has been modelled regardless. The omitted c codes can be seen in the testing tables in the Appendix and are labelled "out of scope". This project does model the EVM stack, memory, storage and accounts , but not at the complexity of the actual EVM since I cannot access their memory ,storage and account system.

Chapter 2

Project Outputs

Project Definition Document

Output	Project Definition Document
Description	Document that outlined the original set of requirements, project objectives, work plan and project risks that were requested by my academic client
Type of Output	Pdf Document
Recipient	Academic Client and any audience of the project report
Usage	Used to approve my project prior to development
Results Location	Appendix A and on USB as PDD_GeraJahja_final.pdf

EVM Decompiler

Output	EVM Decompiler
Description	A fully functional software product that decompiles EVM byte code to relevant C Code . Approximately 5350 lines of Java code (including comments) , this number does not count the sha3Hash package code, which was not written by me.
Type of Output	This product is built as -- Java classes, amounting to --- lines of Java code (counting comments), of which --- classes were written by me and the rest are reused from the sources indicated in the Results section. The complete software plus its design documentation can be found in the USB memory stick I provided as offline submission, or on GitHub at the URL : https://github.com/kristielGJ/DecompilingEVMtoC
Recipient	Academic Client and any audience of the project report
Usage	Used to decompile EVM bytecode to c code
Results Location	USB or https://github.com/kristielGJ/DecompilingEVMtoC

Testing of Disassembly of the EVM byte code Table

Output	Testing of Disassembly of the EVM byte code
Description	Automated testing in java that checks whether the EVM code has been disassembled, if these pass then the EVM is ready to be decompiled to .
Type of Output	Documented results of running the code in table format
Recipient	Academic Client and any audience of the project report
Usage	To track whether the results are accurate and map correctly to the Ethereum yellow paper.
Results Location	Appendix B

Testing of Disassembly of the generated C code Table

Output	Testing of generated C code
Description	Generated test coverage, back-to-back testing where I have generated line /path coverage and then taken those inputs to run on the original EVM code to see whether they are valid.
Type of Output	Documented results of running the code in table format
Recipient	Academic Client and any audience of the project report
Usage	To track whether the results are accurate and executable
Results Location	Appendix B

Deployment Guide

Output	Deployment Guide
Description	A guide explaining how to run my project
Type of Output	ReadMe on GitHub
Recipient	Academic Client and any audience of the project report
Usage	Used to see the different ways in which the program can be run.
Results Location	Appendix B

Software Documentation

Output	Software Documentation
Description	A report showing the development of this project
Type of Output	Pdf file
Recipient	Academic Client and any audience of the project report
Usage	Documentation of development of this project
Results Location	This document

Meeting Records

Output	Meeting Records
Description	A document stating all the client meetings and the topics discussed.
Type of Output	Pdf file
Recipient	Academic Client and any audience of the project report
Usage	Tracking the discussions and decisions of what is required for implementation is noted in this document.
Results Location	Appendix B, and pdf named meetingNotes.pdf on USB or GitHub

Folder of computed test results

Output	Test results
Description	A folder with bytecode . txt files, and the c. files they generated from being loaded into the decompiler java code
Type of Output	Folder of .txt and . c files
Recipient	Academic Client and any audience of the project report
Usage	Each bytecode .txt file has a .c file that is an output from loading the bytecode file into the software product.
Results Location	USB file or GitHub, folder labelled “tests”

Chapter 3

Literature Review

Ethereum is a popular blockchain platform that supports full-featured smart contracts. Developers usually write smart contracts in Solidity, which is compiled into low-level Ethereum VM (EVM) bytecode for the blockchain's distributed virtual machine. This bytecode is the ultimate semantics of the contract.

What is the Ethereum Virtual Machine?

Ethereum uses blockchain technology and is best known for its native cryptocurrency. The blockchain supports the development of smart contracts , which are processed by the Ethereum virtual machine.

EVM (Ethereum Virtual Machine) is a stack-based interpreter of EVM bytecode. EVM does not have any "system interface" handling or "hardware support"—there is no physical machine to connect with, it is completely virtual.

The EVM is a Turing-complete state machine, because all execution steps are limited to a finite number of computational steps. In comparison, on Bitcoin (another popular digital currency)the Stack Machine is a Turing incomplete machine. For the EVM, all execution processes are limited to a finite number of computational steps by the amount of gas available for any given smart contract execution. Every opcode (for example ADD) has a gas cost, and contracts can be set a gas limit.

EVM was designed in a stack-based architecture, with its word size being 256-bits. The components that stored information on the EVM are divided into 3 parts:

- immutable program code ROM, loaded with the bytecode of the smart contract to be executed
- volatile memory, with every location explicitly initialized to zero
- permanent storage that is part of the Ethereum state, also zero-initialized

The EVM opcodes are listed below, and a detailed table in the Appendices with their respective EVM bytecodes (see "Table of opcodes" in the Appendix):

Mathematical processing
ADD
MUL
SUB
DIV
SDIV
MOD
SMOD
ADDMOD
MULMOD
EXP
SIGNEXTEND
SHA3

Register interactions
STOP
JUMP
JUMPI
PC
JUMPDEST

Command to interact with block
BLOCKHASH
COINBASE
TIMESTAMP
NUMBER
DIFFICULTY
GASLIMIT

System commands
LOGx
CREATE
CALL
CALLCODE
RETURN
DELEGATECALL
STATICCALL
REVERT
INVALID
SELFDESTRUCT

Logic
LT
GT
SLT
SGT
EQ
ISZERO
AND
OR
XOR
NOT
BYTE

Environment
GAS
ADDRESS
BALANCE
ORIGIN
CALLER
CALLVALUE
CALLDATALOAD
CALLDATASIZE
CALLDATACOPY
CODESIZE
CODECOPY
GASPRICE
EXTCODESIZE
EXTCODECOPY
RETURNDATASIZE
RETURNDATACOPY

Command to interact with block
BLOCKHASH
COINBASE
TIMESTAMP
NUMBER
DIFFICULTY
GASLIMIT

Interactive Commands with stack
POP
MLOAD
MSTORE
MSTORE8
SLOAD
SSTORE
MSIZE
PUSHx (where x is between 1-32)
DUPx (where x is between 1-16)
SWAPx (where x is between 1-16)

What is Decompilation

Decompilation is the process of converting object code (i.e. executables) to a form of high-level programming that is readable by humans. In this project, the Ethereum virtual machine compiles hexadecimals that all correspond to commands (for example, 01 is the ADD command). This research project converts these commands to a C program. The C program must be executable since we will be feeding it through CPROVER tools that look for bugs in C programs. This adds a higher complexity to the project, as I will need to model the Ethereum virtual machine in C code so that it will run, and the generated code cannot be pseudo code.

Decompilation is a type of reverse engineering that performs the opposite operations of a compiler, as shown by the diagram bellow. Decompiling goes from high-level to low-level programming languages, whereas compilers convert high-level programs to binary, thus fitting the name “reverse” engineering. Compilers allow high level languages like solidity and C to be understood and run by the computer.

This project is a decompiler, which translates EVM byte code to C.

C Prover Tools

CBMC is a Bounded Model Checker for C and C++ programs. It supports C89, C99, most of C11 and most compiler extensions provided by gcc (GCC is The GNU Compiler Collection – a compiler produced by the GNU that can be used for C code) and Visual Studio.

“CBMC verifies memory safety (which includes array bounds checks and checks for the safe use of pointers), checks for exceptions, checks for various variants of undefined behavior, and user-specified assertions. The verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure."(- https://www.cprover.org/cbmc/)

This project will allow my client to investigate whether these tools are helpful in identifying any bugs in EVM byte code, but requires the code to be in C, which is the output of my software.

State of the art for Verification of EVM

EVM is binary, and there exists a high-level programming language called Solidity that allows for easier human readable development of EVM. I investigated what else is out there for that top link. Currently, verification of EVM and Solidity (“competitors” of my approach) that directly do verification include :

- Decompilers that translate EVM to pseudo solidity code(such as https://www.ethervm.io/decompile)
- Decompilers that extract information from EVM bytecode (e.g. https://www.npmjs.com/package/evm)
- Decompilers that create human readable code and aim to look for vulnerabilities (https://www.trustlook.com/services/smart.html, https://github.com/nevillegrech/gigahorse-toolchain)
- IDE’s that allow people to program using high level languages such as solidity, and compile contracts without having to worry about the byte code itself (for example, remix, truffle , etc.)

Generally identification of obvious bugs (i.e. syntax errors) can be found by IDEs such as truffle and remix. However when it comes to the bytecode itself, it may be useful to test the logic of the bytecode itself, to see whether there are underlying issues with the EVM commands that are generated. Also, some developers of smart contracts do not use IDEs and instead program solidity using Inline Assembly (the language is called Yul). This project would be very useful for that purpose.

Other Decompilers that have similar elements

JVM decompilers were studied for comparison, as JVM and EVM can technically be compared (as they are both binary i.e., executables) and existing research used for decompiling JVM was useful during the research phase of my project.

The JVM was designed to provide a runtime environment that works irrespective of the underlying host OS or hardware, enabling compatibility across many systems. This is also the reason I decided to program my software in Java. High-level programming languages such as Java are compiled into the bytecode instruction set of their respective virtual machine. In the same way, high-level smart contract programming languages such as Solidity are compiled to the EVM's own bytecode.

Unlike JVM bytecode, EVM does not have representation of structs , methods or objects in its bytecode, which makes it difficult to decompile. In JVM bytecode, stack depth is fixed under different control-flow paths, meaning that execution cannot reach the same program point with different stack sizes. In the EVM bytecode, these execution constraints do not exist, which make the detection of control-flow constructs very difficult, which limits the scope of certain opcodes being decompiled. (Control-flow constructs relate to CFG composition, which is discussed later in the literature review.)

All control-flow edges (i.e., jumps) are to variables, not constants. The destination of a jump is a value that is read from the stack, making it difficult to determine what's happening with the byte code. In contrast, JVM bytecode has a clearly defined set of targets for every jump.

JVM bytecode has defined method invocation and return instructions. In EVM bytecode, function calls inside a contract are translated to just jumps. To call a function, the code pushes a return address to the stack, pushes arguments, pushes the destination block's identifier (a hash), and performs a jump (which pops the top stack element, to use it as a jump destination). To return, the code pops the caller's identifier(in the basic block) from the stack and jumps to it.

Type of algorithms and architecture required

The best way to approach the modelling of this problem would be to use visitor methods , as I will be programming using Java. Language processing is a highly related topic that uses the same method. This will be the design pattern I will be primarily using when processing the EVM bytecode. The purpose of a visitor pattern is to define a new operation without introducing the modifications to an existing object structure, therefore I can safely access parts of my objects without altering them. This requires me to add functions that accept a visitor class to each element of the structure of my program.

The positive side of this is that there will be no dependency on my component's interfaces, and if they are different, that's fine, since I will have a separate algorithm for processing per concrete element. These elements will be each opcode, and their respective details. However it requires me to make a visitor function for each opcode, which is a lengthy process- but the safety of using visitors rather than directly interfering with my objects is worth it.

The main problem that I am investigating is to see whether CPROVER tools can identify problems with EVM byte code, which if successful could aid in increasing the security of smart

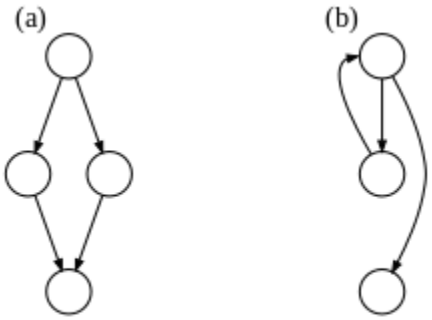
contracts. The Ethereum cryptocurrency has had issues with security due to malicious contracts, since the currency is stored on a blockchain which is open to everyone. Any improvement in the protection of the currency can save people a lot of money, as it will prevent people manipulation smart contracts that allow access to other people's accounts.

This open nature smart contracts make the analysis and validation of contracts essential, therefore it will be useful to see if we can use existing tools (i.e. CPROVER) used for other languages. The task of Decompilation is hindered by the low-level stack-based design of the EVM bytecode that has hardly any abstractions as found as in other languages, such as Java's virtual machine. For example, there is no notion of functions or call. This means that a decompiler that translates EVM bytecode needs to invent its own conventions for implementing local calls over the stack.

The difficulties of the project are that once a contract is in EVM byte code, the reconstruction back to the original program is near impossible, as the language is simplified so that the program can be run by the computer. Furthermore, converting this to C (which is a completely different language that does not have connection to EVM) requires me to develop equivalent programs for each EVM command. The first step towards this is to simulate the EVM stack itself. If I was to attempt to reconstruct more than this, it would require extensive knowledge of creating a CFG graph.

A control-flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. This is usually done by hand, and can aid in identifying uses of if statements, loops and other high-level programming techniques. The way to do this would be to split the bytecode into blocks where JUMP commands occur and to draw out the edges that connect these jump commands. This will correspond to an existing graph that symbolises either an if statement, while loop etc. In theory this sounds like a simple task, however programming this concept is extremely difficult, and a lot of research projects have been based around accurate CFG reconstruction.

See some examples of CFG graphs to the side, a) representing an if -then-else statement, and b) displaying the relevant edges of a while loop:



Chapter 4

4.1 Methodology

4.1.1 Iterative Development

I have decided to use an Iterative development methodology given the time I have to complete this project.

The project was split into three builds and had **numbered objectives** for each:

Minimum Viable Product: Completed in March:

- Researched all existing implementations of similar projects (4 days)
- Write-up report
- The program took take some EVM code and decompile it , translate into a list of Op code (2 weeks)

- Get client feedback, test that this product has no syntax or logical errors. (Client proposed the method could be structured better , these changes were made.)

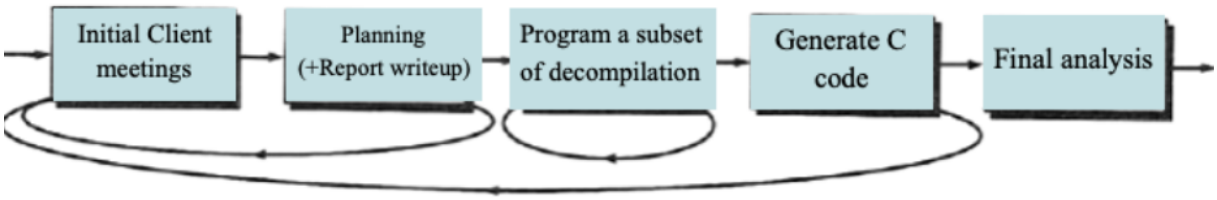
The Main Product: functional but with no extra subsets of op code: Complete by mid-April :

- Extend Report with updates (3-4 days)
- The program will take the decompiled Op code and generate reliable C code AND will successfully be usable by CPROVER tests (3+weeks)
- Get client feedback, test that this product has no syntax or logical errors. (2-3 hours)

Additional Features: Complete towards the start of May:

- Look at the behaviour of the decompiled C program and add additional features based on the main product's current output(1+week(s))
- Add additional op code features (extending the decompiler and code generator made in the first two iterations) (2+ weeks)

Refer to this iterative workflow diagram – Each stage went through testing or a client meeting before moving on to the next step, to ensure the results produced were relevant and useful. This method was effective, as every set of opcodes designed were then checked the client, corrected, and then the project could progress to the next subset of opcodes.



Given that there only exists EVM decompilers and no previous decompilers of EVM to C, this required invention of equivalent methods for each functionality found in the EVM. And since the decompiler was programmed using java, the same semantics were needed for parts of the code generation. This means for each op code I was required to understand how it worked in EVM, then produce the same in C and understand it well enough to program it in java also (for example, understanding the use of bytes, mapping, stack manipulation, hashing and more).

The only code that was not written by me was the sha3Hash package. This package takes a byte array and creates a SHA3 Hash. This was used for one of the Opcodes, named "SHA3", which takes the last two values from the EVM stack and computes its SHA3 hash. Given that the creation of c code for each opcode was time consuming, my client advised me to use an existing implementation rather than attempting to do it from scratch, as it was only needed for one out of 70+ opcodes.

4.2 Disassembling EVM

4.2.1 Analysis

This was the first stage of the project, necessary in order to later extend this code and produce a decompiler. Majority of this iteration was spent researching what was required for the disassembly and what each opcode did. The Literature review was very relevant during this stage, as the visitor methods discussed were first implemented in this iteration.

Ethereum's yellow paper was used to produce a table of each stack number (hexadecimals (i.e. the stack column) corresponding to their respective EVM binaries.) The table was

updated with the production of each class, visitor function and successful disassembly of each opcode (see appendix b.) The yellow paper lists all the op codes, and details on their uses.

4.2.2 Design

My client and I decided to use a terminal to run the program, given that Cprover is also run-in terminal, this allows quick access to each, and allows more time to be spend on the actual Decompilation of the EVM byte code.

Given that Java programs are platform-independent and can be easily compiled and run-in terminal, I chose to develop the decompiler using it. All that is required to run the program is the device must have a JRE (Java Runtime Environment). Furthermore, Java produces class files, which was a perfect fit for the project, given that the program implements an OOP design.

The plan was to use visitor functions, like language processing, to easily access objects while not interfering with any of the data. This was achieved through the usage of Classes, Interfaces and visitor functions – an Object-Oriented Design. For each opcode, a separate class was programed with details such as its name, its opcode, memory addresses and more. T Interfaces and visitor functions were then used access and display relevant data.

4.2.3 Implementation

The main program was developed to take a user input, split the inputs into relevant bytecodes , and display the name of the associated EVM opcode.

Depending on the instruction, different sizes were required, for example, ADD is 01, whereas PUSH32 is 7F0000000000000000000000000000000000000000000000000000000000000000 (the 0's after 7f can be any 32-bit memory address).

I began by implementing classes for each opcode (opcodes package in the src), an interface that connected with visitor functions that accessed each class. The display of the disassembly for each opcode to the terminal was achieved by checking whether there was an occurrence of each opcode after splitting the bytecode into a list of singular hexadecimals.

The main code that displays the decompiled code and the visitor functions are all in the decompile package. The interfaces package manages and accepts visitors, whereas the opcodes package holds all the opcode classes.

The main program would then visit the functions with an object instance of the opcode object, and data would be retrieved via getter functions in the classes. When a new opcode object was created, the constructor would set up the variables, ready to retrieve in the visitors.

4.2.4 Evaluation

Testing was conducted on my Mac, and it was discovered that only a string of length 1023 opcodes could be inputted to the program. I tested on a Linux machine and did not have this issue, which suggested there was probably an issue with my laptop's Java or an install / platform issue. It was decided that in the next iteration, a save and read file method would be implemented for ease of usage, reading in a .txt file of byte code, and writing out a .c file of

the decompiled C code. All the opcodes passed the disassembly tests, allowing for the next iterations that investigated Decompilation to start.

4.2.5 Testing

Analysis of the tests conducted on the disassembly code was performed in order to deduce whether the results obtained matched the expected results (these can be seen in the test table in appendix b) . The main emphasis was ensuring that the program accurately different lengths of disassembled EVM bytecode into its relevant opcodes.

4.3 Decomplication to C

4.3.1 Analysis

The Research necessary for the Decompilation of byte code required me to learn how to program using c. Then using this knowledge I wrote code for each opcode (see testing tables).The code had to follow the same standard and classes that stored this C code were developed. These are found in the ccode package.

A sha3 package was added, that was not written by me. The sha3Hash code was taken from https://github.com/mchrapek/sha3-java and was used to compute a hash for the SHA3 opcode.

I added an additional getter and setter to all the opcode classes, named “Ccode”, that allowed me to set the c code when visitor functions were called, and return the code via a getter.

4.3.2 Design

This part of the project was mainly extending the previous classes of opcodes that passed the disassembly tests, adding a c code variable, and computing this in the visitor functions, for each individual op code.

This design prevents the creation of any new interfaces, as while we disassemble it makes sense to generate the c-code simultaneously rather than calling two separate visitors.

For memory and storage, the same stack simulation in the generated c code, would have to be implemented internally in the java code. This was needed to keep track of the address that would be required for jumps and retrieval. This meant that in the visitor functions, when disassembling the code, the following were computed:

- Print the disassembly (First iteration)
- Compute the relevant c code and return it to the main program
- Update a stack in java for ease of generation of memory and storage related opcodes.

The c code I designed for the decompiler were all tested in a compiler and in Cprover. Cprover supports the usage of 256-bit integers whereas executable c only takes 64. So, because of this the code compiled has an option to be either analyzable by c prover or able to be compiled and run.

The c code models the EVM Stack, using a list and pointer for the top of the stack. Then later iterations allowed me to increase the subsets of opcodes, and model the memory, which is an array of pairs (pairs defined as a struct), with the location and value stored for each memory

position. The storage and accounts followed a similar model. This means for the project to succeed I had to understand EVM, C and Java, to ensure the Decompilation was accurate.

Each opcode has its own c label, so that jump instructions can be added (i.e., "*goto label_0"* in C is a jump to that label).

4.3.3 Implementation

The generation of C code from the computed opcodes was split into the following parts (which all had their own classes in the cCode package in the src):

- Modeling the EVM stack in C
- Modeling of EVM gas usage in C
- Modeling the EVM accounts system in C
- Modeling the memory in C
- Modeling jumps and the pc counter etc. in C

The visitor functions now disassembled and return generated c code, with labels that increment, and the main program stores the generated c code in an array. This ensures the order of the decompiled c is exactly like that of the disassembly.

The main code is found in the decompile package, that was redesigned. Now the decompile folder has a EvmDissasemble.java class that contains all the methods needed for the previous iteration. The main code that displays the decompiled code and the previously implemented disassembly is found in the EvmDecompile.java class.

Since EVM and C have different compilers and syntax, this meant that invention of equivalent methods was needed. As aforementioned, there are no other EVM to C Decompilers, this meant that I was required to spend time on each opcode, develop an equivalent program in C for them, while also ensuring the rest of the code follows the same design and logic.

The program gives options to either input the EVM bytecode directly in terminal or read it in as a text file, The Outputs the code also gives an option to save as a c file, for ease of CProver testing, as Cprover requires you to enter the file you want to validate.

4.3.4 Testing

Testing was conducted during and after development. Errors in the generated C code symbolize errors in the bytecode as each c code that was implemented passed compiler tests prior to them being added to the project.

Therefore, if the C code does not run, that is due to the order of the byte code fails the generated c assert statements, that makes sure the stack is not empty, over fills and that ensures the gas used is never more than the gas limit. This successfully decompiles a large subset of the code, excluding opcodes that require out of scope development such as accessing accounts in the EVM. This is discussed in detail in the results section of the report.

4.3.5 Evaluation

Analysis of the tests conducted on the whole system was performed in order to deduce whether the results obtained matched the expected results. The solution was tested with both existing bytecode from real smart contracts as well as individual byte codes (see test tables for decompiled c)

Chapter 5

Results

Following from the methodology chapter, this section of the report investigates the developments made at each iteration, their respective results and analysis. Overall, out of 255 opcodes, only 29 are out of scope for Decompilation, meaning only around 11% of the EVM Opcodes do not produce accurate C Code. However, 100% of the opcodes are disassembled.

5.1 Disassembling EVM

Disassembly is the translation of bytecodes to opcodes. This was the first iteration and was compulsory for later Decompilation. The aim was to identify bytecode, create opcode objects when they were identified, so that opcode information, such as the name and the bytecode could be displayed. This was something that was prevalent in existing decompilers of EVM and I believed to be useful for the overall flow of the project. Identifying the opcodes was the first step towards mapping equivalent C code to each opcode. Further down in this document, you will see that personalised C code was an addition to the values stored in these opcode classes.

Using OOP and Visitor functions

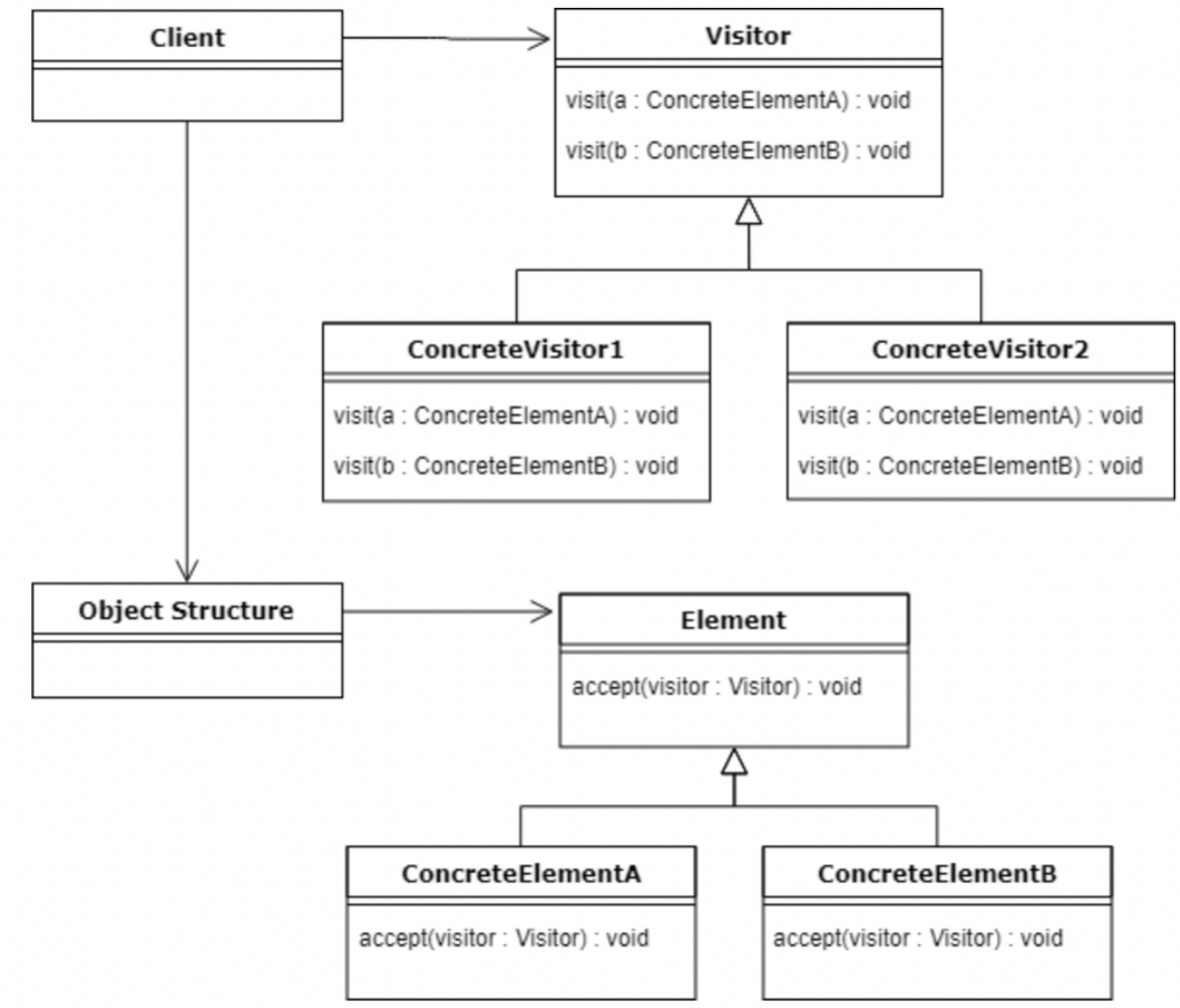
Before Visitor functions were implemented , I tested the idea by creating a simple add disassembler, where if 01 is detected, it would return 01 ADD.

Each visitor accepts an instance of an opcode object. This is then used to call getters of the opcode, which have initialised data such as their name and their bytecode number. These are concatenated and the string is displayed in the terminal when the visitor function is called. For example, if the input string contains 01 , a new add object is made, the visitor function accepts a visitor, and this visit(add Add) function is called with the add object as a parameter. The following code shows the string methods in the visit function that were used to achieve this.

```
//disassembly

System.out.println("0"+Integer.toHexString(Add.getOpcode()).toUpperCase() +
"  "+ Add.getName());//output the instruction(Disassembly), used for
Decompilation
```

Once this experiment succeeded, I did the same for every other opcode.
The following class diagram shows how the classes link together, and that this method uses interfaces to interact with classes rather than creating methods in each class and calling them. This will later allow me to generalise calls , for example, numerical functions will all have the same call to a visitor, with the only difference being the op code that is passed to it:



Disassembling Numeric Opcodes

Given that we have a visitor design, the production over 50 of opcodes meant I had to come up with an algorithm make my method more efficient. The previous method of checking if a string input "contains" an opcode meant I was going to have to write over 50 if statements, for each opcode. The opcodes are strings so that I can easily manipulate them in java, however if the string is numeric, I call their position in an array of opcode objects. For example, the stop opcode is 00, which is where the index of the object stop is stored in an array of " GetInstructionsFromOpcode[] instructions", which is the interface that accepts visitors.

The following code was used to detect whether the string was numeric or not. This code can be found in the EvmDissasemble.java class.

```
/*check is a string is a number , from :https://www.baeldung.com/java-check-string-number */
    public static boolean isNumeric(String strNum) {
        if (strNum == null) {
            return false;
        }
```

```
        try {
            double d = Double.parseDouble(strNum);
        } catch (NumberFormatException nfe) {
            return false;
        }
        return true;
    }
```

The Boolean allowed for the switching of methods depending on whether the opcode was a number. This code that could be reused for every numeric opcode .On the other hand, non-numerical opcodes required if statements per opcode. However, this method was still able to reduce the number of if statements by more that 50%, which was huge improvement.

This method was used to implement the rest of the opcodes which had a length of 2. At this stage of the development of disassembly ,all opcodes apart from push, dup and swap were implemented.

Disassembling Push, Log, Dup and Swap Opcodes

The Push, Log, Dup and Swap opcodes had one thing in common, a number ranging from 1-16 for dup and swap, 1-32 for Push, and 0-5 for Log . It was decided that an algorithm that computed the number next to push, dup and swap was needed . This was a priority because it would have been a waste of time to implement each command as a separate object. To prevent the creation of 69 extra opcode classes a prototype using the first two values of the opcode was computed by myself.

The algorithm developed took an opcode as a parameter. This was used since if the opcode began with 7 or 6 it was a push command, Log commands began with A , and any other value was considered a dup or swap. Based on the opcode, a calculated number was added to the second element of the opcode so that it added up to the number connected to the Command. For example push 1 is 60, so take 0 and add 1.

```
    /*
    Looks at second number and calculates number for log, dup,swap and push
    , i.e. for 63 (push4), it returns 3+1 = 4 (to display the number for push "4")
    also gets Lable00_ number for decompiled C code
    */
    public static int getNumber(String opcode) {

        String[] arr = opcode.split("");
        if (isNumeric(arr[1]) == true) {// integer conversion 0-9
            if (opcode.startsWith("7")) {
                return Integer.parseInt(arr[1]) + 17;// returns push 17-26

            } else if (opcode.startsWith("A")) {
                return Integer.parseInt(arr[1]);// returns log 1-4
            } else {
                return Integer.parseInt(arr[1]) + 1;// returns dup/push/swap 1-10
            }
        } else {// hexadecimal conversion for a-f
```

```
            if (opcode.startsWith("7")) {
                return Integer.parseInt(arr[1], 16) + 17;// returns push 26-32
            } else {
                return Integer.parseInt(arr[1], 16) + 1;// returns dup/push/swap
11-16
            }
        }
    }
```

Additionally, Push commands are not the same as every other opcode, since their length varies from 4 to 66 depending on how large the push number is. This meant it was not possible to split an input into an array opcodes of length 2. Once the input string was split into opcodes of size 2, a loop was used to concatenate the opcodes in the array depending on whether the first two values of a push command were identified. For example, for 60, the program would only concatenate the current index to the next item in the array, then remove it. 61 allowed for concatenation of two items in the array to 61, 62 was concatenated with 3 items, and so on.

These algorithms allowed for a less cluttered design, less code that worked more efficiently, and have an 100% pass rate (given that all the opcodes for the EVM can be identified and disassembled) All of the methods are in the decompile package, separate to the opcode classes etc.

5.2 Decompiling EVM to C

At this stage of the Project, all the disassembly is done, and all the information required for each opcode in my code was present in the project. The first change I made was to add a separate class for Decompilation and disassembly in the decompile package. From here a decision of how it was best to add the C code that I had computed for each opcode into the project, following the same OOP design.

cCodes Classes

Design

Since the EVM commands all had their own classes , I decided that the models I programmed for the C Decompilation would have their own class structure. Given that the EVM is a stack-based architecture, the first class I added to the cCodes package was stack.java. This class contained every possible stack related c code I need for the opcodes I decompiled. Following this design, the same was done for variable.java, label.java, etc. these classes only contain c code for ease of reference, so that I could program in java while outputting C for the user.

Prototypes

The first c code I developed was written with my consultant Martin. Since he has a great knowledge of c and knows what is best for CProver tools, I followed his design (using labels_, goto's and assert() statements for validation). Here is an example of a simple add stack manipulation program (top is a pointer):

```
label_0 :
 {
    int tmp1 = stack[top - 2];
    int tmp2 = stack[top - 1];
```

```
    int tmp3 = tmp1 + tmp2;
    top-=2;
    stack[top] = tmp3;
    ++top;

    assert(top != STACKHEIGHT);
}
```

For the code to work, a main program was needed to contain these labels. Therefore in the decompile package, the class EvmDecompile sets up a default main function in c , with imports, type definitions, and any declarations needed for the labels to run in the main function.

It was decided I would use the pre-existing visitor functions , as to not overly clutter my code. I added an extra value to each opcode in the opcodes package named “cCode”. This string variable was given both a setter and a getter. This was added so that when the visitor call was accessed , the visit() function would set the cCode and return this string value to the main program so that it could be stored for later display.

Other methods such as writing a label in text files and loading them in seemed too messy and inefficient. Given that I already had many classes, I did not add more than necessary for this iteration cycle. And since a lot of the c code was going to have similar functionality, it made sense to store every possibility in classes and only retrieve the code that was needed per opcode.

Arithmetic Opcodes to C

Design

Since Martin and I had decided on implementing the stack model, I noticed that all the arithmetic labels would be the same, apart from a few such as the “NOT” command. While referring to the Ethereum yellow paper, I was able to note which arithmetic commands took two items from the stack and applied an operator to these values. This allowed me to generate a function called arithmeticCodeGenerator() in the visitOpcode class. The function takes in the following parameters:

- Int lableno -used to ensure the label is always incrementing and there are never two labels with the same number
- Int no – used to ensure the tmp variables displayed are unique, a number is added, e.g tmp1,tmp2 etc.
- String operator- the operator used for tmp3 (see code in previous section)
- Int gasno – this feature is discussed later (modelling gas usage of EVM)
- Boolean unsigned - some arithmetic operations are conducted on signed or non-signed integers. For this I defined two typedefs, one signed and one not, and this Boolean decides what the variables will be defined as

Outputs from analysis

The function worked perfectly for this first subset of generated c. I was able to display the code by returning the c code from the visitor functions during disassembly, appending the returned value to an array, and displaying this array once the Decompilation was complete. Please see the appendix for a table of the c code that was generated.

Since the arithmetic c code creation was a success, the rest of the c code was written in a similar fashion, using similar models. For example, push commands simply add a value to the top of the stack, Pop decreases the stack top pointer by one , etc.

Push, Swap and Dup Opcodes to C

Design

Since the arithmetic c code creation was a success, the rest of the c code was written in a similar fashion, using similar models. For example, push commands simply add a value to the top of the stack, Pop decreased the stack top pointer by one , etc.

The dup Opcodes follow a similar pattern to the arithmetic ; however the difference is that the second variable varies based on which Dup is called. For example dup3 duplicates the value three positions down from the stack. Since C does not have stacks and I implemented this model using a list and a pointer, this was not difficult, as I simply assigned the top value of the stack , to this pre-existing element.

Similarly, Swap does the same, reassigning certain values to another value in the array using the tmp variables. Most of the opcodes developed have an interaction with the stack, so this was the first development of C code I completed for the project.

Accounts, Storage and Memory Opcodes to C

Design

Memory is a separate storage to the EVM stack, and since access to the EVM's storage and memory is unattainable at this time, I was advised to model my own "mapping" between the stack, and storage/ memory.

The actual size of the EVM's memory is a huge number (2^256) , so to make this less complex I have given the user the flexibility to change the size of the memory once they save the decompiled code as a c file.

Memory and Storage both store a location and a value. Thus I programmed a struct in c so that I was able to store two values in an array (location and value), with a memory pointer to keep track of how many spaces of active memory there are. The location value is the connection between the stack and the memory, since the memory location is decided based on the stack. Similarly, storage uses the same struct, as shown bellow:

```
struct pair {
    var location;
    var value;
};
struct numpair {
    int address;
    double balance;
};

struct pair memory[MEMORYSIZE] = {};
struct pair storage[MEMORYSIZE] = {};
struct numpair accounts[MEMORYSIZE] = {};
```

When loading values and storing them to memory and storage , it is possible to access specific values by there struct names (for example MLOAD loops through the memory and checks the location of each element). This gave me the idea to develop another struct that models accounts, since there are commands that request the account balance and address respectively. This meant the structs were a good fit and supported the opcodes well.

In the evm accounts are more complex and include hashes and “salt” for security, but for simplicity I have designed the accounts model to simply store an address and a balance. This allowed me to develop code for some sections that would have been out of scope otherwise.

Gas tracking in the EVM to C

An individual idea I had was to add Gas tracking to the code. In the evm, each opcode costs gas, some of these gas “prices” are difficult to calculate, so I researched thoroughly so that I could add gas validation to the code. If a user runs out of gas they cannot make any more transactions, which has been implemented in c using gas checks (i.e. assert statements for CPROVER compatibility.)

Each opcode increments the gas value, even if it is out of scope, unless the gas cost is unknown. This allows for accurate gas prediction, which is a useful tool when developing smart contracts. The assert statements used are below:

```
gasUsed +=3;
assert(gasUsed < gasLimit);
```

The addition of the feature allows users that are not interested in C to find some interest in the outcomes of this project, as this feature can help for planning how much gas to use for your smart contracts (as gas can equate to real money).

5.3 Reading and writing to files

The user has an option to read in a .txt file rather than typing/ pasting the byte code as an input. This was added to accommodate for the large bytecodes created for real contracts. This was implemented with a Scanner and is surrounded with a try -catch statement in the case of the user entering the wrong text file name. This feature in in the EvmDecompile.java class where all methods related to displaying the decompiled code is stored.

```
 ____________________________________________________________________
|     EVM BYTE CODE -> C DECOMILER                                   |
|                                                                    |
|        Type 'EXIT' to quit  the program                            |
|        Type 'FILE' to read a txt file containing your byte code    |
|        or enter your Ethereum contract bytecode (e.g 60806040...)  |
|____________________________________________________________________|

file
Enter name of .txt file:

wrongname
There is a problem with your file
Enter name of .txt file:
```

The save file feature allows for saving to a .c file, since if real smart contracts are decompiled, there can be hundreds of lines of c that are generated, which is easier to manage in a .c file. Furthermore, CProver reads in .c files, so this feature is mainly to allow for CProver testing.

```
 ____________________________________________
|     Save to file                           |
| Options:                                   |
|        1. Save decompilation to a .c file  |
|        2. Decompile a new contract         |
|____________________________________________|

Please choose an option from above:
```

5.4 Analysable vs executable C code

Once the user inputs their bytecode, they are prompted on whether they want to generate Executable C code or Analysable C. The difference between the two is that one supports 128-bit values, which is the main reason this c code cannot be compiled normally in c, since some push values are too large. Cprover supports 128-bit values so the c programs will compile when my clients want to look at the analysable code. The executable option is best for when the stack stores smaller values, such as Push 1-10.

Terminal:

```
 ____________________________________________________________________
|     EVM BYTE CODE -> C DECOMILER                                   |
|                                                                    |
|        Type 'EXIT' to quit  the program                            |
|        Type 'FILE' to read a txt file containing your byte code    |
|        or enter your Ethereum contract bytecode (e.g 60806040...)  |
|____________________________________________________________________|

60006001
Processing...
EVM BYTECODE: 60006001


 ____________________
|     C Code Decompiler    |
| Options:                 |
|        1. Executable C   |
|        2. Analysable C   |
|__________________________|

Choose an option from the menu above:
```

Difference in the Generated C:

Analysable	Executable
`typedef int128_t var;` `typedef int128_t uvar;`	`typedef int64_t var;` `typedef uint64_t uvar;`

```
label_1 :
{
    stack[top] =  0x10000000100000001;
    ++top;
    assert(top != STACKHEIGHT);
    gasUsed +=3;
    assert(gasUsed < gasLimit);
}
```

```
> make -s
main.c:53:19: error: integer literal is too large
 to be represented in any integer type
    stack[top] =  0x10000000100000001;
                  ^
1 error generated.
make: *** [<builtin>: main.o] Error 1
exit status 2
>
```

Figure 1Example of large number in A normal C compiler , where int_64 == var

5.5 Label and Variable name generation

I added a feature that allowed label_ and tmp variable names to be unique, by appending a number that incremented per opcode for the label, and a local variable that was updated for the tmp variable numbers. This was achieved simply by adding an additional parameter (order number) to the visitor functions. This number is the index of the loop that computes the disassembly and Decompilation of c , therefore it will always be in order.
As shown by the c code snippet bellow (This is an OUTPUT of my software, an example of the decompiled code) , the labels and variables generated are never the same:

Input: 60006001016002

Output: (Just the main function)

```
int main (int argc, char **argv) {

 /*** Start of decompiled code ***/

 label_0 :
 {
    stack[top] =  0x00;
    ++top;
    assert(top != STACKHEIGHT);
    gasUsed +=3;
    assert(gasUsed < gasLimit);
 }
 label_1 :
 {
    stack[top] =  0x01;
    ++top;
    assert(top != STACKHEIGHT);
    gasUsed +=3;
    assert(gasUsed < gasLimit);
 }
 label_2 :
 {
    uvar tmp1 = stack[top - 2];
    uvar tmp2 = stack[top - 1];
    uvar tmp3 = tmp1 + tmp2;
    top-=2;
    stack[top] = tmp3;
    ++top;

    assert(top != STACKHEIGHT);
    gasUsed +=3;
    assert(gasUsed < gasLimit);
 }
 label_3 :
 {
    stack[top] =  0x02;
    ++top;
    assert(top != STACKHEIGHT);
```

```
    gasUsed +=3;
    assert(gasUsed < gasLimit);
 }

/*** End of decompiled code ***/
return 0;

}
```

As shown above, the variables start with an extension of 1, that incremented every time a new variable object is created. This prevents errors when running the code , given that these are all variable declarations.

This c code compiled with an additional for loop to print the values shows that the logic works, see below:

Expected outcome:
(1+0) ,2,0,0,etc…

Added code for testing:

```
int i;
  for (i = 0; i < STACKHEIGHT;i++)
{
  printf("%d ", stack[i]);
}
```

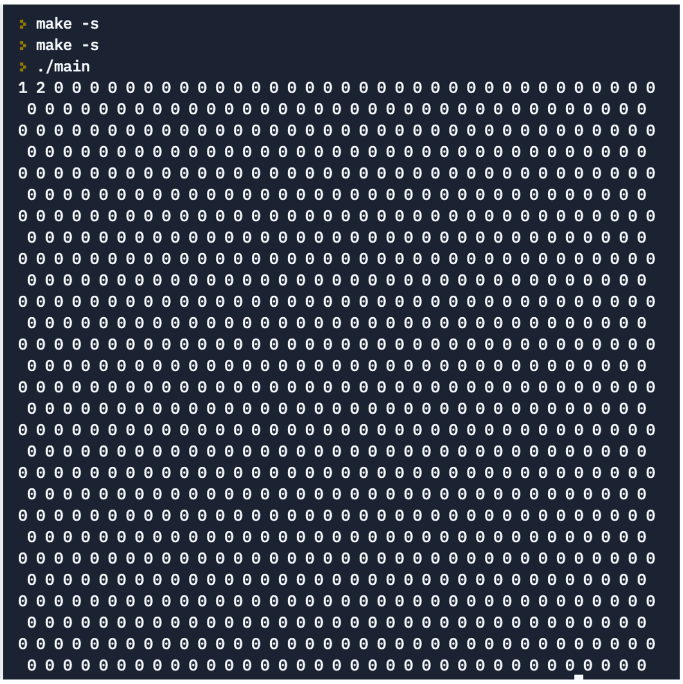


Figure 2Output: (1024 stack values)

5.6 Preventing errors

The code considers many possible ways of incorrect inputs causing the code to crash, for example, if the bytecode is an odd length, this means it is incorrect, since all bytecodes are even in number. Throughout the code I use try catch statements to ensure the program does not break. Another example is if a file does not exist, the program does not crash, it simply re prompts the user for the file name.

Figure 3 Odd numbered bytecodes do not crash the program

5.7 Real Smart Contracts testing

The opcodes were tested individually (I would mostly be using the file ALLOPCODES.txt (see SRC package), or singular/ small strings of specific opcodes) while developing.
After all the implementation was done, I tested my program with real bytecode from actual smart contracts.

The website https://aphd.github.io/smart-corpus/ is a repo with 120k smart contracts. The entire dataset was directly downloaded, and a random selection were chosen to test the project. The only alteration needed was to change the file extensions from .bytecode to .txt and to remove the 0x at the start of some files so that they were compatible with the decompiler.

In the test folder you will find that there are folders named “smartcontractx”, and the successful Decompilation of the .txt folder for both analysable (for Cprover) and executable c code , named “bytecodex.c” and “bytecodeAnalysex.c” . However these cannot be compiled like the simple tests (named simpletestx.c) as the numbers are too large for the 64bit integers limit that c has, therefore these are only useful for analysing in cprover. Aside from my test tables that look at each opcode , the test folder holds examples of my results, since the code is too long to be displayed on a table or spreadsheet.

The real smart contract byte codes are easy to decompile using the read file option rather than the typing input, since real bytecode is very large , and reading files prevents lagging entering the bytecode.

Chapter 6

Conclusions and Discussion

In conclusion, the outputs of the software programmed can successfully produce a subset of EVM byte code that has been decompiled to C code. The program displays this decomplication in the terminal and can produce a .c code file with the decompiled code. The subset of opcodes that have been decompiled have been designed to accommodate CPROVER tools and are populated with analysable c code to aid in the finding if bugs in these contracts.

The main question behind this project was to Investigate to what degree do these tools allow us to analyse the EVM contracts, i.e. is it able to aid us in detecting bugs. After the work I have done, the answer is ambiguous. While these tools can help us in detecting if the byte code is wrong, if the positioning of opcodes affects the compilation, if we want to be able to discover deep hidden bugs , a direct tool is needed. For example, a c library that allows you to program smart contracts in C, or a plugin of some sort. This investigation has proven to identify problems with bytecode, but the bugs that are of high risk require connection to the accounts, the coins and the evm blockchain itself, which is difficult to do in a completely different programming language with a separate compiler.

A problem I did not expect was handling large numbers. While the c code it is compatible with Cprover tools, it would have been interesting to have come up with a method to allow big numbers to run as executable code, and not just analysable once a certain size of number has been used.

Given that not a lot of research has been looked at when it comes to the Ethereum blockchain and c ,it would be a good future project to also see if other languages are more compatible for Decompilation from evm byte code, compared to C. The EVM stack has a lot of imperfections that could be greatly improved, considering it is responsible for a large portion of some peoples wealth.

I enjoyed working on the project, as I have never learnt about blockchains and Ethereum prior to this project. I also learnt how to program in C, while developing my knowledge of virtual

machines, bytecode and more. The project had a wide scope of possibilities , which was overwhelming at times given that there were no prior software products for me to compare my work to. Also a lot of the possibilities such as cfg's have been researched for years and are still difficult to implement. However this is also a strength, as it allowed me to develop each stage by thoroughly researching and understanding every aspect of the problem before attempting to solve it.

Unfortunately I was not able to develop function recovery, with "goto labels" being the extent of jump commands. Function recovery and CFG development is the most difficult part of Decompilation, and if I was given more than one semester on this project, I would have enjoyed doing research into that side of EVM, since these tools are not easily available for Smart contract developers. If this project was to be extended, that would be the best investigation, as it is highly needed in the industry and a product like that could make a lot of profit (financially, whilst aiding the community of smart contractors).

All in all I enjoyed working on this project, I learnt a lot about several different fields of computer science and was able to use my own knowledge to pave a link between them all.

Reference List

Harvard style referencing was used for the books, articles, videos and website links bellow:

- ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. (n.d.). [online] Available at: https://ethereum.github.io/yellowpaper/paper.pdf.
- The Guardian. (2017). "$300m in cryptocurrency" accidentally lost forever due to bug. [online] Available at: https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether
- Hollander, L. (2022). EVM Bytecode Decompiler. [online] GitHub. Available at: https://github.com/MrLuit/evm
- Ethereum Stack Exchange. (n.d.). EVM - How can you decompile a smart contract? [online] Available at: https://ethereum.stackexchange.com/questions/188/how-can-you-decompile-a-smart-contract
- ethereum.org. (n.d.). Introduction to the Ethereum stack. [online] Available at: https://ethereum.org/en/developers/docs/ethereum-stack/#ethereum-client-apis
- www.cprover.org. (n.d.). The CPROVER Manual. [online] Available at: http://www.cprover.org/cprover-manual/cbmc/tutorial/
- GitHub. (2022). rattle. [online] Available at: https://github.com/crytic/rattle
- Online Solidity Decompiler. (n.d.). Online Solidity Decompiler. [online] Available at: https://www.ethervm.io/decompile

- ethereum.org. (n.d.). Ethereum development documentation. [online] Available at: https://ethereum.org/en/developers/docs/
- Aho, A.V., Ravi Sethi and Ullman, J.D. (2002). Compilers : Principles, Techniques, and Tools. Reading, Mass.: Addison-Wesley.
- OpenZeppelin blog. (2018). Deconstructing a Solidity Contract — Part III: The Function Selector. [online] Available at: https://blog.openzeppelin.com/deconstructing-a-solidity-contract-part-iii-the-function-selector-6a9b6886ea49/
- OpenZeppelin blog. (2018). Deconstructing a Solidity Contract —Part I: Introduction. [online] Available at: https://blog.openzeppelin.com/deconstructing-a-solidity-contract-part-i-introduction-832efd2d7737/
- www.youtube.com. (n.d.). Ethereum/EVM Smart Contract Reverse Engineering & Disassembly - Blockchain Security #3. [online] Available at: https://www.youtube.com/watch?v=I6VDBvX9Pkw&t=41s
- ethereum.org. (n.d.). Reverse Engineering a Contract. [online] Available at: https://ethereum.org/en/developers/tutorials/reverse-engineering-a-contract/
- FreddyCoen (2022). EVM development Starter Kit. [online] Medium. Available at: https://freddycoen.medium.com/evm-starter-kit-1790bcc992ef
- www.youtube.com. (n.d.). EVM: From Solidity to byte code, memory and storage. [online] Available at: https://www.youtube.com/watch?v=RxL_1AfV7N4
- Unix & Linux Stack Exchange. (n.d.). convert executable back to C source code. [online] Available at: https://unix.stackexchange.com/questions/229802/convert-executable-back-to-c-source-code
- readthedocs.org. (n.d.). Solidity | Read the Docs. [online] Available at: https://readthedocs.org/projects/solidity/
- TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANALYSIS OF SYSTEMS : 28th. (2022). S.L.: Springer Nature.
- Cassez, F., Fuller, J. and Asgaonkar, A. (2022). Formal Verification of the Ethereum 2.0 Beacon Chain. Tools and Algorithms for the Construction and Analysis of Systems, pp.167–182.
- Bytecode and transactions . (n.d.). [online] Available at: https://cse.iitk.ac.in/users/dwivedi/Blockchain/bytecode.pdf
- Hollander, L. (2019). *The Ethereum Virtual Machine — How does it work?* [online] Medium. Available at: https://medium.com/mycrypto/the-ethereum-virtual-machine-how-does-it-work-9abac2b7c9e.
- En.wikipedia.org. 2022. *Control-flow graph - Wikipedia*. [online] Available at: https://en.wikipedia.org/wiki/Control-flow_graph

- cypherpunks-core.github.io. (n.d.). *Chapter 13: The Ethereum Virtual Machine · GitBook*. [online] Available at: https://cypherpunks-core.github.io/ethereumbook/13evm.html#evm_architecture

- Mochi.Market (2021). *Learn about Ethereum Virtual Machine*. [online] Mochilab. Available at: https://medium.com/mochilab/learn-about-ethereum-virtual-machine-de0a574b13ba

- www.cprover.org. (n.d.). *The CBMC Homepage*. [online] Available at: https://www.cprover.org/cbmc/.

- Szczukocki, D. (2018). *Visitor Design Pattern in Java | Baeldung*. [online] www.baeldung.com. Available at: https://www.baeldung.com/java-visitor-pattern.

- Dannen, C. (2017). *Introducing ethereum and solidity : foundations of cryptocurrency and blockchain programming for beginners*. New York: Apress, Cop.

- www.programiz.com. (n.d.). *C Bitwise Operators: AND, OR XOR, Complement and Shift Operations*. [online] Available at: https://www.programiz.com/c-programming/bitwise-operators#complement

- Ethereum Stack Exchange. (n.d.). evm - How can you decompile a smart contract? [online] Available at: https://ethereum.stackexchange.com/questions/188/how-can-you-decompile-a-smart-contract.

Appendices

Appendix A: Project Definition Document

This file is named PDD_GeraJahja_final.pdf , stored on the USB. (see next page)

Project Definition Document (PDD)

Cover Sheet

Decompiling Ethereum EVM ByteCode for Static Analysis

Prepared for: City UOL, Department of Computer Science
Prepared by: Gera Jahja, Computer Science Bsc
Email: gera.jahja@city.ac.uk / g_jahja31@outlook.com
Consultant: Martin Nyx Brain
Academic Client: Martin Nyx Brain and Michał Król
Date: February 2022
This project is an academic proposal supervised by Martin Nyx Brain and Michał Król. The project will be to write a decompiler so that it can convert EVM byte code into C that can be handled by the CPROVER tools.
Word count:1280 (not including tables, diagrams and refferencing section)

Project Proposal

Problem to be solved:

Ethereum is one of the most exciting block chain technologies as it is not just a cryptocurrency but also supports smart-contracts. These are programs that are written in a variety of programming languages and compiled to EVM, a byte-code format similar to JVM, before they are run on the block chain. The security of these contracts is vital as they can control significant amounts of cryptocurrency.

This project requires me to translate EVM byte code to C code. The purpose of this translation is to see whether we can detect bugs using CPROVER tools. If we can successfully use tools used to detect C code bugs on EVM this means we can add verification to EVM (as well as aiding our understanding of the behaviour of the smart contracts) and ensure that the Ethereum currency that is associated with the byte code is protected and less viable to hacking. Decompiling is a part of reverse engineering, I will be using this approach to convert EVM byte code to op code, and then generating C code from this.

When looking to verify whether the software works I will be specifically be looking at :

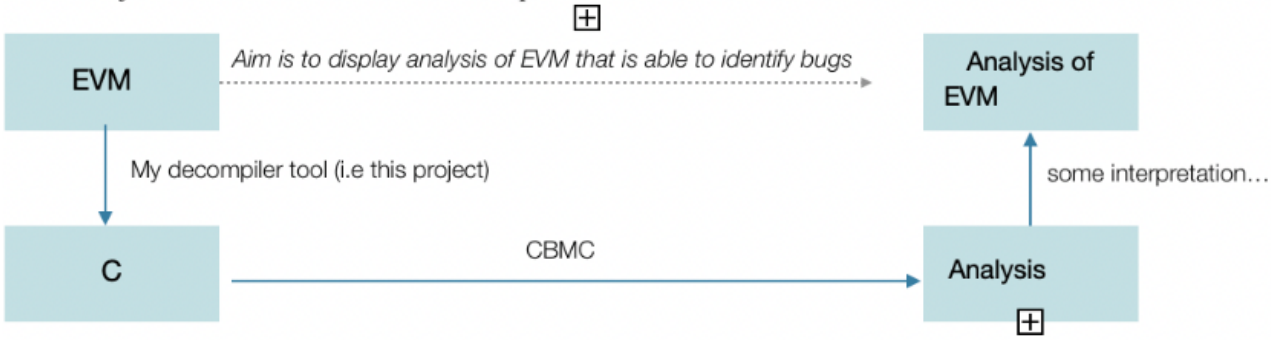
- Will the program crash? Can it be hacked? Can we do it without running the program? (Static Analysis, i.e using CPROVER)

The Verification tool for C is CBMC and there are some verification tools for EVM or Solidity, etc.

Project Objectives:

-Produce a subset of EVM byte code that has been decompiled to C code.
-Develop a program that displays this decompilation
-Ensure this subset (of EVM byte code)can be run through the CPROVER tool CBMC
-Investigate to what degree do these tools allow us to analyse the EVM contracts , I.e is it able to aid us in detecting bugs?

These objectives will follow this workflow plan:



Project Beneficiaries:

Beneficiaries of the project include:

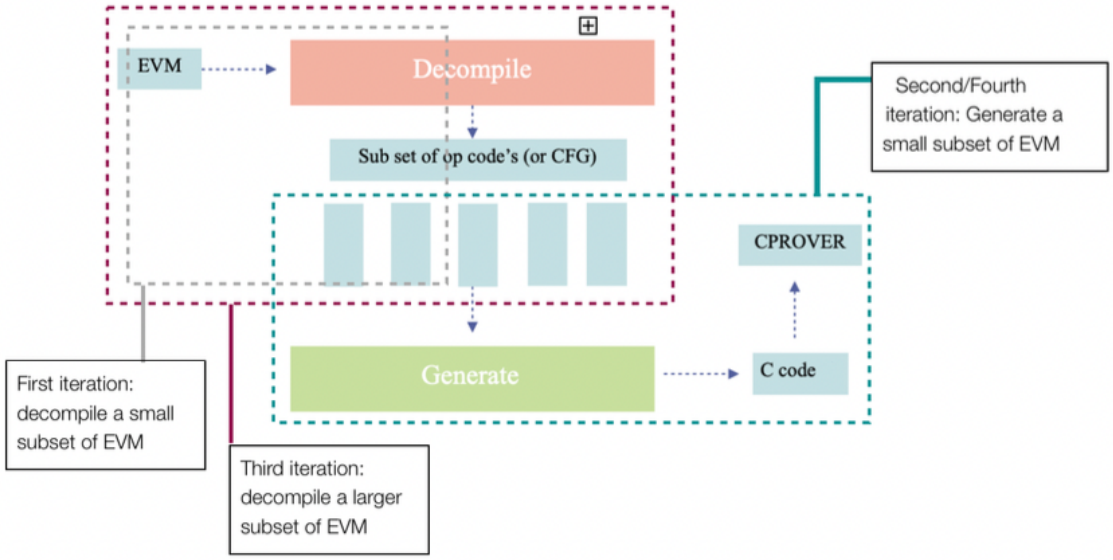
- My clients and I-Martin is quite interested in the verification end of the problem we are trying to solve, while Michael is interested in the Ethereum cryptocurrency side of things. My aim is to pave a connection between the two.

- Academics - The findings of this development may be of interests to academics in similar fields of work, such as cyber security, decompilation and more.

- People using blockchain - If we are able to validate smart contracts it can prevent huge financial losses for people that are part of the Ethereum blockchain. In the past there have been times where hacking into the currency has lead to cryptocurrency being destroyed. In 2017, "$300m of cryptocurrency was lost after a series of bugs in a popular digital wallet service led one curious developer to accidentally take control of and then lock up the funds" Locating these bugs can prevent similar cases from occurring, adding more security to the currency and more protection for people using block chain.

- In general, building tools that are useful is the main benefit of this project. Finding and ideally preventing bugs with evidence that it works would be the ideal outcome of this project.

Proposed Project structure (decided with my academic clients during a meeting on 03/02/2022) for my Main Product (To be developed using java):

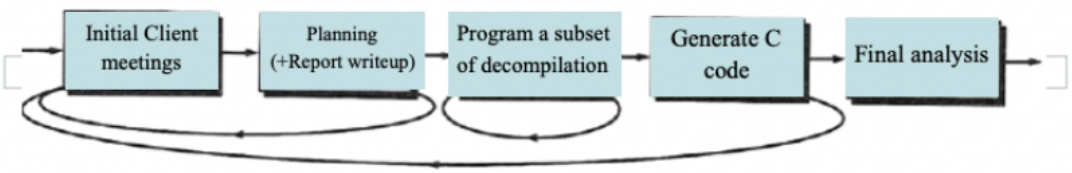


Iterative Development:

I will be splitting my project into three builds and will have **numbered objectives** for each:

1. **Minimum Viable Product** : Complete by mid-March :
   - Research all existing implementations of similar projects (2-3 days)
   - Write-up report (3-4 days)
   - The program will take some EVM code and decompile it , translate into a list of Op code (or a CGF) (2+ weeks, depending on whether the development goes smoothly)
   - Get client feedback, test that this product has no syntax or logical errors. (2-3 hours)
2. **The Main Product:** functional but with no extra subsets of op code : Complete by mid-April :
   - Extend Report with updates (3-4 days)
   - The program will take the decompiled Op code and generate reliable C code AND will successfully be usable by CPROVER tests(3+weeks)
   - Get client feedback, test that this product has no syntax or logical errors. (2-3 hours)
3. **Additional Features:** Complete towards the start of May :
   - Look at the behaviour of the decompiled C program and add additional features based on the main product's current output(1+week(s))
   - Add additional op code features (extending the decompiler and code generator made in the first two iterations) (2+ weeks)

Refer to this iterative workflow diagram(each stage must go through testing or a client meeting before moving on to the next step, and preferably each iteration SHOULD take 3 weeks (4 max)):



Risks:

Project Management:

- The EVM byte code may be difficult to decompile into C, and may not produce perfect C code. This should not be an issue if my prior research is well done at the start of the project. This research will be continued throughout the project incase I discover more up to date examples.

- What the CPROVER identifies as a bug may not be a bug for the EVM code… is this an accurate assumption to assume C code bugs are the same as EVM bugs?
  - To refrain this from preventing the success of my project I will be working using an iterative approach, so if the first iterations fail I will have enough time to plan a new method.

Technical:

- What if the systems don't work , I.e platforms and breaking of tools?
  - I will be displaying mitigation and proof of my attempts for the solutions , I am also prepared to use virtual machines , (for windows dependencies as I am using a MacBook)
- Am I able to produce a program to satisfy the Academic Client?
  - Regular Client meetings every week will prevent this from happening

Personal:

- There may be Covid restrictions that prevent some meetings with the client. To prevent this I am prepared to communicate using online meetings if the situation calls for it.

- Will the detection of bugs possibly be incorrect?
  - As we are dealing with cryptocurrency the risks of being responsible for the authentication of smart contracts could cause losses to people relying on the tool if it is used and is incorrect.

- How will the tool be used? And what for?
  - This project could be an example of dual use technology, while I am aware of possible concerning uses of this type of tool , if I find bugs I will not be using my findings in a malicious manner.

  - If we find a bug this could be ethically wrong if it's used for personal gain. If the tool is used as a form of Responsible disclosure then its ethical. However if the tool is used unethically then knowledge of bugs in EVM programs, that are supposed to be safe, could put a lot of smart contracts in danger.

Risk assessment:

Likelihood (of this being a risk) : 1-5 (1 being very unlikely, 5 being very likely)
Severity(how necessary this is) : 1-5 (1 being not impactful, 5 being very impactful)
Score is the Likelihood*Severity, (0-6 is low risk, 6-14 is medium risk, 15+ is high risk)

Objective	Likelihood	Severity	Score	Risk	Prevention
The program will take some EVM code and decompile it , translate into some sort of reliable C code.	1	5	5	Problems with incorrectly decompiling due to lack of knowledge of Solidity and EVM	Look at existing tools or extend an existing decompiler such as https://github.com/MrLuit/evm and get comfortable with EVM and solidity via tutorials and existing smart contracts.
The program will take the decompiled Op code and generate reliable C code	3	5	15	Unable to convert the op code to reliable C	Testing throughout the implementation with the client will prevent this problem
The generated C code will successfully be usable by CPROVER tests	2	5	10	CPROVER tests cannot be applied to the generated code	Re-attempt the second phase and look at existing code translation generators
Look at the behaviour of the decompiled C program and add additional features based on the main product's current output	2	2	4	possibly the C code generated could be incorrect	Look at existing op code to C conversions to identify where the program has gone wrong
Add additional op code features (extending the decompiler and code generator made in the first two iterations)	5	3	15	The additional Op code features may be too complex to decompile and then generate C code from	Tackle easier commands first and prioritise the generation of valid C code over the quantity of op codes decompiles

References:

The Guardian. (2017). *"$300m in cryptocurrency" accidentally lost forever due to bug*. [online] Available at: https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether [Accessed 3 Feb. 2022].

Research Ethics Checklist

CSREC –Review Form – Part A : Ethics Checklist *Version 4.4, October 2015, April 2019*

Research Ethics Review Form: BSc, MSc and MA Projects

Computer Science Research Ethics Committee (CSREC)

http://www.city.ac.uk/department-computer-science/research-ethics

Undergraduate and postgraduate students undertaking their final project in the Department of Computer Science are required to consider the ethics of their project work and to ensure that it complies with research ethics guidelines. In some cases, a project will need approval from an ethics committee before it can proceed. Usually, but not always, this will be because the student is involving other people ("participants") in the project.

In order to ensure that appropriate consideration is given to ethical issues, all students must complete this form and attach it to their project proposal document. There are two parts:

PART A: Ethics Checklist. All students must complete this part. The checklist identifies whether the project requires ethical approval and, if so, where to apply for approval.

PART B: Ethics Proportionate Review Form. Students who have answered "no" to all questions in A1, A2 and A3 and "yes" to question 4 in A4 in the ethics checklist must complete this part. The project supervisor has delegated authority to provide approval in such cases that are considered to involve MINIMAL risk. The approval may be ***provisional*** – *identifying the planned research as* likely to involve MINIMAL RISK. In such cases you must additionally seek ***full approval*** from the supervisor as the project progresses and details are established. ***Full approval*** must be acquired in writing, before beginning the planned research.

A.1 If you answer YES to any of the questions in this block, you must apply to an appropriate external ethics committee for approval and log this approval as an External Application through Research Ethics Online - https://ethics.city.ac.uk/		*Delete as appropriate*
1.1	Does your research require approval from the National Research Ethics Service (NRES)? *e.g. because you are recruiting current NHS patients or staff?* *If you are unsure try - https://www.hra.nhs.uk/approvals-amendments/what-approvals-do-i-need/*	**NO**
1.2	Will you recruit participants who fall under the auspices of the Mental Capacity Act? *Such research needs to be approved by an external ethics committee such as NRES or the Social Care Research Ethics Committee - http://www.scie.org.uk/research/ethics-committee/*	**NO**
1.3	Will you recruit any participants who are currently under the auspices of the Criminal Justice System, for example, but not limited to, people on remand, prisoners and those on probation? *Such research needs to be authorised by the ethics approval system of the National Offender Management Service.*	**NO**
A.2 If you answer YES to any of the questions in this block, then unless you are applying to an external ethics committee, you must apply for approval from the Senate Research Ethics Committee (SREC) through Research Ethics Online - https://ethics.city.ac.uk/		*Delete as appropriate*
2.1	Does your research involve participants who are unable to give informed consent? *For example, but not limited to, people who may have a degree of learning disability or mental health problem, that means they are unable to make an informed decision on their own behalf.*	**NO**
2.2	Is there a risk that your research might lead to disclosures from participants concerning their involvement in illegal activities?	**NO**
2.3	Is there a risk that obscene and or illegal material may need to be accessed for your research study (including online content and other material)?	**NO**

2.4	Does your project involve participants disclosing information about special category or sensitive subjects? *For example, but not limited to: racial or ethnic origin; political opinions; religious beliefs; trade union membership; physical or mental health; sexual life; criminal offences and proceedings*	**NO**
2.5	Does your research involve you travelling to another country outside of the UK, where the Foreign & Commonwealth Office has issued a travel warning that affects the area in which you will study? *Please check the latest guidance from the FCO - http://www.fco.gov.uk/en/*	**NO**
2.6	Does your research involve invasive or intrusive procedures? *These may include, but are not limited to, electrical stimulation, heat, cold or bruising.*	**NO**
2.7	Does your research involve animals?	**NO**
2.8	Does your research involve the administration of drugs, placebos or other substances to study participants?	**NO**
A.3 If you answer YES to any of the questions in this block, then unless you are applying to an external ethics committee or the SREC, you must apply for approval from the Computer Science Research Ethics Committee (CSREC) through Research Ethics Online - https://ethics.city.ac.uk/ **Depending on the level of risk associated with your application, it may be referred to the Senate Research Ethics Committee.**		*Delete as appropriate*
3.1	Does your research involve participants who are under the age of 18?	**NO**
3.2	Does your research involve adults who are vulnerable because of their social, psychological or medical circumstances (vulnerable adults)? *This includes adults with cognitive and / or learning disabilities, adults with physical disabilities and older people.*	**NO**
3.3	Are participants recruited because they are staff or students of City, University of London? *For example, students studying on a particular course or module.* *If yes, then approval is also required from the Head of Department or Programme Director.*	**NO**
3.4	Does your research involve intentional deception of participants?	**NO**
3.5	Does your research involve participants taking part without their informed consent?	**NO**
3.5	Is the risk posed to participants greater than that in normal working life?	**NO**
3.7	Is the risk posed to you, the researcher(s), greater than that in normal working life?	**NO**
A.4 If you answer YES to the following question and your answers to all other questions in sections A1, A2 and A3 are NO, then your project is deemed to be of MINIMAL RISK. **If this is the case, then you can apply for approval through your supervisor under PROPORTIONATE REVIEW. You do so by completing PART B of this form.** **If you have answered NO to all questions on this form, then your project does not require ethical approval. You should submit and retain this form as evidence of this.**		*Delete as appropriate*
4	Does your project involve human participants or their identifiable personal data? *For example, as interviewees, respondents to a survey or participants in testing.*	**NO**

Appendix B: Opcode table with detailed descriptions

Implemented on	Stack	Name	Gas	Initial Stack	Resulting Stack	Mem / Storage	Notes
05/03	00	STOP	0				halt execution
05/03	01	ADD	3	a, b	a + b		(u)int256 addition modulo 2**256
05/03	02	MUL	5	a, b	a * b		(u)int256 multiplication modulo 2**256
05/03	03	SUB	3	a, b	a - b		(u)int256 addition modulo 2**256
05/03	04	DIV	5	a, b	a // b		uint256 division
05/03	05	SDIV	5	a, b	a // b		int256 division
05/03	06	MOD	5	a, b	a % b		uint256 modulus
05/03	07	SMOD	5	a, b	a % b		int256 modulus
05/03	08	ADDMOD	8	a, b, N	(a + b) % N		(u)int256 addition modulo N
05/03	09	MULMOD	8	a, b, N	(a * b) % N		(u)int256 multiplication modulo N
	0A	EXP	A1	a, b	a ** b		uint256 exponentiation modulo 2**256
	0B	SIGNEXTEND	5	b, x	SIGNEXTEND(x, b)		sign extend x
	0C-0F	*invalid*					
05/03	10	LT	3	a, b	a < b		uint256 less-than
05/03	11	GT	3	a, b	a > b		uint256 greater-than
05/03	12	SLT	3	a, b	a < b		int256 less-than
05/03	13	SGT	3	a, b	a > b		int256 greater-than
05/03	14	EQ	3	a, b	a == b		(u)int256 equality
05/03	15	ISZERO	3	a	a == 0		(u)int256 iszero
05/03	16	AND	3	a, b	a && b		bitwise AND
05/03	17	OR	3	a, b	a || b		bitwise OR
05/03	18	XOR	3	a, b	a ^ b		bitwise XOR
05/03	19	NOT	3	a	~a		bitwise NOT
	1A	BYTE	3	i, x	(x >> (248 - i * 8)) && 0xFF		ith byte of (u)int256 x, from the left
	1B	SHL	3	shift, val	val << shift		shift left
	1C	SHR	3	shift, val	val >> shift		logical shift right
	1D	SAR	3	shift, val	val >> shift		arithmetic shift right
	1E-1F	*invalid*					
	20	SHA3	A2	ost, len	keccak256(mem[ost:ost+len])		keccak256
	21-2F	*invalid*					
05/03	30	ADDRESS	2	.	address(this)		address of executing contract
05/03	31	BALANCE	A5	addr	addr.balance		balance, in wei

05/03	32	ORIGIN	2	.	tx.origin		address that originated the tx
05/03	33	CALLER	2	.	msg.sender		address of msg sender
05/03	34	CALLVALUE	2	.	msg.value		msg value, in wei
05/03	35	CALLDATALOAD	3	idx	msg.data[idx:idx+32]		read word from msg data at index idx
05/03	36	CALLDATASIZE	2	.	len(msg.data)		length of msg data, in bytes
05/03	37	CALLDATACOPY	A3	dstOst, ost, len	.	mem[dstOst:dstOst+len] := msg.data[ost:ost+len	copy msg data
05/03	38	CODESIZE	2	.	len(this.code)		length of executing contract's code, in bytes
05/03	39	CODECOPY	A3	dstOst, ost, len	.		mem[dstOst:dstOst+len] := this.code[ost:ost+len] copy executing contract's bytecode
15/03	3A	GASPRICE	2	.	tx.gasprice		gas price of tx, in wei per unit gas **
15/03	3B	EXTCODESIZE	A5	addr	len(addr.code)		size of code at addr, in bytes
15/03	3C	EXTCODECOPY	A4	addr, dstOst, ost, len	.	mem[dstOst:dstOst+len] := addr.code[ost:ost+len]	copy code from addr
15/03	3D	RETURNDATASIZE	2	.	size		size of returned data from last external call, in bytes
15/03	3E	RETURNDATACOPY	A3	dstOst, ost, len	.	mem[dstOst:dstOst+len] := returndata[ost:ost+len]	copy returned data from last external call
15/03	3F	EXTCODEHASH	A5	addr	hash		hash = addr.exists ? keccak256(addr.code) : 0
05/03	40	BLOCKHASH	20	blockNum	blockHash(blockNum)		
05/03	41	COINBASE	2	.	block.coinbase		address of miner of current block

05/03	42	TIMESTAMP	2	.	block.timestamp		timestamp of current block
05/03	43	NUMBER	2	.	block.number		number of current block
05/03	44	DIFFICULTY	2	.	block.difficulty		difficulty of current block
05/03	45	GASLIMIT	2	.	block.gaslimit		gas limit of current block
05/03	46	CHAINID	2	.	chain_id		push current chain id onto stack
05/03	47	SELFBALANCE	5	.	address(this).balance		balance of executing contract, in wei
05/03	48	BASEFEE	2	.	block.basefee		base fee of current block
15/03	49-4F	*invalid*					
05/03	50	POP	2	_anon	.		remove item from top of stack and discard it
05/03	51	MLOAD	3*	ost	mem[ost:ost+32]		read word from memory at offset ost
05/03	52	MSTORE	3*	ost, val	.	mem[ost:ost+32] := val	write a word to memory
05/03	53	MSTORE8	3*	ost, val	.	mem[ost] := val && 0xFF	write a single byte to memory
05/03	54	SLOAD	A6	key	storage[key]		read word from storage
05/03	55	SSTORE	A7	key, val	.	storage[key] := val	write word to storage
05/03	56	JUMP	8	dst	.		$pc := dst mark that pc is only assigned if dst is a valid jumpdest
05/03	57	JUMPI	10	dst, condition	.		$pc := condition ? dst : $pc + 1
05/03	58	PC	2	.	$pc		program counter
05/03	59	MSIZE	2	.	len(mem)		size of memory in current execution context, in bytes
15/03	5A	GAS	2	.	gasRemaining		
20/03	5B	JUMPDEST	1			mark valid jump destination	a valid jump destination for example a jump destination not inside the push data
20/03	5C-5F	*invalid*					
20/03	60	PUSH1	3	.	uint8		push 1-byte value onto stack
20/03	61	PUSH2	3	.	uint16		push 2-byte value onto stack
20/03	62	PUSH3	3	.	uint24		push 3-byte value onto stack

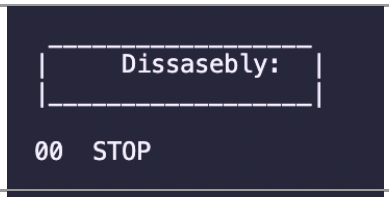
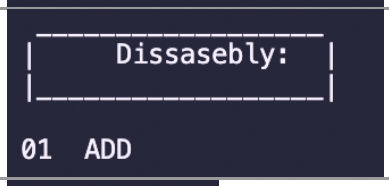
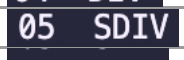
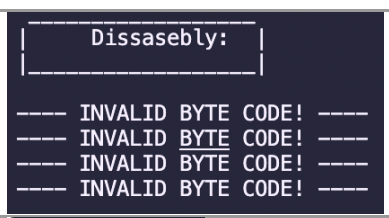
20/03	63	PUSH4	3	.	uint32		push 4-byte value onto stack
20/03	64	PUSH5	3	.	uint40		push 5-byte value onto stack
20/03	65	PUSH6	3	.	uint48		push 6-byte value onto stack
20/03	66	PUSH7	3	.	uint56		push 7-byte value onto stack
20/03	67	PUSH8	3	.	uint64		push 8-byte value onto stack
20/03	68	PUSH9	3	.	uint72		push 9-byte value onto stack
20/03	69	PUSH10	3	.	uint80		push 10-byte value onto stack
20/03	6A	PUSH11	3	.	uint88		push 11-byte value onto stack
20/03	6B	PUSH12	3	.	uint96		push 12-byte value onto stack
20/03	6C	PUSH13	3	.	uint104		push 13-byte value onto stack
20/03	6D	PUSH14	3	.	uint112		push 14-byte value onto stack
20/03	6E	PUSH15	3	.	uint120		push 15-byte value onto stack
20/03	6F	PUSH16	3	.	uint128		push 16-byte value onto stack
20/03	70	PUSH17	3	.	uint136		push 17-byte value onto stack
20/03	71	PUSH18	3	.	uint144		push 18-byte value onto stack
20/03	72	PUSH19	3	.	uint152		push 19-byte value onto stack
20/03	73	PUSH20	3	.	uint160		push 20-byte value onto stack
20/03	74	PUSH21	3	.	uint168		push 21-byte value onto stack
20/03	75	PUSH22	3	.	uint176		push 22-byte value onto stack
20/03	76	PUSH23	3	.	uint184		push 23-byte value onto stack
20/03	77	PUSH24	3	.	uint192		push 24-byte value onto stack
20/03	78	PUSH25	3	.	uint200		push 25-byte value onto stack
20/03	79	PUSH26	3	.	uint208		push 26-byte value onto stack
20/03	7A	PUSH27	3	.	uint216		push 27-byte value onto stack

20/03	7B	PUSH28	3	.	uint224		push 28-byte value onto stack
20/03	7C	PUSH29	3	.	uint232		push 29-byte value onto stack
20/03	7D	PUSH30	3	.	uint240		push 30-byte value onto stack
20/03	7E	PUSH31	3	.	uint248		push 31-byte value onto stack
20/03	7F	PUSH32	3	.	uint256		push 32-byte value onto stack
22/03	80	DUP1	3	a	a, a		clone 1st value on stack
22/03	81	DUP2	3	_, a	a, _, a		clone 2nd value on stack
22/03	82	DUP3	3	_, _, a	a, _, _, a		clone 3rd value on stack
22/03	83	DUP4	3	_, _, _, a	a, _, _, _, a		clone 4th value on stack
22/03	84	DUP5	3	..., a	a, ..., a		clone 5th value on stack
22/03	85	DUP6	3	..., a	a, ..., a		clone 6th value on stack
22/03	86	DUP7	3	..., a	a, ..., a		clone 7th value on stack
22/03	87	DUP8	3	..., a	a, ..., a		clone 8th value on stack
22/03	88	DUP9	3	..., a	a, ..., a		clone 9th value on stack
22/03	89	DUP10	3	..., a	a, ..., a		clone 10th value on stack
22/03	8A	DUP11	3	..., a	a, ..., a		clone 11th value on stack
22/03	8B	DUP12	3	..., a	a, ..., a		clone 12th value on stack
22/03	8C	DUP13	3	..., a	a, ..., a		clone 13th value on stack
22/03	8D	DUP14	3	..., a	a, ..., a		clone 14th value on stack
22/03	8E	DUP15	3	..., a	a, ..., a		clone 15th value on stack
22/03	8F	DUP16	3	..., a	a, ..., a		clone 16th value on stack
22/03	90	SWAP1	3	a, b	b, a		
22/03	91	SWAP2	3	a, _, b	b, _, a		
22/03	92	SWAP3	3	a, _, _, b	b, _, _, a		
22/03	93	SWAP4	3	a, _, _, _, b	b, _, _, _, a		
22/03	94	SWAP5	3	a, ..., b	b, ..., a		
22/03	95	SWAP6	3	a, ..., b	b, ..., a		
22/03	96	SWAP7	3	a, ..., b	b, ..., a		
22/03	97	SWAP8	3	a, ..., b	b, ..., a		
22/03	98	SWAP9	3	a, ..., b	b, ..., a		
22/03	99	SWAP10	3	a, ..., b	b, ..., a		
22/03	9A	SWAP11	3	a, ..., b	b, ..., a		
22/03	9B	SWAP12	3	a, ..., b	b, ..., a		
22/03	9C	SWAP13	3	a, ..., b	b, ..., a		
22/03	9D	SWAP14	3	a, ..., b	b, ..., a		
22/03	9E	SWAP15	3	a, ..., b	b, ..., a		
22/03	9F	SWAP16	3	a, ..., b	b, ..., a		
22/03	A0	LOG0	A8	ost, len	.		LOG0(memory[ost:ost+len])
22/03	A1	LOG1	A8	ost, len, topic0	.		LOG1(memory[ost:ost+len], topic0)

22/03	A2	LOG2	A8	ost, len, topic0, topic1	.		LOG1(memory[ost:ost+len], topic0, topic1)
22/03	A3	LOG3	A8	ost, len, topic0, topic1, topic2	.		LOG1(memory[ost:ost+len], topic0, topic1, topic2)
22/03	A4	LOG4	A8	ost, len, topic0, topic1, topic2, topic3	.		LOG1(memory[ost:ost+len], topic0, topic1, topic2, topic3)
22/03	A5-EF	*invalid*					
22/03	F0	CREATE	A9	val, ost, len	addr		addr = keccak256(rlp([address(this), this.nonce]))
22/03	F1	CALL	AA	gas, addr, val, argOst, argLen, retOst, retLen	success	mem[retOst:retOst+retLen] := returndata	
22/03	F2	CALLCODE	AA	gas, addr, val, argOst, argLen, retOst, retLen	success	mem[retOst:retOst+retLen] = returndata	same as DELEGATECALL but does not propagate original msg.sender and msg.value
22/03	F3	RETURN	0*	ost, len	.		return mem[ost:ost+len]
22/03	F4	DELEGATECALL	AA	gas, addr, argOst, argLen, retOst, retLen	success	mem[retOst:retOst+retLen] := returndata	
22/03	F5	CREATE2	A9	val, ost, len, salt	addr		addr = keccak256(0xff ++ address(this) ++ salt ++ keccak256(mem[ost:ost+len]))[12:]
22/03	F6-F9	*invalid*					
22/03	FA	STATICCALL	AA	gas, addr, argOst, argLen, retOst, retLen	success	mem[retOst:retOst+retLen] := returndata	
22/03	FB-FC	*invalid*					

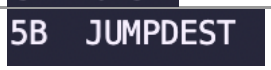
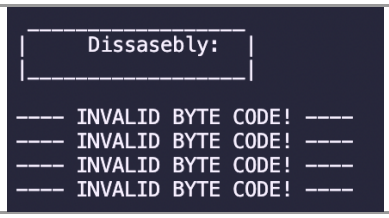
22/03	FD	REVERT	0*	ost, len	.		revert(mem[ost:ost+len])
22/03	FE	INVALID	AF			designated invalid opcode - EIP-141	
22/03	FF	SELFDESTRUCT	AB	addr			destroy contract and sends all funds to addr
22/03							
22/03							

Appendix B: Test table – Disassembly

Stack	Name	Inputs	Expected outputs	Actual output (Screenshots)	Passed/ Failed
00	STOP	00	00 STOP		Passed
01	ADD	01	01 ADD		Passed
02	MUL	02	02 MUL		Passed
03	SUB	03	03 SUB		Passed
04	DIV	04	04 DIV		Passed
05	SDIV	05	05 SDIV		Passed
06	MOD	06	06 MOD		Passed
07	SMOD	07	07 SMOD		Passed
08	ADDMOD	08	08 ADDMOD		Passed
09	MULMOD	09	09 MULMOD		Passed
0A	EXP	0A	0A EXP		Passed
0B	SIGNEXTEND	0B	0B SIGNEXTEND		Passed
0C-0F	*invalid*	0C0D0E0F	*invalid*		Passed
10	LT	10	10 LT		Passed

11	GT	11	11 GT	11 GT	Passed
12	SLT	12	12 SLT	12 SLT	Passed
13	SGT	13	13 SGT	13 SGT	Passed
14	EQ	14	14 EQ	14 EQ	Passed
15	ISZERO	15	15 ISZERO	15 ISZERO	Passed
16	AND	16	16 AND	16 AND	Passed
17	OR	17	17 OR	17 OR	Passed
18	XOR	18	18 XOR	18 XOR	Passed
19	NOT	19	19 NOT	19 NOT	Passed
1A	BYTE	1A	1A BYTE	1A BYTE	Passed
1B	SHL	1B	1B SHL	1B SHL	Passed
1C	SHR	1C	1C SHR	1C SHR	Passed
1D	SAR	1D	1D SAR	1D SAR	Passed
1E-1F	*invalid*	1E1F	*invalid*	Dissasebly: ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ----	Passed
20	SHA3	20	20 SHA3	Dissasebly: 20 SHA3	Passed
21-2F	*invalid*	212223242526 2728292A2B2 C2D2E2F	*invalid*	---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ----	Passed
30	ADDRESS	30	30 ADDRESS	30 ADDRESS	Passed
31	BALANCE	31	31 BALANCE	31 BALANCE	Passed
32	ORIGIN	32	32 ORIGIN	32 ORIGIN	Passed
33	CALLER	33	33 CALLER	33 CALLER	Passed
34	CALLVALUE	34	34 CALLVALUE	34 CALLVALUE	Passed
35	CALLDATALOAD	35	35 CALLDATALOAD	35 CALLDATALOAD	Passed
36	CALLDATASIZE	36	36 CALLDATASIZE	36 CALLDATASIZE	Passed

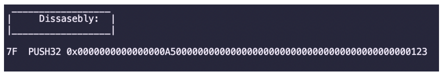
37	CALLDATACOPY	37	37 CALLDATACOPY	37 CALLDATACOPY	Passed
38	CODESIZE	38	38 CODESIZE	38 CODESIZE	Passed
39	CODECOPY	39	39 CODECOPY	39 CODECOPY	Passed
3A	GASPRICE	3A	3A GASPRICE	3A GASPRICE	Passed
3B	EXTCODESIZE	3B	3B EXTCODESIZE	3B EXTCODESIZE	Passed
3C	EXTCODECOPY	3C	3C EXTCODECOPY	3C EXTCODECOPY	Passed
3D	RETURNDATASIZE	3D	3D RETURNDATASIZE	3D RETURNDATASIZE	Passed
3E	RETURNDATACOPY	3E	3E RETURNDATACOPY	3E RETURNDATACCOPY	Passed
3F	EXTCODEHASH	3F	3F EXTCODEHASH	3F EXTCODEHASH	Passed
40	BLOCKHASH	40	40 BLOCKHASH	40 BLOCKHASH	Passed
41	COINBASE	41	41 COINBASE	41 COINBASE	Passed
42	TIMESTAMP	42	42 TIMESTAMP	42 TIMESTAMP	Passed
43	NUMBER	43	43 NUMBER	43 NUMBER	Passed
44	DIFFICULTY	44	44 DIFFICULTY	44 DIFFICULTY	Passed
45	GASLIMIT	45	45 GASLIMIT	45 GASLIMIT	Passed
46	CHAINID	46	46 CHAINID	46 CHAINID	Passed
47	SELFBALANCE	47	47 SELFBALANCE	47 SELFBALANCE	Passed
48	BASEFEE	48	48 BASEFEE	48 BASEFEE	Passed

49-4F	*invalid*	494A4B4C4D4E4F	*invalid*		Passed
50	POP	50	50 POP		Passed
51	MLOAD	51	51 MLOAD		Passed
52	MSTORE	52	52 MSTORE		Passed
53	MSTORE8	53	53 MSTORE8		Passed
54	SLOAD	54	54 SLOAD		Passed
55	SSTORE	55	55 SSTORE		Passed
56	JUMP	56	56 JUMP		Passed
57	JUMPI	57	57 JUMPI		Passed
58	PC	58	58 PC		Passed
59	MSIZE	59	59 MSIZE		Passed
5A	GAS	5A	5A GAS		Passed
5B	JUMPDEST	5B	5B JUMPDEST		Passed
5C-5F	*invalid*	5C5D5E5F	*invalid*		Passed
60	PUSH1	*6000*	60 PUSH1 0x00	60 PUSH1 0x00	Passed
61	PUSH2	*610000*	61 PUSH2 0x0000	61 PUSH2 0x0000	Passed
62	PUSH3	*62000000*	62 PUSH3 0x000000	62 PUSH3 0x000000	Passed
63	PUSH4	*6300000000*	63 PUSH4 0x00000000	63 PUSH4 0x00000000	Passed
64	PUSH5	*640000000000*	64 PUSH5 0x0000000000	64 PUSH5 0x0000000000	Passed
65	PUSH6	*65000000000000*	65 PUSH6 0x000000000000	65 PUSH6 0x000000000000	Passed
66	PUSH7	*6600000000000000*	66 PUSH7 0x00000000000000	66 PUSH7 0x00000000000000	Passed

67	PUSH8	*670000000000000000*	67 PUSH8 0x0000000000000000	67 PUSH8 0x0000000000000000	Passed
68	PUSH9	*680000000000000000*	68 PUSH9 0x0000000000000000	68 PUSH9 0x000000000000000000	Passed
69	PUSH10	*690000000000000000000*	69 PUSH10 0x00000000000000000000	69 PUSH10 0x00000000000000000000	Passed
6A	PUSH11	*6A0000000000000000000000*	6A PUSH11 0x0000000000000000000000	6A PUSH11 0x0000000000000000000000	Passed
6B	PUSH12	*6B000000000000000000000000*	6B PUSH12 0x000000000000000000000000	6B PUSH12 0x000000000000000000000000	Passed
6C	PUSH13	*6C00000000000000000000000000*	6C PUSH13 0x00000000000000000000000000	6C PUSH13 0x000000000000000000000000000	Passed
6D	PUSH14	*6D0000000000000000000000000000*	6D PUSH14 0x00000000000000000000000000	6D PUSH14 0x0000000000000000000000000000	Passed
6E	PUSH15	*6E000000000000000000000000000000*	6E PUSH15 0x000000000000000000000000000000	6E PUSH15 0x00000000000000000000000000000	Passed
6F	PUSH16	*6F00000000000000000000000000000000*	6F PUSH16 0x0000000000000000000000000000	6F PUSH16 0x00000000000000000000000000000000	Passed
70	PUSH17	*700000000000000000000000000000000000*	70 PUSH17 0x00000000000000000000000000000000	70 PUSH17 0x000000000000000000000000000000000	Passed

71	PUSH18	*710000000000000000000000000000000000*	71 PUSH18 0x000000000000000000000000000000000000	71 PUSH18 0x000000000000000000000000000000000000	Passed
72	PUSH19	*720000000000000000000000000000000000*	72 PUSH19 0x000000000000000000000000000000000000	72 PUSH19 0x000000000000000000000000000000000000	Passed
73	PUSH20	*730000000000000000000000000000000000*	73 PUSH20 0x000000000000000000000000000000000000	73 PUSH20 0x000000000000000000000000000000000000	Passed
74	PUSH21	*740000000000000000000000000000000000*	74 PUSH21 0x000000000000000000000000000000000000	74 PUSH21 0x000000000000000000000000000000000000	Passed
75	PUSH22	*750000000000000000000000000000000000*	75 PUSH22 0x000000000000000000000000000000000000	75 PUSH22 0x000000000000000000000000000000000000	Passed
76	PUSH23	*760000000000000000000000000000000000*	76 PUSH23 0x000000000000000000000000000000000000	76 PUSH23 0x000000000000000000000000000000000000	Passed
77	PUSH24	*770000000000000000000000000000000000*	77 PUSH24 0x000000000000000000000000000000000000	77 PUSH24 0x000000000000000000000000000000000000	Passed
78	PUSH25	*780000000000000000000000000000000000*	78 PUSH25 0x000000000000000000000000000000000000	78 PUSH25 0x000000000000000000000000000000000000	Passed

		0000000000000000	0000000000000000000000000000000000000000000000000000000000000000		
79	PUSH26	*7900000000000000000000000000000000000000000000000000000000000000*	79 PUSH26 0x0000000000000000000000000000000000000000000000000000000000000000	79 PUSH26 0x0000000000000000000000000000000000000000000000000000000000000000	Passed
7A	PUSH27	*7A0000000000000000000000000000000000000000000000000000000000000000*	7A PUSH27 0x0000000000000000000000000000000000000000000000000000000000000000	7A PUSH27 0x0000000000000000000000000000000000000000000000000000000000000000	Passed
7B	PUSH28	*7B0000000000000000000000000000000000000000000000000000000000000000*	7B PUSH28 0x0000000000000000000000000000000000000000000000000000000000000000	7B PUSH28 0x0000000000000000000000000000000000000000000000000000000000000000	Passed
7C	PUSH29	*7C0000000000000000000000000000000000000000000000000000000000000000*	7C PUSH29 0x0000000000000000000000000000000000000000000000000000000000000000	7C PUSH29 0x0000000000000000000000000000000000000000000000000000000000000000	Passed
7D	PUSH30	*7D0000000000000000000000000000000000000000000000000000000000000000*	7D PUSH30 0x0000000000000000000000000000000000000000000000000000000000000000	7D PUSH30 0x0000000000000000000000000000000000000000000000000000000000000000	Passed
7E	PUSH31	*7E0000000000000000000000000000000000000000000000000000000000000000*	7E PUSH31 0x0000000000000000000000000000000000000000000000000000000000000000	7E PUSH31 0x0000000000000000000000000000000000000000000000000000000000000000	Passed

		0000000000000000	0000000000000000000000000000000000000000000		
7F	PUSH32	*7F0000000000000000a50000000000000000000000000000000000000000000000123*	7F PUSH32 0x0000000000000000A50000000000000000000000000000000000000000000000123	7F PUSH32 0x0000000000000000A50000000000000000000000000000000000000000000000123 See:  	Passed
80	DUP1	80	80 DUP1	80 DUP1	Passed
81	DUP2	81	81 DUP2	81 DUP2	Passed
82	DUP3	82	82 DUP3	82 DUP3	Passed
83	DUP4	83	83 DUP4	83 DUP4	Passed
84	DUP5	84	84 DUP5	84 DUP5	Passed
85	DUP6	85	85 DUP6	85 DUP6	Passed
86	DUP7	86	86 DUP7	86 DUP7	Passed
87	DUP8	87	87 DUP8	87 DUP8	Passed
88	DUP9	88	88 DUP9	88 DUP9	Passed
89	DUP10	89	89 DUP10	89 DUP10	Passed
8A	DUP11	8A	8A DUP11	8A DUP11	Passed
8B	DUP12	8B	8B DUP12	8B DUP12	Passed
8C	DUP13	8C	8C DUP13	8C DUP13	Passed
8D	DUP14	8D	8D DUP14	8D DUP14	Passed
8E	DUP15	8E	8E DUP15	8E DUP15	Passed
8F	DUP16	8F	8F DUP16	8F DUP16	Passed
90	SWAP1	90	90 SWAP1	90 SWAP1	Passed
91	SWAP2	91	91 SWAP2	91 SWAP2	Passed
92	SWAP3	92	92 SWAP3	92 SWAP3	Passed
93	SWAP4	93	93 SWAP4	93 SWAP4	Passed
94	SWAP5	94	94 SWAP5	94 SWAP5	Passed
95	SWAP6	95	95 SWAP6	95 SWAP6	Passed
96	SWAP7	96	96 SWAP7	96 SWAP7	Passed
97	SWAP8	97	97 SWAP8	97 SWAP8	Passed
98	SWAP9	98	98 SWAP9	98 SWAP9	Passed
99	SWAP10	99	99 SWAP10	99 SWAP10	Passed
9A	SWAP11	9A	9A SWAP11	9A SWAP11	Passed
9B	SWAP12	9B	9B SWAP12	9B SWAP12	Passed
9C	SWAP13	9C	9C SWAP13	9C SWAP13	Passed

<table>
<tr><td>9D</td><td>SWAP14</td><td>9D</td><td>9D
SWAP14</td><td>9D SWAP14</td><td>Passed</td></tr>
<tr><td>9E</td><td>SWAP15</td><td>9E</td><td>9E
SWAP15</td><td>9E SWAP15</td><td>Passed</td></tr>
<tr><td>9F</td><td>SWAP16</td><td>9F</td><td>9F
SWAP16</td><td>9F SWAP16</td><td>Passed</td></tr>
<tr><td>A0</td><td>LOG0</td><td>A0</td><td>A0 LOG0</td><td>A0 LOG0</td><td>Passed</td></tr>
<tr><td>A1</td><td>LOG1</td><td>A1</td><td>A1 LOG1</td><td>A1 LOG1</td><td>Passed</td></tr>
<tr><td>A2</td><td>LOG2</td><td>A2</td><td>A2 LOG2</td><td>A2 LOG2</td><td>Passed</td></tr>
<tr><td>A3</td><td>LOG3</td><td>A3</td><td>A3 LOG3</td><td>A3 LOG3</td><td>Passed</td></tr>
<tr><td>A4</td><td>LOG4</td><td>A4</td><td>A4 LOG4</td><td>A4 LOG4</td><td>Passed</td></tr>
<tr><td>A5-EF</td><td>invalid</td><td>A5A6A7A8A9
AAABACAD
AEAFB1B2B3
B4B5B6B7B8
B9BABBBCB
DBEBFC1C2
C3C4C5C6C7
C8C9CACBC
CCDCECFD1
D2D3D4D5D6
D7D8D9DAD
BDCDDDEDF
E1E2E3E4E5E
6E7E8E9EAE
BECEDEEEF</td><td>invalid</td><td>____________________
| Disassembly: |
|____________________|
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----
---- INVALID BYTE CODE! ----</td><td>Passed</td></tr>
</table>

				---- INVALID BYTE CODE! ---- ---- INVALID BYTE CODE! ----	
F0	CREATE	F0	F0 CREATE	F0 CREATE	Passed
F1	CALL	F1	F1 CALL	F1 CALL	Passed
F2	CALLCODE	F2	F2 CALLCODE	F2 CALLCODE	Passed
F3	RETURN	F3	F3 RETURN	F3 RETURN	Passed
F4	DELEGATECALL	F4	F4 DELEGATECALL	F4 DELEGATECALL	Passed
F5	CREATE2	F5	F5 CREATE2	F5 CREATE2	Passed

F6-F9	*invalid*	F6F7F8F9	*invalid*		Passed
FA	STATICCALL	FA	FA STATICCALL	FA STATICCALL	Passed
FB-FC	*invalid*	FBFC	*invalid*		Passed
FD	REVERT	FD	FD REVERT	FD REVERT	Passed
FE	INVALID	FE	FE INVALID	FE INVALID	Passed
FF	SELFDESTRUCT	FF	FF SELFDESTRUCT	FF SELFDESTRUCT	Passed

Appendix B: Test table – Decompilation

Gas increments are based on the Ethereum yellow paper, each op code has a gas cost.
C files are structured like so, with labels in the test table generated after the comment "/*** Start of decompiled code ***/ ":

```c
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <stdint.h>

typedef int64_t var;

typedef uint64_t uvar;

#define GASLIMIT 10000
#define STACKHEIGHT 1024
#define MEMORYSIZE 10000

struct pair {
    var location;
    var value;
};
struct numpair {
    int address;
    double balance;
};
```

```
struct pair memory[MEMORYSIZE] = {};
struct pair storage[MEMORYSIZE] = {};
struct numpair accounts[MEMORYSIZE] = {};

var stack[STACKHEIGHT];
var top;

var memPoint;
var storePoint;
var accountNo;
var pcCounter;
var gasUsed;
var gasLimit = GASLIMIT;

int main (int argc, char **argv) {

/*** Start of decompiled code ***/

/*** End of decompiled code ***/
return 0;

}
```

Name	Inputs	Expected outputs	Actual output (C code generated)	Passed/ Failed
STOP	00	exit(0);	`exit(0);`	Passed

ADD	01	uvar tmp1 = stack[top - 2];
		uvar tmp2 = stack[top - 1];
		uvar tmp3 = tmp1 + tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			`label_0 :`
			`{`
			`uvar tmp1 = stack[top - 2];`
			`uvar tmp2 = stack[top - 1];`
			`uvar tmp3 = tmp1 + tmp2;`
			`top-=2;`
			`stack[top] = tmp3;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

MUL	02	uvar tmp1 = stack[top - 2];
		uvar tmp2 = stack[top - 1];
		uvar tmp3 = tmp1 * tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			`label_0 :`
			`{`
			`uvar tmp1 = stack[top - 2];`
			`uvar tmp2 = stack[top - 1];`
			`uvar tmp3 = tmp1 * tmp2;`
			`top-=2;`
			`stack[top] = tmp3;`
			`++top;`
				Passed

			assert(top != STACKHEIGHT);
			gasUsed +=5;
			assert(gasUsed < gasLimit);
			}	

SUB	03	uvar tmp1 = stack[top - 2];
		uvar tmp2 = stack[top - 1];
		uvar tmp3 = tmp1 - tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_0 :
			{
			uvar tmp1 = stack[top - 2];
			uvar tmp2 = stack[top - 1];
			uvar tmp3 = tmp1 - tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

DIV	04	uvar tmp1 = stack[top - 2];
		uvar tmp2 = stack[top - 1];
		uvar tmp3 = tmp1 /tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_0 :
			{
			uvar tmp1 = stack[top - 2];
			uvar tmp2 = stack[top - 1];
			uvar tmp3 = tmp1 /tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SDIV	05	var tmp1 = stack[top - 2];
		var tmp2 = stack[top - 1];
		var tmp3 = tmp1 /tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_0 :
			{
			var tmp1 = stack[top - 2];
			var tmp2 = stack[top - 1];
			var tmp3 = tmp1 /tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=5;
			assert(gasUsed < gasLimit);
			}
				Passed

MOD	06	uvar tmp1 = stack[top - 2];
		uvar tmp2 = stack[top - 1];
		uvar tmp3 = tmp1 %tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_0 :
			{
			uvar tmp1 = stack[top - 2];
			uvar tmp2 = stack[top - 1];
			uvar tmp3 = tmp1 % tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
				Passed

			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`	

SMOD	07	var tmp1 = stack[top - 2];
		var tmp2 = stack[top - 1];
		var tmp3 = tmp1 %tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			`label_0 :`
			`{`
			`var tmp1 = stack[top - 2];`
			`var tmp2 = stack[top - 1];`
			`var tmp3 = tmp1 %tmp2;`
			`top-=2;`
			`stack[top] = tmp3;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=5;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

ADDMOD	08	No value added to stack in EVM	`//ADDMOD`	-
MULMOD	09	No value added to stack in EVM	`//MULMOD`	-

EXP	0A	uvar tmp1 = stack[top - 2];
		uvar tmp2 = stack[top - 1];
		uvar tmp3 = pow(tmp1,tmp2);
		top-=2;
		stack[top] = tmp3;
		++top;
			`label_0 :`
			`{`
			`uvar tmp1 = stack[top - 2];`
			`uvar tmp2 = stack[top - 1];`
			`uvar tmp3 = pow(tmp1,tmp2);`
			`top-=2;`
			`stack[top] = tmp3;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=10;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

SIGNEXTEND	0B	var tmp1 = stack[top - 2];
		var tmp2 = stack[top - 1];
		var tmp3 = tmp1~ tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			`label_0 :`
			`{`
			`uvar tmp1 = stack[top - 2];`
			`uvar tmp2 = stack[top - 1];`
			`uvar tmp3 = tmp1~ tmp2;`
			`top-=2;`
			`stack[top] = tmp3;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=10;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

invalid	0C0D 0E0F	-	-	Passed

LT	10	uvar tmp1 = stack[top - 2];
		uvar tmp2 = stack[top - 1];
		uvar tmp3 = tmp1 < tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_1 :
			{
			uvar tmp1 = stack[top - 2];
			uvar tmp2 = stack[top - 1];
			uvar tmp3 = tmp1 < tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

GT	11	uvar tmp1 = stack[top - 2];
		uvar tmp2 = stack[top - 1];
		uvar tmp3 = tmp1 > tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_1 :
			{
			uvar tmp1 = stack[top - 2];
			uvar tmp2 = stack[top - 1];
			uvar tmp3 = tmp1 > tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SLT	12	var tmp1 = stack[top - 2];
		var tmp2 = stack[top - 1];
		var tmp3 = tmp1 < tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_1 :
			{
			var tmp1 = stack[top - 2];
			var tmp2 = stack[top - 1];
			var tmp3 = tmp1 < tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SGT	13	var tmp1 = stack[top - 2];
		var tmp2 = stack[top - 1];
		var tmp3 = tmp1 < tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_1 :
			{
			var tmp1 = stack[top - 2];
			var tmp2 = stack[top - 1];
			var tmp3 = tmp1 > tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
				Passed

			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`	

EQ	14	uvar tmp1 = stack[top - 2];
		uvar tmp2 = stack[top - 1];
		uvar tmp3 = tmp1 == tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			`label_1 :`
			`{`
			`uvar tmp1 = stack[top - 2];`
			`uvar tmp2 = stack[top - 1];`
			`uvar tmp3 = tmp1 == tmp2;`
			`top-=2;`
			`stack[top] = tmp3;`
			`++top;`
			
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

ISZERO	15	uvar tmp1 = stack[top - 1];
		uvar tmp2 = !(tmp1);
		--top;
		stack[top] = tmp2;
		++top;
			`label_17 :`
			`{`
			`uvar tmp1 = stack[top - 1];`
			`uvar tmp2 = !(tmp1);`
			`--top;`
			`stack[top] = tmp2;`
			`++top;`
			
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

AND	16	var tmp1 = stack[top - 2];
		var tmp2 = stack[top - 1];
		var tmp3 = tmp1 & tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			`label_1:`
			`{`
			`var tmp1 = stack[top - 2];`
			`var tmp2 = stack[top - 1];`
			`var tmp3 = tmp1 & tmp2;`
			`top-=2;`
			`stack[top] = tmp3;`
			`++top;`
			
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

OR	17	var tmp1 = stack[top - 2];
		var tmp2 = stack[top - 1];
		var tmp3 = tmp1 | tmp2;
		top-=2;
		stack[top] = tmp3;
			`label_1:`
			`{`
			`var tmp1 = stack[top - 2];`
			`var tmp2 = stack[top - 1];`
			`var tmp3 = tmp1 | tmp2;`
				Passed

		++top;	top-=2;
			stack[top] = tmp3;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}	

XOR	18	var tmp1 = stack[top - 2];
		var tmp2 = stack[top - 1];
		var tmp3 = tmp1 ^ tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_1:
			{
			var tmp1 = stack[top - 2];
			var tmp2 = stack[top - 1];
			var tmp3 = tmp1 ^ tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

NOT	19	uvar tmp1 = stack[top - 1];
		uvar tmp2 = ~(tmp1);
		--top;
		stack[top] = tmp2;
		++top;
			label_21 :
			{
			uvar tmp1 = stack[top - 1];
			uvar tmp2 = ~(tmp1);
			--top;
			stack[top] = tmp2;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
				Passed

BYTE	1A	gasUsed +=3;	gasUsed +=3;	

SHL	1B	var tmp1 = stack[top - 2];
		var tmp2 = stack[top - 1];
		var tmp3 = tmp1 << tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_1:
			{
			var tmp1 = stack[top - 2];
			var tmp2 = stack[top - 1];
			var tmp3 = tmp1 << tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SHR	1C	var tmp1 = stack[top - 2];
		var tmp2 = stack[top - 1];
		var tmp3 = tmp1 >> tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_1:
			{
			var tmp1 = stack[top - 2];
			var tmp2 = stack[top - 1];
			var tmp3 = tmp1 >> tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SAR	1D	var tmp1 = stack[top - 2];
		var tmp2 = stack[top - 1];
		var tmp3 = tmp1 >> tmp2;
		top-=2;
		stack[top] = tmp3;
		++top;
			label_1:
			{
			var tmp1 = stack[top - 2];
			var tmp2 = stack[top - 1];
			var tmp3 = tmp1 >> tmp2;
			top-=2;
			stack[top] = tmp3;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

invalid	1E1F	-	-	Passed

SHA3	60006
	00120
		IF AT LEAST 2 VALUES ARE IN THE STACK (I.E PUSHED):
		stack[top]=SHA3(TOP-1+TOP-2);
			/*** Start of decompiled code ***/
			
			label_0 :
			{
			stack[top] = 0x00;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
			
			label_1 :
			{
			stack[top] = 0x01;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
			
			label_2 :
			{
			gasUsed +=30;
			assert(gasUsed < gasLimit);
			stack[top]='+aR3AJ4WcekqzR2iX9p7Gt4nrgZdD5MWKyiRMLPbt0s=';
			
			}
			/*** End of decompiled code ***/	

invalid	21222 32425 26272 8292 A2B2 C2D2 E2F	-	-	Passed

ADDRESS	30	stack[top] = accounts[accountNo-1].address;
		++top;
			label_1:
			{
			stack[top] = accounts[accountNo-1].address;
			++top;
			
			gasUsed +=2;
			assert(gasUsed < gasLimit);
			}
				Passed

BALANCE	31	stack[top] = accounts[accountNo-1].balance;
		++top;
			label_1:
			{
			stack[top] = accounts[accountNo-1].balance;
			++top;
			
			gasUsed +=400;
			assert(gasUsed < gasLimit);
			}
				Passed

ORIGIN	32	-	gasUsed +=2; assert(gasUsed < gasLimit);	out of scope
CALLER	33	-	gasUsed +=2; assert(gasUsed < gasLimit);	out of scope
CALLVALUE	34	-	gasUsed +=2; assert(gasUsed < gasLimit);	out of scope
CALLDATALOAD	35	-	gasUsed +=3; assert(gasUsed < gasLimit);	out of scope
CALLDATASIZE	36	-	gasUsed +=2; assert(gasUsed < gasLimit);	out of scope
CALLDATACOPY	37	-	gasUsed +=3; assert(gasUsed < gasLimit);	out of scope
CODESIZE	38	-	gasUsed +=2; assert(gasUsed < gasLimit);	out of scope
CODECOPY	39	-	gasUsed +=3; assert(gasUsed < gasLimit);	out of scope

GASPRICE	3A	Return gas used, add to top of -stack	label_37 :
			{
			stack[top] = gasUsed;
			++top;
				Passed

			`gasUsed +=2;`
			`assert(gasUsed < gasLimit);`
			`}`	

EXTCODESIZE	3B	-	`gasUsed +=700;` `assert(gasUsed < gasLimit);`	out of scope
EXTCODECOPY	3C	-	`gasUsed +=700;` `assert(gasUsed < gasLimit);`	out of scope
RETURNDATASIZE	3D	-	`gasUsed +=2;` `assert(gasUsed < gasLimit);`	out of scope
RETURNDATACOPY	3E	-	`gasUsed +=3;` `assert(gasUsed < gasLimit);`	out of scope
EXTCODEHASH	3F	-	`gasUsed +=700;` `assert(gasUsed < gasLimit);`	out of scope
BLOCKHASH	40	-	`gasUsed +=20;` `assert(gasUsed < gasLimit);`	out of scope
COINBASE	41	-	`gasUsed +=2;` `assert(gasUsed < gasLimit);`	out of scope
TIMESTAMP	42	-	`gasUsed +=2;` `assert(gasUsed < gasLimit);`	out of scope
NUMBER	43	-	`gasUsed +=2;` `assert(gasUsed < gasLimit);`	out of scope
DIFFICULTY	44	-	`gasUsed +=2;` `assert(gasUsed < gasLimit);`	out of scope
GASLIMIT	45	-	`gasUsed +=2;` `assert(gasUsed < gasLimit);`	out of scope
CHAINID	46	-	`gasUsed +=2;` `assert(gasUsed < gasLimit);`	out of scope
SELFBALANCE	47	-	-	out of scope
BASEFEE	48	-	`gasUsed +=2;` `assert(gasUsed < gasLimit);`	out of scope
invalid	494A4B4C4D4E4F	-	-	Passed

POP	50	--top;
		assert(top >=0);
			label_1 :
			{
			--top;
			assert(top >=0);
			gasUsed +=2;
			assert(gasUsed < gasLimit);
			}
				Passed

MLOAD	51	Loop through all memory and find the value that maps to the location of the top of the stack	Label_1:
			{
			int i;
			for (i = 0; i <= memPoint; i++)
			{
			if (memory[i].location == stack[top-1])
			{
			stack[top]= memory[i].value;
			top++;
			}
			}
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

MSTORE	52	Create a new memory with the location set to the top of the stack, and value set to the second value on the stack	label_1:
			{
			memory[memPoint] = (struct pair){stack[top-1],stack[top-2] };
			top-=2;
			memPoint++;
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

MSTORE 8	53	Create a new memory with the location set to the top of the stack, and value set to the second value on the stack	label_1:
			{
			memory[memPoint] = (struct pair){stack[top-1],stack[top-2] };
			top-=2;
			memPoint++;
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SLOAD	54	Loop through all storage and find the value that maps to the location of the top of the stack	label_1:
			{
			int i;
			for (i = 0; i <= storePoint; i++)
			{
			if (storage[i].location == stack[top-1])
				Passed

			{
			stack[top]= storage[i].value;
			top++;
			}
			}
			gasUsed +=800;
			assert(gasUsed < gasLimit);
			}	

SSTORE	55	Create a new storage with the location set to the top of the stack, and value set to the second value on the stack	label_1:
			{
			storage[storePoint] = (struct pair){stack[top-1],stack[top-2] };
			top-=2;
			storePoint++;
			gasUsed +=20000;
			assert(gasUsed < gasLimit);
			}
				Passed

JUMP	56	Jump to determined label	label_1 :
			{
			gasUsed +=8;
			assert(gasUsed < gasLimit);
			goto label_57;
			pcCounter++;
			}
				Passed

JUMPI	57	-conditional jump-	-conditional jump-	No code, this is used for cfg's

PC	58	Set the top of the stack to the pcCounter value	label_1 :
			{
			stack[top] = pcCounter;
			++top;
			
			pcCounter++; gasUsed +=2;
			assert(gasUsed < gasLimit);
			}
				Passed

MSIZE	59	Set the top of the stack to amount of memory used so far	label_1 :
			{
			stack[top] = memPoint;
			++top;
			gasUsed +=2;
			assert(gasUsed < gasLimit);
			}
				Passed

GAS	5A	Store the gas used (after charging for this opcode)on the top of the stack	label_1:
			{
			gasUsed+=2;
				Passed

			`stack[top] = GASLIMIT- gasUsed;`
			`++top;`
			`}`	

JUMPDEST	5B	-conditional jump-	-conditional jump-	No code, this is used for cfg's
invalid	5C5D 5E5F	-	-	Passed

PUSH1	*6000*	stack[top] = 0x00;	`label_1:`
			`{`
			`stack[top] = 0x00;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

PUSH2	*61000 0*	stack[top] = 0x0000;	`label_1:`
			`{`
			`stack[top] = 0x0000;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

PUSH3	*62000 000*	stack[top] = 0x000000;	`label_1:`
			`{`
			`stack[top] = 0x000000;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

PUSH4	*63000 00000*	stack[top] = 0x00000000;	`label_1:`
			`{`
			`stack[top] = 0x00000000;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

PUSH5	*64000 00000 00*	stack[top] = 0x0000000000;	`label_1:`
			`{`
			`stack[top] = 0x0000000000;`
			`++top;`
				Passed

			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`	

PUSH6	*65000000000000*	stack[top] = 0x000000000000;	`label_1:`
			`{`
			`stack[top] = 0x000000000000;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

PUSH7	*6600000000000000*	stack[top] = 0x00000000000000;	`label_1:`
			`{`
			`stack[top] = 0x00000000000000;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

PUSH8	*6700000000000000*	stack[top] = 0x0000000000000000;	`label_1:`
			`{`
			`stack[top] = 0x0000000000000000;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

PUSH9	*6800000000000000000*	stack[top] = 0x000000000000000000;	`label_1:`
			`{`
			`stack[top] = 0x000000000000000000;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

PUSH10	*690000000000000000000*	stack[top] = 0x00000000000000000000;	`label_1:`
			`{`
			`stack[top] = 0x00000000000000000000;`
			`++top;`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

PUSH11	6A00000000000000000000	stack[top] = 0x00000000000000000000;	label_1:
			{
			stack[top] = 0x00000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH12	6B0000000000000000000000	stack[top] = 0x0000000000000000000000;	label_1:
			{
			stack[top] = 0x00000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH13	6C000000000000000000000000	stack[top] = 0x000000000000000000000000;	label_1:
			{
			stack[top] = 0x000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH14	6D00000000000000000000000000	stack[top] = 0x00000000000000000000000000;	label_1:
			{
			stack[top] = 0x00000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH15	6E0000000000000000000000000000	stack[top] = 0x0000000000000000000000000000;	label_1:
			{
			stack[top] = 0x0000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH16	6F000000000000000000000000000000	stack[top] = 0x00000000000000000000000000000000;	label_1:
			{
			stack[top] = 0x00000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH17	70000000000000000000000000000000	stack[top] = 0x00000000000000000000000000000000;	label_1:
			{
			stack[top] = 0x00000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH18	*71000000000000000000000000000000*	stack[top] = 0x00000000000000000000000000000000;	{
			stack[top] = 0x00000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH19	*72000000000000000000000000000000*	stack[top] = 0x00000000000000000000000000000000;	{
			stack[top] = 0x00000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH20	*73000000000000000000000000000000*	stack[top] = 0x00000000000000000000000000000000;	{
			stack[top] = 0x00000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH21	*74000000000000000000000000000000000000A0*	stack[top] = 0x0000000000000000000000000000000000000000;	{
			stack[top] = 0x0000000000000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH22	*7500..*	stack[top] = 0x0000000000000000000000000000000000000000000000;	{
			stack[top] = 0x0000000000000000000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH23	*7600..*	stack[top] = 0x000000000000000000000000000000000000000000000000;	{
			stack[top] = 0x0000000000000000000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH24	*7700..*	stack[top] = 0x00000000000000000000000000000000000000000000000000;	{
			stack[top] = 0x0000000000000000000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH25	*7800..*	stack[top] = 0x0000000000000000000000000000000000000000000000000000;	{
			stack[top] = 0x0000000000000000000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
				Passed

			} 	

PUSH26	*7900..*	stack[top] = 0x0000000000000000000000000000000000000000000000000000000000;	{
			stack[top] = 0x0000000000000000000000000000000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH27	7A	stack[top] = 0x0000000000000000000000000000000000000000000000000000000000;	{
			stack[top] = 0x0000000000000000000000000000000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH28	7B	stack[top] = 0x0000000000000000000000000000000000000000000000000000000000;	{
			stack[top] = 0x0000000000000000000000000000000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH29	7C	stack[top] = 0x0000000000000000000000000000000000000000000000000000000000;		Passed

PUSH30	7D	stack[top] = 0x0000000000000000000000000000000000000000000000000000000000;	{
			stack[top] = 0x0000000000000000000000000000000000000000000000000000000000;
			++top;
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

PUSH31	7E	stack[top] = 0x0000000000000000000000	{	Passed

		0000000000000000000000000000000000000000;	stack[top] = 0x0000000000000000000000000000000000000000000000000000000000000000; ++top; assert(top != STACKHEIGHT); gasUsed +=3; assert(gasUsed < gasLimit); }	
PUSH32	7F	stack[top] = 0x0000000000000000000000000000000000000000000000000000000000000000;	{ stack[top] = 0x0000000000000000000000000000000000000000000000000000000000000000; ++top; assert(top != STACKHEIGHT); gasUsed +=3; assert(gasUsed < gasLimit); }	Passed
DUP1	80	var tmp1 = stack[top - 1];	{ var tmp1 = stack[top - 1]; stack[top] = tmp1; ++top; assert(top != STACKHEIGHT); gasUsed +=3; assert(gasUsed < gasLimit); }	Passed
DUP2	81	var tmp1 = stack[top - 2];	{ var tmp1 = stack[top - 2]; stack[top] = tmp1; ++top; assert(top != STACKHEIGHT); gasUsed +=3; assert(gasUsed < gasLimit); }	Passed
DUP3	82	var tmp1 = stack[top - 3];	{ var tmp1 = stack[top - 3]; stack[top] = tmp1; ++top; assert(top != STACKHEIGHT); gasUsed +=3; assert(gasUsed < gasLimit); }	Passed
DUP4	83	var tmp1 = stack[top - 4];	{ var tmp1 = stack[top - 4];	Passed

			stack[top] = tmp1;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}	

DUP5	84	var tmp1 = stack[top - 5];	{
			var tmp1 = stack[top - 5];
			stack[top] = tmp1;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

DUP6	85	var tmp1 = stack[top - 6];	{
			var tmp1 = stack[top - 6];
			stack[top] = tmp1;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

DUP7	86	var tmp1 = stack[top - 7];	{
			var tmp1 = stack[top - 7];
			stack[top] = tmp1;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

DUP8	87	var tmp1 = stack[top - 8];	{
			var tmp1 = stack[top - 8];
			stack[top] = tmp1;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

DUP9	88	var tmp1 = stack[top - 9];	{
			var tmp1 = stack[top - 9];
			stack[top] = tmp1;
				Passed

			``
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
			``	

DUP10	89	var tmp1 = stack[top - 10];	```
			{
			var tmp1 = stack[top - 10];
			stack[top] = tmp1;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit); }
			``
				Passed

DUP11	8A	var tmp1 = stack[top - 11];	```
			{
			var tmp1 = stack[top - 11];
			stack[top] = tmp1;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
			``
				Passed

DUP12	8B	var tmp1 = stack[top - 12];	```
			{
			var tmp1 = stack[top - 12];
			stack[top] = tmp1;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
			``
				Passed

DUP13	8C	var tmp1 = stack[top - 13];	```
			{
			var tmp1 = stack[top - 13];
			stack[top] = tmp1;
			++top;
			
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
			``
				Passed

DUP14	8D	var tmp1 = stack[top - 14];	```
			{
			var tmp1 = stack[top - 14];
			stack[top] = tmp1;
			++top;
			``
				Passed

			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`	

DUP15	8E	var tmp1 = stack[top - 15];	`{`
			`var tmp1 = stack[top - 15];`
			`stack[top] = tmp1;`
			`++top;`
			
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

DUP16	8F	var tmp1 = stack[top - 16];	`{`
			`var tmp1 = stack[top - 16];`
			`stack[top] = tmp1;`
			`++top;`
			
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

SWAP1	90	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 2];
		stack[top - 1] = tmp2;
		stack[top - 2] = tmp1;
			`{`
			`var tmp1 = stack[top - 1];`
			`var tmp2 = stack[top - 2];`
			`stack[top - 1] = tmp2;`
			`stack[top - 2] = tmp1;`
			
			`assert(2<= top);`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

SWAP2	91	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 3];
		stack[top - 1] = tmp2;
		stack[top - 3] = tmp1;
			`{`
			`var tmp1 = stack[top - 1];`
			`var tmp2 = stack[top - 3];`
			`stack[top - 1] = tmp2;`
			`stack[top - 3] = tmp1;`
			
			`assert(2<= top);`
			`assert(top != STACKHEIGHT);`
			`gasUsed +=3;`
			`assert(gasUsed < gasLimit);`
			`}`
				Passed

SWAP3	92	var tmp1 = stack[top - 1];	`{` `var tmp1 = stack[top - 1];`	Passed

		var tmp2 = stack[top - 4];
		stack[top - 1] = tmp2;
		stack[top - 4] = tmp1;
			var tmp2 = stack[top - 4];
			stack[top - 1] = tmp2;
			stack[top - 4] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}	

SWAP4	93	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 5];
		stack[top - 1] = tmp2;
		stack[top - 5] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 5];
			stack[top - 1] = tmp2;
			stack[top - 5] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SWAP5	94	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 6];
		stack[top - 1] = tmp2;
		stack[top - 6] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 6];
			stack[top - 1] = tmp2;
			stack[top - 6] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SWAP6	95	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 7];
		stack[top - 1] = tmp2;
		stack[top - 7] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 7];
			stack[top - 1] = tmp2;
			stack[top - 7] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SWAP7	96	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 8];
		stack[top - 1] = tmp2;
		stack[top - 8] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 8];
			stack[top - 1] = tmp2;
			stack[top - 8] = tmp1;
				Passed

			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}	

SWAP8	97	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 9];
		stack[top - 1] = tmp2;
		stack[top - 9] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 9];
			stack[top - 1] = tmp2;
			stack[top - 9] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SWAP9	98	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 10];
		stack[top - 1] = tmp2;
		stack[top - 10] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 10];
			stack[top - 1] = tmp2;
			stack[top - 2] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SWAP10	99	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 2];
		stack[top - 1] = tmp2;
		stack[top - 11] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 11];
			stack[top - 1] = tmp2;
			stack[top - 2] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SWAP11	9A	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 12];
		stack[top - 1] = tmp2;
		stack[top - 12] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 12];
			stack[top - 1] = tmp2;
			stack[top - 12] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
				Passed

			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}	

SWAP12	9B	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 13];
		stack[top - 1] = tmp2;
		stack[top - 13] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 13];
			stack[top - 1] = tmp2;
			stack[top - 13] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SWAP13	9C	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 14];
		stack[top - 1] = tmp2;
		stack[top - 14] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 14];
			stack[top - 1] = tmp2;
			stack[top - 14] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SWAP14	9D	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 15];
		stack[top - 1] = tmp2;
		stack[top - 15] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 15];
			stack[top - 1] = tmp2;
			stack[top - 15] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SWAP15	9E	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 16];
		stack[top - 1] = tmp2;
		stack[top - 16] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 16];
			stack[top - 1] = tmp2;
			stack[top - 16] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

SWAP16	9F	var tmp1 = stack[top - 1];
		var tmp2 = stack[top - 17];
		stack[top - 1] = tmp2;
		stack[top - 17] = tmp1;
			{
			var tmp1 = stack[top - 1];
			var tmp2 = stack[top - 17];
			stack[top - 1] = tmp2;
			stack[top - 17] = tmp1;
			
			assert(2<= top);
			assert(top != STACKHEIGHT);
			gasUsed +=3;
			assert(gasUsed < gasLimit);
			}
				Passed

LOG0	A0	assert(0);	{
			
			assert(0);
			gasUsed +=375;
			assert(gasUsed < gasLimit);
			
			}
				Passed

LOG1	A1	assert(0);	{
			
			assert(0);
			gasUsed +=750;
			assert(gasUsed < gasLimit);
			
			}
				Passed

LOG2	A2	assert(0);	{
			
			assert(0);
			gasUsed +=1125;
			assert(gasUsed < gasLimit);
			
			}
				Passed

LOG3	A3	assert(0);	{
			
			assert(0);
			gasUsed +=1500;
			assert(gasUsed < gasLimit);
			
			}
				Passed

LOG4	A4	assert(0);	{
			
			assert(0);
			gasUsed +=1875;
			assert(gasUsed < gasLimit);
				Passed

			`}`	
invalid	A5A6 A7A8 A9A AAB ACA DAE AFB1 B2B3 B4B5 B6B7 B8B9 BAB BBC BDB EBFC 1C2C 3C4C 5C6C 7C8C 9CAC BCC CDC ECFD 1D2D 3D4D 5D6D 7D8D 9DA DBD CDD DEDF E1E2 E3E4 E5E6 E7E8 E9EA EBEC EDEE EF	-	-	Passed

CREATE	F0	accounts[accountNo] = (struct numpair){accountNo, 0.0};
		//balance is initially 0
			`{`
			`    accounts[accountNo] = (struct numpair){accountNo, 0.0};`
			`    accountNo++;`
			`    gasUsed +=32000;`
			`    assert(gasUsed < gasLimit);`
			`}`
				Passed

CALL	F1	-	-	out of scope
CALLCODE	F2	-	-	out of scope
RETURN	F3	exit(0);	`exit(0);`	Passed
DELEGATECALL	F4	-	-	out of scope
CREATE2	F5	accounts[accountNo] = (struct numpair){accountNo, 0.0};	`{ accounts[accountNo] = (struct numpair){accountNo, 0.0}; accountNo++; gasUsed +=32000; assert(gasUsed < gasLimit); }`	Passed
invalid	F6F7 F8F9	-	-	Passed
STATICCALL	FA	gasUsed +=40;	`gasUsed +=40; assert(gasUsed < gasLimit);`	out of scope
invalid	FBFC	-	-	Passed
REVERT	FD	exit(0);	`exit(0);`	Passed
INVALID	FE	-	-	Passed
SELFDESTRUCT	FF	-	-	Passed

Appendix B: Meeting Records

These notes were taken on my mac notes app, and are also in the pdf file meetingNotes.pdf:
Project research references:

https://unix.stackexchange.com/questions/229802/convert-executable-back-to-c-source-code - no variable names etc
Furthermore, eth.build is a nice learning platform for Smart Contracts with many examples.
https://ethereum.org/en/developers/docs/ethereum-stack/#ethereum-client-apis ??
Remix is an online environment for developing Smart Contracts. It has some example contracts that can be used as a starting points.
Decompiling : https://suif.stanford.edu/dragonbook/
CProver tools: http://www.cprover.org/cprover-manual/cbmc/tutorial/
Existing decompiler: **https://github.com/MrLuit/evm** ,
https://www.trustlook.com/services/smart.html , https://www.ethervm.io/decompile
https://github.com/crytic/rattle EVM static analysis of op codes

I'd start with Ethereum documentation and the Ethereum Yellow Paper

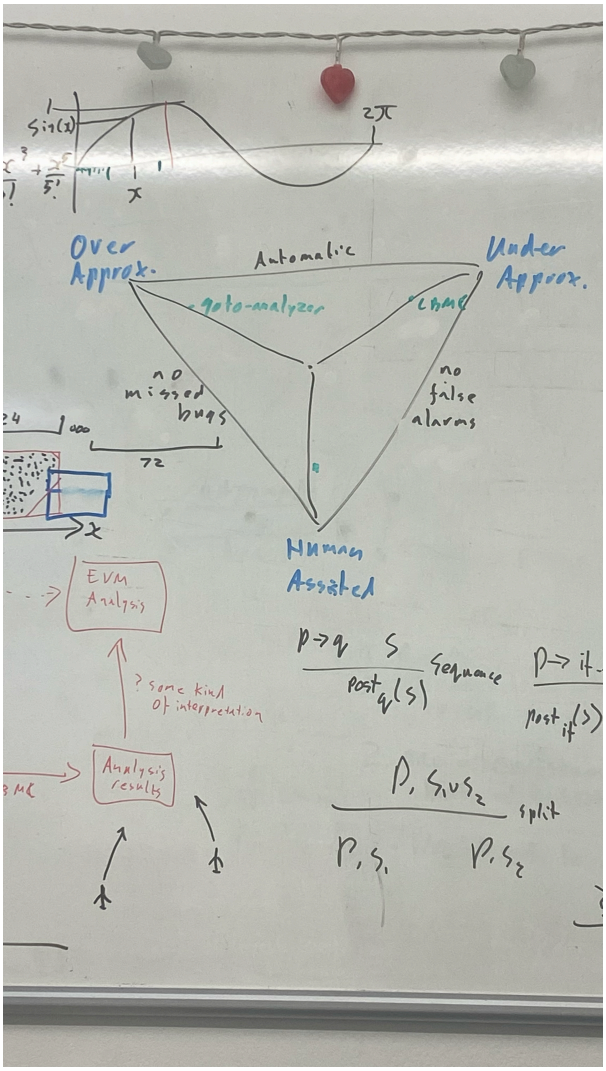
17th Feb meeting with Martin:
Decompiling .net is also more relevant to our project
Literature on Decompiling, look more at JVM Decompilation, will be closer technically (decompiling from binary i.e. executables) with JVM we have types and boundaries.

State of the art for elm or for solidity , what else is out there for that top link. Look at verification of EVM and Solidity (“competitors” of my approach) that directly do verification.
Existing systems (disassembles vs full Decompilation that return source code)
Approved PDD

28 Feb meeting : Design section should include:
How many intermediate formats are we using?
WHAT PARSES ARE WE DOING IN THE DATA AND HOW DO WE STRUCTURE THEM?
parsers, one that outputs it in the same format as the input, demonstrate that it works both ways
3rd parse: start trying to map to c , directly from disassembly
Tree structure it? CFG Recovery? Am I going to take my arrays and completely convert them? Control flow graph? When I print stuff out will it use the control flow graphs.
More of a Research project rather than class design
Getting the end to end, getting the intermediate formats right and the parsing correct.
Started with x , ended with Y
If there’s an unrecognised op code
2nd parse: Human readable (like the table)
Parse from disassembled to c , might not be the best or most useful output, but we will find this out earlier rather than later. Inform us on what Decompilation activities we want to do.
1st parse: Parsers written using visitor patient ,(compiler internals) opcode. Java has a visit method that allows you to pass something
Iterate over the array, make a switch when you come to an instruction. Explicit or using visitor function.

10th march
For example, when processing byte code like “ADD”, how do you deal with the values you are adding?

Now: Function where you feed in the string, from that you produce an array of objects of type op code - modify, so that instead it prints the semantics of each op code (research)
Check buildexample.txt to improve the code!
Parse in , generate it out in human readable



24th March
Add Automated testing
The Halting problem for normal programs - you cannot write a program that can take in an arbitrary Turing machine and tell you whether it will ALWAYS terminate
Relatively easy to prove , come up with an algorithm that works for some Turing machines (it's easy to prove some Turing machines terminate) - same applies to programs: if given a program with a loop, how do you know it will terminate?
Wind up with a pyramid (pyramid of verification, you can have 2 sides, but to have all 3 is difficult , e.g a verification tool that is automatic and has no missed bugs, might give false alarms (for example warning messages on visual studios- shows EVERY possible problem that could occur):

(See email from this date with example code and using prover)
See email from Martin for documents , and commands that were run on the prover tool, with the c file.

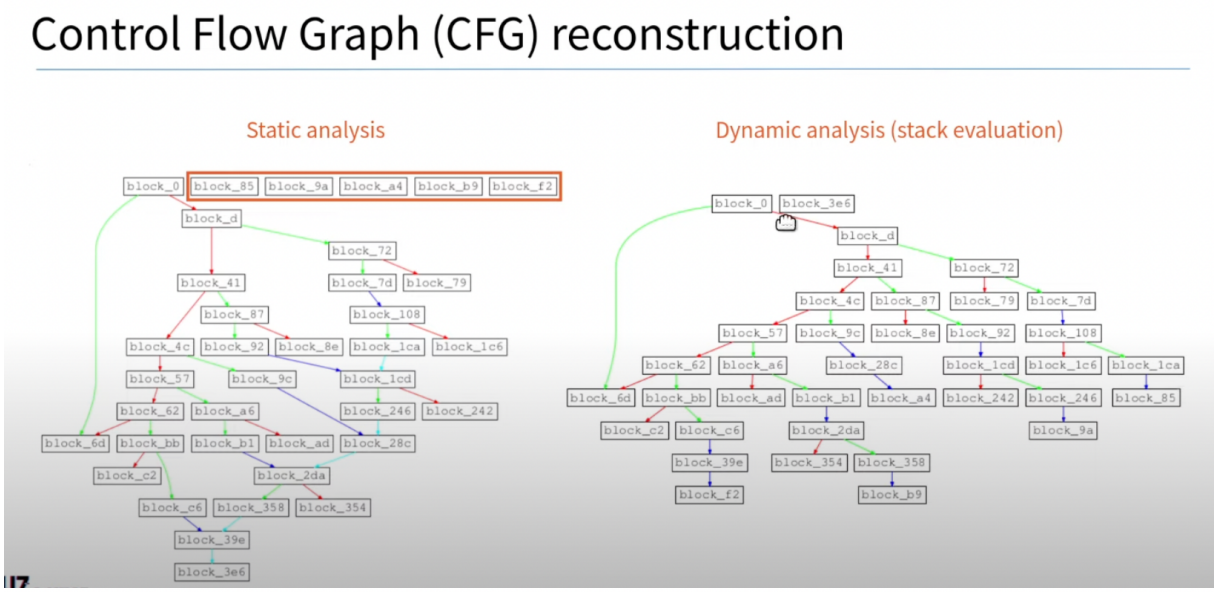
14th April:
Links for blockchain verification https://link.springer.com/book/10.1007/978-3-030-99524-9

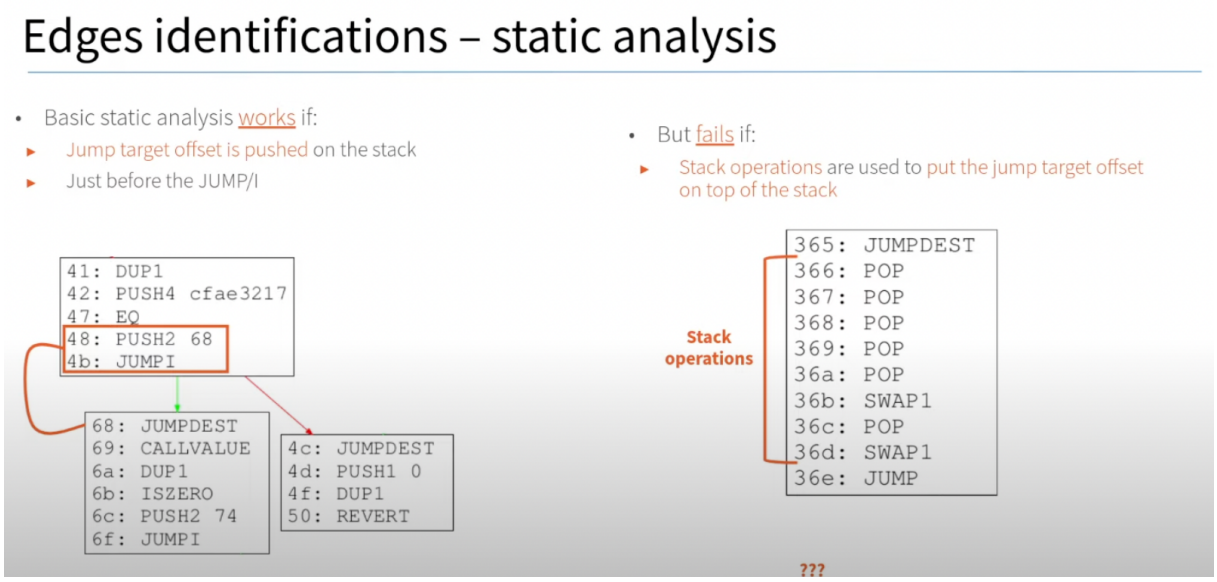
Ethereum/EVM Smart Contract Reverse Engineering & Disassembly - Blockchain Security #3
https://www.youtube.com/watch?v=I6VDBvX9Pkw&t=41s
Disassemble from EVM byte code to op code

Decompose into basic blocks of opcodes- Separate op coded based on:

Opcodes	Simplify description	Position within a Basicblock
JUMP	Unconditional jump	Last instruction
JUMPI	Conditional jump	Last instruction
RETURN , STOP INVALID SELFDESTRUCT , REVERT	Halt execution	Last instruction
JUMPDEST	Marks a position within the code that is a valid target destination for jumps	First instruction

CFG (CONTROL FLOW GRAPH) reconstruction, if you do not know the jump location, then you must use dynamic analysis rather than static, the example bellows shows that if we have the location then we can track the flow of the program:





Function identification and function names: based on the cfg's constructed we can identify functions

https://ethereum.stackexchange.com/questions/188/how-can-you-decompile-a-smart-contract

Compilation back to the original source code is impossible because all variable names, type names and even function names are removed. It might be technically possible to arrive at some source code that is like the original source code but that is very complicated, especially when the optimiser was used during compilation. I don't know of any tools that do more than converting bytecode to opcodes. ("How can you decompile a smart contract? - Ethereum Stack Exchange")
Since contracts can access their own code and thus (ab)use the code for storing data, it is not always clear whether some part of the code is used as code or only as mere data and whether it makes sense to try and decompile it. It is computationally undecidable whether some piece of the code is reachable or not. ("How can you decompile a smart contract? - Ethereum Stack Exchange")
Note that there is no dedicated area to store creation-time fixed data (like lookup tables, etc). Apart from the code of the contract, it would also be possible to store the data in storage, but that would be way more expensive, so putting such data in the code is a common thing.

Note: why do I write that you match on opcodes and not instructions (i.e. opcode and operand pair)? This is because if you are doing pattern matching you want to abstract a little bit; using the opcode for the instruction does that. In those cases where operand information should be included, what is done in the Python decompiler is that the opcode changes to reflect this additional piece of abstraction. There is nothing that dictates that you must match on existing EVM opcodes. You can make up new opcodes, insert opcodes that might indicate control structure boundary or change some opcode names to assist in pattern matching. ("How can you decompile a smart contract? - Ethereum Stack Exchange")

——28/04—
See example .c file, simulate the EVM stack first, then try to reconstruct functions, variables, etc…
Visitor , for every instruction

--Last meeting before submission 09/05 ---
Var needs to be a 256-bit integer : int64_t for executable , int128_t for compliable/analysable (have a switch between an executable and analysable , give an option to make runnable code vs Cprover code)

For unsignable, call them uint64_t, #include <stdint.h>
Interactive Commands with stack
1. work out how the values of the program counter correspond to the instructions
2. Create a map between program counter values / byte instructions and labels in the C.

[2.1. Catch and error on cases when a jump would take you into the middle of a multi-byte instruction.]

[2.2. Use the map to implement the PC instruction.]

3. Be able to identify whether the top element of the stack is a constant or not when a jump is called. It is fine to just handle the simple cases of this.

4. If you are jumping to a constant location, then you can use the compile time map to work out which label to goto

The map in 2 is working out which locations it is valid to jump to. This is what is described in section 9.4.3 in the yellow paper.

5. extend 4 so that conditional jumps are _if (stack[1]) { goto ___ ; }_

(Sorry conditional jumps to known fixed instructions!)

Handling computed / non-constant jump targets is Hard. IF you, do it, you should make a big deal about it in your write up.
Options...

6.A. Each time you come to a jump that you can't show to be to a constant location, generate code that uses a switch statement and go-tos for each possible location to handle it.

Using the map you built in 2...

```
switch (stack[top]) {
        case 0 : goto label_000;
        break; // no case 1 as 0 is a 2-byte instruction
        case 2 : goto label_002;
        break; .
}
```

SHA3:
Put in an existing implementation of SHA 3 (have a switch for either a computable vs analysable interpretation)
Uninterpreted function is best for an analysable solution (for Cprover only), output is non-deterministic, documentation for uninterpreted functions: http://www.cprover.org/cprover-manual/modeling/nondeterminism/)
example:
https://github.com/diffblue/cbmc/blob/develop/regression/cbmc/uninterpreted_function/uf2.c

A NOTIFICATION (Publish/ subscribe system that lets you know when things happen, the log writes to some notion of output)
Since contracts are passive, this is not as useful to implement so we simply use an assert so that we can at least show that the client is being “shown” something

- [x] LOGx: Append a log record with x topics, where x is any integer from 0 to 4 inclusive, can be a Simple assert statement: assert(0);
- [x] CREATE: Create a new account with associated code
Map of account number and a balance (pairs like the memory)

Scope meeting:
out of scope : “Message-call” and calls, and block commands- only to smart contract accounts, we have accounts that are limited to balance, this would require modelling of states, which requires a separate hash-map.

Block is out of scope, as you need the storage of every modification

Appendix B: Deployment Guide

This is found on my GitHub repository, in the readme file.

How to Compile and run the Decompiler Java program:

Requirements:
You must have Java , and the JVM downloaded on your device, see https://www.java.com/en/

Visual Code:

Simply open the src folder in Visual Studio Code, go to the decompile package and run the main function in the java fine : EvmDecompile.java. make sure you open the whole folder with visual studio, not just src.

Run in terminal:

1. Run a command to find all files with the extension .java once you have navigated to the correct src folder:

   find . -name "*.java"

2. Save these names in a text file:

   find . -name "*.java" >source.txt

3. Compile every java file using javac :

   javac @source.txt

4. Once the files have been compiled, run the EvmDecompile class, where the main function Is located:

   java -cp . src.decompiler.EvmDecompile

Example:

```
gerajahja@Geras-MacBook-Pro ~ % cd desktop                                    ]
gerajahja@Geras-MacBook-Pro desktop % cd DecompilingEVMtoCClone               ]
gerajahja@Geras-MacBook-Pro DecompilingEVMtoCClone % java -cp . src.decompile.Ev]
mDecompile
Enter your Ethereum contract bytecode (e.g 60806040...) :
```

Appendix B: List of libraries/ Software used

Libraries imported in code:

```
import java.util.*;
import java.io.*;
import java.util.concurrent.TimeUnit;
import java.math.BigInteger;
```

All files in the src/sha3Hash package are not written by me, they are from: https://github.com/mchrapek/sha3-java:

./sha3Hash/main/java/pl/thewalkingcode/sha3/Type.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/transformation/Rho.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/transformation/Pi.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/transformation/Theta.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/transformation/Iota.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/transformation/Chi.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/utils/HexTools.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/exceptions/AlgorithmInvalidState.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/operations/Absorb.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/operations/Squeeze.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/operations/Permute.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/operations/Padding.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/Config.java
./sha3Hash/main/java/pl/thewalkingcode/sha3/Sha3.java

Also the following function was taken from a tutorial, in EvmDissasemble.java lines 14-26

```
/*check is a string is a number , from :https://www.baeldung.com/java-check-string-
number */
    public static boolean isNumeric(String strNum) {
        if (strNum == null) {
            return false;
        }
        try {
            double d = Double.parseDouble(strNum);
        } catch (NumberFormatException nfe) {
            return false;
        }
        return true;
    }
```

List of java files written by me: (all in the src folder):

./decompile/EvmDisassemble.java
./decompile/EvmDecompile.java
./decompile/visitOpCode.java
./tests/visitorTestAdd.java
./cCodes/structs.java
./cCodes/storage.java
./cCodes/label.java
./cCodes/include.java
./cCodes/account.java
./cCodes/variable.java
./cCodes/assertVal.java
./cCodes/memory.java
./cCodes/function.java
./cCodes/stack.java
./cCodes/returnVal.java
./cCodes/define.java
./opcodes/sdiv.java
./opcodes/pop.java
./opcodes/callvalue.java
./opcodes/stop.java
./opcodes/mul.java
./opcodes/shr.java
./opcodes/revert.java
./opcodes/extcodesize.java
./opcodes/selfbalance.java
./opcodes/jumpdest.java
./opcodes/mstore8.java
./opcodes/timestamp.java
./opcodes/return_.java
./opcodes/callcode.java
./opcodes/calldataload.java
./opcodes/returndatacopy.java
./opcodes/add.java
./opcodes/sstore.java
./opcodes/swap.java
./opcodes/dup.java
./opcodes/iszero.java
./opcodes/pc.java
./opcodes/codesize.java
./opcodes/mstore.java
./opcodes/exp.java
./opcodes/calldatasize.java
./opcodes/create.java
./opcodes/gt.java
./opcodes/chainid.java
./opcodes/shl.java
./opcodes/coinbase.java
./opcodes/div.java
./opcodes/jump.java

./opcodes/slt.java
./opcodes/basefee.java
./opcodes/address.java
./opcodes/extcodehash.java
./opcodes/difficulty.java
./opcodes/addmod.java
./opcodes/push.java
./opcodes/returndatasize.java
./opcodes/call.java
./opcodes/number.java
./opcodes/staticcall.java
./opcodes/gaslimit.java
./opcodes/signextend.java
./opcodes/sar.java
./opcodes/delegatecall.java
./opcodes/and.java
./opcodes/sub.java
./opcodes/jumpi.java
./opcodes/codecopy.java
./opcodes/calldatacopy.java
./opcodes/invalid.java
./opcodes/xor.java
./opcodes/gasprice.java
./opcodes/log.java
./opcodes/selfdestruct.java
./opcodes/bytee.java
./opcodes/smod.java
./opcodes/caller.java
./opcodes/gas.java
./opcodes/not.java
./opcodes/mod.java
./opcodes/blockhash.java
./opcodes/balance.java
./opcodes/mload.java
./opcodes/eq.java
./opcodes/origin.java
./opcodes/extcodecopy.java
./opcodes/create2.java
./opcodes/lt.java
./opcodes/sload.java
./opcodes/or.java
./opcodes/sh3.java
./opcodes/mulmod.java
./opcodes/msize.java
./opcodes/sgt.java
./zdrafts/testingPush.java
./zdrafts/visitorTestAdd.java
./interfaces/Dissasemble.java
./interfaces/GetInstructionsFromOpcode.java