DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CAPTURING THE STATE OF THE ETHEREUM ECOSYSTEM

LUKE GWALTNEY
SUMMER 2018

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Engineering
with honors in Computer Science

ABSTRACT

The Ethereum protocol has emerged as the most promising way to develop decentralized applications (also called smart contracts) onto a blockchain. However, the definitions of how these applications should behave remain unclear. Already there have been cases in which a vulnerability in one smart contract resulted in the locking up or theft of several millions of US dollars' worth of cryptocurrency. Several tools have been constructed to detect similar bugs in these contracts before they are published to the blockchain. Nevertheless, these tools can be improved upon to ensure more security for the Ethereum ecosystem. A step in the right direction is developing a firm understanding of how these contracts behave in practice.

In this study, we give a brief overview of the available security tools for smart contracts. We then parse the Ethereum blockchain's transaction history to gather Ether balance, popularity, and an instruction count for each contract. By doing this, we hope to find a relationship between the characteristics of a contract and the likelihood the contract is flagged for a vulnerability by the current contract security tools. The relationship, or relationships, that we find with this data can be used to develop deeper security tools that better define how these smart contracts should behave.

TABLE OF CONTENTS

LIST OF FIGURES

LISTINGS

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Dr. Patrick McDaniel for pushing me to make this paper the final product that it is. I would also like to thank Eric Pauley, Quinn Burke, and Jake Levenson for working with me to discover the nature of blockchain and for carrying this project forward. Thanks also to Berkay Celik, for helping me in the final stages of the submission process. And finally, thank you to my mother and father, who don’t know anything about smart contracts, but have supported me in everything I do.

CHAPTER ONE

Introduction

On December 17, 2017, the price of Bitcoin hit an all-time high of $19,783.21. The same year, it crossed the $1000 mark on New Year's Day [10]. Similar cryptocurrencies experienced market highs in the last quarter of the year. With the value that investors are pouring into these blockchain-based platforms, there is an increased responsibility for engineers and security professionals to analyze these kinds of environments. Otherwise, the consequences could be severe.

In fact, consequences have already been severe. In July of 2016, a vulnerability in one of the smart contracts built on the Ethereum network led to a robbery of approximately $60 million in Ether [4]. Another bug systematically locked $200 million in Ether that could no longer be accessed by the rest of the network [8]. In response to these events, security tools have been developed whose purpose is to define how participants of these crypto-networks can and should interact with one another in safe ways. These tools have been tailored specifically to smart contracts on the Ethereum network by using program analysis on Ethereum-specific bytecode, but their heuristics can be used as the applications of these technologies continue to develop.

In this paper, we seek to summarize the aforementioned tools and collect characteristic data on the code they analyze, in hopes to suggest ways to improve them. It is first necessary, however, to give a complete tutorial on the blockchain, Ethereum, and smart contracts for the reader's understanding.

CHAPTER TWO

Background

2.1 The Blockchain

In 2009, the world of online commerce was changed forever when Satoshi Nakamoto published the Bitcoin white paper. The text proposed a systematic way for users to complete transactions directly with one another that did not require the intervention of a third party but instead built its trust system on cryptographic proof [1]. In the place of a central authority that minted new currency and authorized transactions, all participants would algorithmically validate each transfer of funds and creation of new coins. This was made possible by the innovation of a new kind of cryptographically secure data structure: the blockchain.

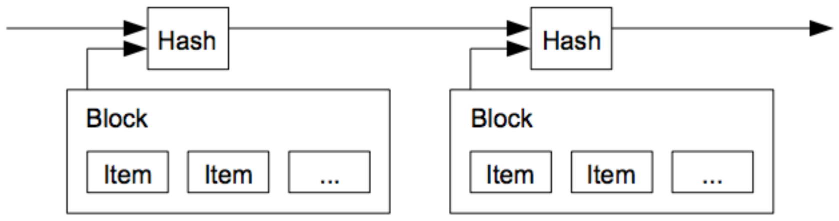


Figure 1: A schema of blockchain architecture [1]

A blockchain is a series of blocks that are chained together by a timestamp-based hash algorithm. This is demonstrated in *Figure 1*. Each block is a data structure that contains items that have information pertinent to that specific block. These blocks are connected by a hash of the current block’s timestamp and the hash of the block before it. For Bitcoin, the blockchain is an electronic ledger of the system’s transactions. The individual blocks contain a list of transactions that share the coins associated with that block. New coins are minted as new blocks are discovered and appended to the end of the chain. This process is called mining.

Mining occurs when members of the Bitcoin network compete with one another to find the next block on the chain. Once a competitor finds a block, other members validate the new block and begin competing for the next block on the chain. More specifically, competitors will try several different nonces for their hypothetical new block that, once hashed with the previous block’s hash, generate a result that begins with a significant number of zero bits. Once this nonce is found, the founder adds this new block and broadcasts the new blockchain to the rest of the network, which validates the founder’s success and starts the process all over again.

The reasoning for mining as a way to mint new coins is simple: it incentivizes participants of the Bitcoin network to use their computing power to expand the world of Bitcoin rather than to launch attacks against it [1]. Once a block is mined, the miner of the block is rewarded the coins of that block.

Bitcoin’s innovation paved the way for several blockchain-based technologies to follow. The most notable of these is Ethereum.

2.2 Ethereum: A Blockchain Platform

Ethereum is not a cryptocurrency. It is instead a blockchain-backed platform on which users can build decentralized applications using a Turing-complete scripting language [2]. It operates using its own cryptocurrency, Ether, which can be used to not only to transfer funds from one Ethereum account to another but also to pay the computational costs for a transaction.

The Ethereum network is, much like Bitcoin's, made up of several addressable accounts. Each account has a unique nonce, ether balance, contract code, and storage space. These accounts can be broken up into two distinct categories. There are externally owned accounts, which can be thought of simply as users that are able to send and receive messages and Ether to and from other accounts. There are also contract accounts. These contract accounts, commonly referred to as smart contracts, use their contract code to execute tasks every time they are sent messages from another account. These tasks can include spinning off other messages to other accounts, reading and writing data to the contract account's internal storage, or even creating new contracts [3].

This platform can be described as a transaction-based state machine. A transaction in Ethereum is an instruction, cryptographically signed by its sender, which can either send messages or create new contracts. In either case, the sender specifies the receiver of the transaction, an amount of Wei (10^{-18} Ether) to be sent to that receiver, as well as two fields (gasLimit and gasPrice, described later) that determine the transactional fee sent to the miner of the transaction's block based on the computational cost of that transaction. When the transaction creates an account, an additional field is specified that contains that contracts code.

These transactions act as the transition function for Ethereum's various states. There are two kinds of states in Ethereum: account states and the world state. The world state is simply a mapping from network addresses to their account states, while these account states are individual data structures that hold a user’s (or contract’s) current balance and number of performed transactions. This state-like behavior of Ethereum is important because it self-regulates what transactions can and cannot do by defining illegal state transitions. For instance, an account cannot send an amount of ether that is greater than its current balance. A transaction must also include enough Wei to pay for computational fees.

Because Ethereum uses a Turing-complete programming language to execute contracts, there needs to be a way to set a limit on how long any given contract can run. This is the idea behind gas. All transactions must define how much Wei it will pay for each instruction of Ethereum bytecode (gasPrice) as well as a limit (gasLimit) on how much total Wei should be spent. These dictate the total price in Wei that a sender will pay to a miner for including a transaction on his block as well as actually executing said transaction. At the beginning of a transaction, the miner is sent an amount of Wei equal to the gasLimit. As the transaction is executed, this value is decremented according to the gasPrice. If the miner runs out of gas before the execution is finished, the transaction fails and the account state is reverted to the original state before any execution of the transaction. Otherwise, the transaction is completed, the account state is changed accordingly, and any leftover gas is refunded to the sender [2].

Because Ethereum transactions can be used for not only transferring funds but also anything that is computationally representable, the applications for this platform are limited only by the imagination of the smart contract author. One of the most notable, however, is the Decentralized Autonomous Organization, or DAO. At the very beginning of the Ethereum yellow paper, the author hints at Ethereum being the solution for “algorithmic enforcement of agreements” that would become “a significant force in human cooperation” in the future [3]. A DAO uses the power of Ethereum to define this kind of solution.

A Decentralized Autonomous Organization is a smart contract that defines a community of shareholders that can submit votes to change contract code or spend the organizations’ funds [2]. Think of a board of executives that decide the future of a company. However, with the DAO, there is no need for a chief executive to consider the votes and make the final call. Subdivisions of this idea include Decentralized Autonomous Communities, which sets the voting power of all shareholders to be the same; there are also Decentralized Autonomous Corporations, which weigh certain shareholders’ votes differently based on ownership and rank within the corporation.

This assumes of course that all Ethereum users can agree on the same execution for all smart contracts and transactions. Users come to this consensus by each running the code for these instructions on the Ethereum Virtual Machine.

2.3 The Ethereum Virtual Machine and Smart Contracts

Each participant on the Ethereum network uses the Ethereum Virtual Machine (EVM) to run all the smart contracts referred to by the blockchain. The EVM runs on machine-specific bytecode which is heavily stack based. A complete listing of the instructional set is provided in Appendix A as it is specified in the Ethereum Yellow Paper [3]. It includes instructions for arithmetic operations, comparison and bitwise logic operations, stack and memory operations, system operations, and instructions for obtaining transaction and block information.

System operations are what make the transfer of Ether from one account to another possible. Four of these instructions are important to understand when considering system security: CALL, CALLCODE, DELEGATECALL, and SELFDESTRUCT. CALL is used to message call into a specified account. This can be used to send a transaction to an account and, if it is a contract, run that account's code. CALLCODE is similar to call, but instead message calls into the same contract using the code of a different contract. The same is true for DELEGATECALL, but it, unlike CALLCODE, keeps the transaction's values for sender and value. SELFDESTRUCT halts the execution of a contract and sends its Ether balance to an existing account.

It is unusual for a smart contract programmer to write a contract in bytecode. They instead use higher-level languages that compile to EVM bytecode, the most popular of which is Solidity. In Solidity, each contract can consist of fields and functions. However, EVM bytecode does not support functions. To handle this, one of the first steps the Solidity compiler takes is creating a mechanism to match functions using their function signatures. If this mechanism cannot match a given signature to a function in a given contract, it invokes the contract's fallback function. All contracts have a fallback function that has no name and no arguments and is user-defined [4].

Solidity has two main functions that are used to send messages from contract to account: *call* and *send*. The *call* function is used when a contract would like to invoke a specific function of another contract, while *send* will simply transfer an amount of Ether to another contract and invoke its fallback function [4].

Listing 1: A simple smart contract in Solidity

```
pragma solidity ^0.4.20;

contract Bob {
     address Alice;

     function() public payable {
          Alice.send(msg.value);
     }
}
```

Listing 1 gives an example of an extremely simple contract. Bob’s contract has declared an address variable that can be used to call and send Ether to Alice’s contract. In Bob’s fallback function, he sends to Alice the Ether amount of the function caller. Therefore, Bob’s contract sends Alice all the funds that it receives.

These contracts are not always so simple, but even the simplest smart contracts can be written in ways that are easily exploitable by a malicious attacker. Because the nature of these contracts is to transfer funds between users, it is of the utmost importance that the authors of these contracts are aware of their vulnerabilities.

2.4 Smart Contract Vulnerabilities

Smart contracts are public to all participants on the Ethereum network, and they are somewhat permanent once published to the blockchain. This makes dealing with vulnerable contracts particularly difficult. Though only public in the form of EVM bytecode, this is still potentially dangerous because the bytecode is not too challenging for an attacker to interpret. Any bugs that are caught after an author publishes their contract to the blockchain cannot simply be patched. They must be removed altogether or altered in a way coherent with its original implementation. This being said, there are four main vulnerabilities that authors should take into consideration when designing their contracts: exception handling, timestamp dependence, transaction order dependence, and reentrancy probability.

Exceptions can be raised throughout the execution of a contract. The reasons for an exception vary. A contract call can run out of gas before it has fully executed. The limit of the call stack, which is 1024 frames in the Ethereum Virtual Machine, can be exceeded. The list goes on. However, Solidity and contract programmers are not uniform in the way they handle these exceptions. For instance, the two functions *send* and *transfer* are two similar ways that Ethereum smart contract authors can send Ether amounts to different accounts. The transfer function automatically checks for a return value, notifying the sender when an exception has occurred. However, programmers have to check the return value of a *send* manually [7].

Listing 2: An exception-mishandling King of the Ether Throne contract

```
pragma solidity ^0.4.20;

contract KingOfTheEtherThrone {
        address public king;
        uint public prize = 100;
        address owner;

        function KingOfTheEtherThrone() {
                owner = msg.sender;
                king = msg.sender;
        }

        function () public payable {
                if (msg.value < prize) throw;
                king.send(prize);
                king = msg.sender;
                prize = prize + 5;
        }
}
```

The primary malicious example that has been used in several papers [4, 5, 7] is the King of the Ether Throne. A very simple example of this is depicted in *Listing 2*. It works by first defining a king who holds the throne. In order to be dethroned, another account has to pay the prize, initially set to 100 Wei. The prize is increased by five Wei each time the king changes, contract effectively granting the last king 5 Wei each time he holds the throne. In this way, the contract can be thought of as a linear Ponzi scheme.

The potential vulnerability in this contract lies in the compensation to the previous king. Note that the *send* function is used to transfer the prize amount to the previous throne holder. Because the return value of this *send* invocation is not checked by the contract's code, there is no way for it to know whether the transfer of Wei was executed, and the king was granted the proper prize. Therefore, a malicious contract could very easily send the correct prize to the contract if he knew that this send would not be completed because an exception would be thrown in the previous king’s fallback. If this were the case, the prize transfer would not go through, the malicious contract would keep its Wei, and the previous king would be dethroned without compensation.

Contract coders also are limited in terms of randomness. They must instead generate pseudo-random numbers using an initialization seed. A common choice of seed for this is the timestamp of the block on which the contract resides. At first, this may seem a secure way to generate a random number. However, miners in Ethereum are free to select the timestamp associated with their block so long as it is later than the timestamp of the previous block and no more than 2 hours ahead in the future [2]. This being said, it is not too bold to say that a malicious miner could adjust the timestamp of the block in order to affect the randomness of any pseudo-random number generator included on the block.

A similar problem with the blockchain nature of Ethereum is that the user of a transaction cannot accurately determine when the contract they invoke will be enacted, and therefore cannot be sure in what kind of state the contract will be. This kind of vulnerability is referred to as transaction-order dependency. This is because transactions are not added to blockchain atomically, and therefore race conditions can occur between two separate transactions.

For instance, the TriviaGame contract depicted in *Listing 3* shows a simple game that exchanges the correct solution to a trivia question for an Ether reward. The game host can set the correct solution and corresponding reward, while participants can send in their guesses. The rewards those participants when their guess is true. This is all handled in the fallback function of the contract.

However, consider the scenario when the game host sends Transaction A, which updates the solution and reward, to the TriviaGame contract at the same time a participant sends Transaction B, which guesses the correct solution. Because the order which the contract’s block’s miner chooses to include these two transactions is arbitrary, and the transactions themselves do not occur atomically, it is uncertain whether or not the participant that sent Transaction B will be compensated for guessing the question correctly. This, of course, is an innocent example, but the kind of vulnerability could potentially be extremely dangerous.

Listing 3: A transaction-order dependent contract

```
pragma solidity ^0.4.20;

contract TriviaGame {
      address public gameHost;
      bytes public solution;
      uint public reward;

      function () public payable {
            if (msg.sender == gameHost) {
                  solution = msg.data;
                  reward = msg.value;
            } else {
                  if (keccak256(msg.data) == keccak256(solution)) {
                        msg.sender.transfer(reward);
            }
      }
}
```

Another important vulnerability to consider is reentrancy. This vulnerability has received a lot of attention from the cybersecurity field after the attack on a theDAO contract in July of 2016. The contract was a crowd-funding platform that had raised approximately $150 million in Ether before an attacker gained control of almost $60 million of that by exploiting a reentrancy bug [4].

The basic idea of reentrancy is that the fallback method of a contract can allow an attacker to re-enter a host contract after the initial call. This can result in a loop of repeated withdraws of information or, as in the case of the June 2016 theDAO exploit, Ether.

Reentrancy is best explained with an example. A commonly used [5, 6] contract which has been analyzed as reentrancy-prone is pictured in *Listing 4*. It depicts a very simple bank, which has multiple accounts linked to Ethereum addresses. From this bank, an account holder can choose to add a certain amount of Ether to their account, view the current balance of their account, or withdraw all of their current balance. However, there exists a bug in the way this balance is withdrawn.

Listing 4: A reentrancy-prone banking contract

```
pragma solidity ^0.4.20;

contract Bank {
     mapping (address => uint) accounts;

     function addToBalance() public payable {
          accounts[msg.sender] += msg.value;
     }

     function balanceInquiry() public view returns (uint) {
          return accounts[msg.sender];
     }

     function withdrawBalance() public {
          if (msg.sender.call.value(accounts[msg.sender])()) {
               accounts[msg.sender] = 0;
          }
     }
}
```

It is important for one to note that the caller’s account is set to zero after the value of the account is sent to the caller in the if-statement. It is put in an if-statement here to check that the call to send the balance does not return false (in the case of an exception that does not complete the transaction). Because this account is zeroed after the account’s contract is called, it is possible for that contract to withdraw continuously from the Bank until the gas limit is reached or the Bank runs out of funds.

In *Listing 5*, it is evident how an outside contract would quickly be able to gain control over a majority of the funds from the Bank in *Listing 4*. Mallory can first call addAmount to deposit an amount of Ether to her bank account. She can then call the fallback function (which can actually be called by any other contract when they send Mallory Ether) which will withdraw the amount she originally deposited. This fallback will then be continually called by the if-statement condition in the Bank’s contract (line 15) as the condition sends the Ether to Mallory, again invoking her fallback function. This loop will only terminate when an out-of-gas exception is raised or all the Ether in the Bank is withdrawn.

Listing 5: A malicious contract exploiting the contract in Listing 4

```
pragma solidity ^0.4.20;

contract Mallory {
        Bank public bankAccount;

        function addAmount(uint donation) public payable {
                bankAccount.addToBalance.value(donation);
        }

        function () public payable {
                bankAccount.withdrawBalance();
        }
}
```

There are other vulnerabilities to be aware of as well. Typecasting in Solidity and other higher-level contract languages can be checked at compile time, but in many cases, they do not check addresses nor do they match interfaces. Type mismatches are handled in different ways. Making fields within your contract private is a possibility, but since the contract is published to the blockchain, the values of these fields can be inferred by potential attackers [4].

So far, measures have been taken in order to catch the above bugs before being published to the Ethereum blockchain. Some measures have been internal to Ethereum and the bytecode. For instance, there was a hard-fork of the Ethereum blockchain in October 2016 to address problems that were arising because of the call stack size being exceeded. After the fork, limits were placed on how much gas a caller could allocate to ensure the limit of 1024 frames would not be exceeded

[4]. There have also been external tools created that use program analysis to determine if individual contracts are safe from the above vulnerabilities.

CHAPTER THREE

Current Security Solutions

3.1 Oyente

The first external tool that is beneficial to examine is Oyente. Oyente serves as a pre-deployment mitigation tool that seeks to help smart contract developers write better contracts, as well as avoid publishing contracts with serious vulnerabilities [5]. It does so by performing symbolic execution on contracts to determine whether execution paths are feasible or infeasible.

More specifically, the Oyente paper *Making Smart Contracts Smarter* [5] breaks Oyente's analysis down into four discrete steps. First, it builds a rough control flow graph that statically breaks the EVM bytecode into basic blocks of execution (graph nodes) and jumps (graph edges). Further edges may be constructed later if they cannot be determined in this step.

Next, an explorer does a depth-first search of the control flow graph, possibly with several different states at a time. This explorer utilizes Microsoft's Z3 theorem prover in order to determine whether branch conditions are definitively true or false along the given path. If neither is the case, both branch paths are explored. It is here where edges additional to the ones constructed in the first step are added. When it is finished, the explorer creates a set of feasible traces through the contract's bytecode.

These traces are then analyzed to figure whether the contract suffers any of the vulnerabilities mentioned in the last chapter: transaction-order dependency, timestamp dependency, exception handling, or reentrancy. To determine whether or not a contract is transaction-order dependent, Oyente checks if two different traces have different Ether flows. If this is the case, the contract is said to be transaction-order dependent. Timestamp dependence is even simpler to detect: if the block's timestamp is variable in any of the contract's traces, the contract is flagged for this vulnerability. For exception disorders, Oyente determines if the return value of a call is checked by the caller. If it is not, the contract is flagged for an exception vulnerability.

Lastly, the core analysis checks for a reentrancy vulnerability. This is somewhat tougher to diagnose. Oyente does so by checking path conditions before each call. If the condition has a variable that is mutated by the call, it is likely that the code executed from that call can re-enter the calling contract and make the same call to itself. Therefore, any contract with conditions that can be influenced by the calls that happen after them is flagged as vulnerable.

Though Oyente has defined a rigorous analysis of contracts and is far better at diagnosing some kinds of vulnerabilities than others, it is not perfect. For instance, when the tool against 19,366 smart contracts at the tools inception, it flagged 1,385 distinct circumstances for mishandled exceptions, 135 for transaction-order dependencies, 52 for timestamp dependencies, and 186 for reentrancy bugs [5]. However, because not all the contracts they tested had their source code (in Solidity) published and readily available, they were unable to verify all the contracts they tested.

Of the contracts with source code available, Oyente was able to confirm that the 116 contracts flagged with exception mishandling that had source code available were true positives. The seven published contracts that were flagged for timestamp dependencies were also confirmed as true positives. However, there were nine confirmed false positives compared to only 23 confirmed true positives in the case of transaction-order dependency, and only one of the two published contracts with a reentrancy bug was actually exploitable [5]. This one confirmed exploitable reentrancy vulnerability was the theDAO contract.

3.2 Porosity

Another open-source external tool is Porosity, developed by Comae Technologies in July of 2017 [6]. It, like Oyente, seeks to define and detect vulnerabilities in smart contracts that can be exploited by malicious contract owners.

Porosity claims to be a decompiler that can generate Solidity code from the original EVM bytecode one would find on the Ethereum blockchain. From the Solidity output, it can perform static and dynamic program analysis. It works by taking the binary of a contract's runtime code and disassembling it into a readable output. From there, it looks at the runtime dispatcher of each contract. This dispatcher is used to select the correct function to use when a contract is invoked. Porosity uses this as well as ABI definitions of the contracts functions to generate function hashes.

Similar to Oyente, Porosity then uses jump instructions in the disassembled bytecode to create static, then dynamic, control flow graphs. It can then translate these graphs into pseudo C code. From this code, the tool can detect bugs such as reentrancy, call stack vulnerabilities, and time dependence vulnerabilities [6].

Porosity's white paper fails to provide any of the accuracy statistics that are provided by Oyente. However, because the two are similar in their methodology and vulnerability definitions, it is reasonable to believe that Porosity does not perform significantly better than Oyente.

3.3 Maian

While Oyente and Porosity are similar in their approach to diagnosing vulnerabilities in smart contracts, Maian is the first tool that explores what it defines as trace vulnerabilities, or vulnerabilities that are a result of many invocations of the same contract over its lifetime [8]. These trace vulnerabilities are broken into three categories. There are prodigal contracts, which are designed to give Ether away to arbitrary addresses that the contract has not interacted with in the past. Almost all the aforementioned vulnerabilities fall under this prodigal category. There are also suicidal contracts, which can be killed by another contract, and greedy contracts, which are only able to consume Ether and have no way to dispense of it. This leads to a locking of funds that in one instance locked away $200 million worth of Ether [8].

The tool has roughly two steps: symbolic execution and concrete validation. In the symbolic execution stage, Maian takes the bytecode of the contract it is analyzing along with certain analysis specifications as input and runs every possible execution trace with a set of symbolic variables on a custom Ethereum Virtual Machine. It is important to note here that Maian does not keep track of the entire blockchain state for tractability reasons: it only symbolically represents the contract's transactions and certain block characteristics as inputs. Once it finds a trace that satisfies a set of predetermined properties, it flags that contract, generates concrete variables for the given trace, and passes those along to the concrete validation stage that tests it on a private fork of the EVM. This validates it as a true positive or rejects it as a false positive.

In the concrete validation stage, Maian is testing concrete values as inputs to the contract and determining whether the contract behaves vulnerably as it was flagged. Maian owns accounts on the private Ethereum blockchain that it uses to send the flagged contract a transaction with the inputs that were previously specified by the symbolic execution stage. If the contract was flagged as prodigal, the concrete validator determines whether the contract leaks Ether to the engine's account after this transaction is processed. If the contract was flagged as suicidal, the engine checks the contracts bytecode after the transaction was sent. If it has been killed and is indeed suicidal, the bytecode will be reset to '0x'. If the contract was flagged as greedy, the engine checks whether the contract has a way to accept Ether, but does not have CALL, DELEGATECALL, or SELFDESTRUCT in its bytecode. If this is the case, it is a true positive, as it has no way to deal out the Ether it is collecting.

The authors of Maian claim impressive true positive rates, which is not surprising given their thorough algorithm. After analyzing 970,898 smart contracts, Maian was able to identify 1,253 prodigal contracts, of which 97% were validated to be true positives. It also verified that 99% of the 1,423 contracts it flagged as suicidal were true positives. Of the 1,083 contracts Maian identified as greedy, 69% of those were true positives [8]. Though the latter percentage is less than the other two, a total 89% true positive rate is quite good, and a significant improvement on Oyente.

CHAPTER FOUR

Vulnerability Analysis of Smart Contracts

4.1 Motivation

It is clear that of the external tools developed to detect vulnerabilities in existing smart contracts, Maian is the most accurate. It is logical then that any future solutions developed for this purpose would be wise to use Maian as a template. We believe a major improvement on Maian and the existing tools would be to further analyze inter-contract relationships. As discussed previously, the current tools are great at programmatically analyzing a contract’s intra-contract execution using control flow graphs and symbolic execution. This is extremely useful when trying to detect vulnerabilities such as timestamp dependence and exception mishandling.

However, the nature of these contracts is that they run code based on transactions they receive from and send to other accounts on the Ethereum network. This can be done by using the CALL opcode. In addition, many contracts use library contracts that they access using a DELEGATECALL or CALLCODE. These library contracts have various applications, one of which might be defining a token other than Ether. Once called, the library contracts effectively dump their code into the calling contract.

While many have designed analysis around this behavior, the current tools do not analyze the code that the contract being scrutinized obtains from a DELEGATECALL or similar instruction. For instance, Maian uses the presence of DELEGATECALL to validate greedy contracts, but it does not support it, and many other EVM instructions, in terms of symbolic execution.

Therefore, in order to improve on what has already been designed, it is necessary to gather data on the characteristics of the smart contracts on the current Ethereum blockchain to gain insight on how these contracts are being used in practice. More specifically, we would like to gather statistics such as how much Ether each contract has, how many times each contract is referenced by other contracts, and how often these contracts are using each instruction of the EVM instruction set. We then plan to run all the contracts on the Maian tool.

We believe that in doing this, we will find a great amount of deviation in the complexity and wealth of active contracts and that most of the vulnerabilities that Maian is able to detect are in simpler, lesser used, and less wealthy contracts. This is important because if Maian is finding the bulk of vulnerabilities in contracts that are simple to analyze and that don’t have a tremendous amount of Ether, the tool is not particularly useful. We are suspicious that this might be the case, as Maian’s current algorithm does little to analyze inter-contract relationships. We also are expecting to find that a relatively large percentage of contracts in the Ethereum ecosystem are using instructions that Maian and other tools do not support, such as DELEGATECALL. If this is the case, we believe those contracts are vulnerable in a way that Maian would not be able to detect.

4.2 Sample Vulnerable Contracts

Before we begin an analysis of all the contracts on the Ethereum blockchain, it is helpful to write some sample contracts that demonstrate the vulnerabilities that Maian was designed to catch.

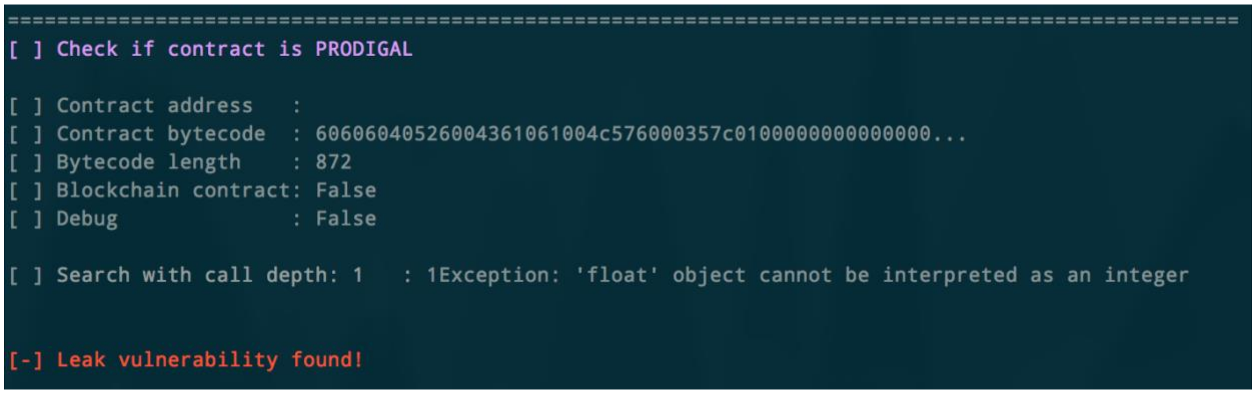
Listing 6: A prodigal bank contract

```
pragma solidity ^0.4.20;

contract ProdigalBank {
    address owner;
    uint256 balance;

    function withdraw(uint256 amount) public {
        if (msg.sender == owner && amount <= balance) {
            balance -= amount;
            pay(msg.sender, amount);
        }
    }

    function pay(address recipient, uint256 amount) public {
        recipient.transfer(amount);
    }

    function () public payable {
        balance += msg.value;
    }
}
```

The first vulnerability we chose to model was the prodigal account. We wrote a very simple bank contract which has one owner and an amount of Ether associated with that owner's address, shown in *Listing 6*. The owner can interact with this account by sending a transaction to the account, which will invoke the fallback function. This function adds the amount of Ether to the owner's balance. The owner can withdraw from the account by calling the withdraw function will a specified amount that will be subtracted from the balance and sent to the owner using the contract's pay function.

The prodigal vulnerability lies within the pay function. Because this function is public, it can be called by any account on the Ethereum network. Therefore, any account can list themselves, or any arbitrary non-owner address as the recipient for the bank's funds.

When we ran the bytecode of this contract through Maian, we found what we expected: the contract was flagged for a prodigal vulnerability, and not for a suicidal or greedy vulnerability. Sample output for the non-GUI implementation of Maian is provided in *Figure 2*. In particular, *Figure 2* depicts the results of the prodigal bank contract. The output for the greedy and suicidal vulnerability checks look very similar.



```
==========================================================================================
[ ] Check if contract is PRODIGAL

[ ] Contract address   :
[ ] Contract bytecode  : 60606040526004361061004c576000357c0100000000000000...
[ ] Bytecode length    : 872
[ ] Blockchain contract: False
[ ] Debug              : False

[ ] Search with call depth: 1   : 1Exception: 'float' object cannot be interpreted as an integer


[-] Leak vulnerability found!
```

Figure 2: Sample Maian results

We next wrote a simple greedy bank contract, shown in *Listing 7*. This contract is very similar to the prodigal bank contract. The owner (in fact, anyone) can deposit funds by sending Ether to the contract via a transaction, which is added to the balance in the fallback function. However, the contract has no method for the owner of the bank account to withdraw funds that it has deposited. This, of course, is what makes the contract greedy. Although there is no prodigal vulnerability which can leak Ether to arbitrary accounts, without any functionality to send the balance of the contract to another address, either via a call or self-destruct instruction, the account simply accumulates Ether with no way to use it.

Listing 7: A greedy bank contract

```
pragma solidity ^0.4.20;

contract GreedyBank {
      address owner;
      uint256 balance;

      function () public payable {
            balance += msg.value;
      }
}
```

When running Maian's vulnerability checks on this contract, we again observed the results we were expecting. A flag was raised for a greedy vulnerability, but not for prodigal or suicidal vulnerabilities.

Finally, we included a banking smart contract that demonstrates the suicidal behavior. It is beneficial for many contracts to include some functionality that effectively deletes the contract and relinquishes its funds to a hopefully explicit address.

In *Listing 8*, we wrote code for another banking contract, very similar to the prodigal banking contract. It accumulates the owner's balance the same way the previous two contracts have: the owner sends a transaction with the deposit amount, which invokes the fallback function to add that amount to the balance. However, it differs from the previous two in that when the owner wants to withdraw from its account, it kills the bank contract through a self-destruct call that sends all of its Ether holdings to the owner.

The exploitable behavior of this contract should be obvious: as in the case of the prodigal contract, a public function can be taken advantage of to leak Ether. In this case, the empty function can be given any address that will be the benefactor of the self-destruct instruction.

Listing 8: A suicidal bank contract

```
pragma solidity ^0.4.20;

contract SuicidalBank {
     address owner;
     uint256 balance;

     function withdrawal() public {
          if (msg.sender == owner && balance > 0) {
               empty(msg.sender);
          }
     }

     function empty(address recipient) public {
          self-destruct(recipient);
     }

     function () public payable {
          balance += msg.value;
     }
}
```

When we tested this contract using Maian, we found that it was flagged for not only a suicidal vulnerability but a prodigal vulnerability as well. This is not what we were initially expecting, but makes sense. Because the definition of a prodigal contract is one that can leak Ether to an arbitrary contract, any suicidal contract that sets an arbitrary address as the beneficiary of the killed contract's funds can be considered prodigal as well. Since the contract in *Listing 8* does so, it should be flagged as both suicidal and prodigal.

However, a contract can be suicidal but not prodigal. When we changed the empty function in our suicidal bank contract to not accept any parameters, and simply self-destruct to the owner of the account, Maian only flagged the contract as suicidal. This contract is still potentially dangerous because if it can be killed by anyone, contracts that rely on it might lock up or leak Ether when it is killed.

Though the three contracts we discuss in this section are very simple, they are still important to understand as they serve as a control for the rest of the contracts that are actually on the Ethereum blockchain. In the next section, we describe how we collect data on those contracts.

4.3 Methodology

In order to gather statistics on the smart contracts currently deployed on the Ethereum blockchain, we used Parity, a lightweight Ethereum client which allows users to create Ethereum accounts, perform transactions, and interact with the Ethereum blockchain [9]. More specifically, we used this tool in order to download the entire blockchain locally.

Once we were connected with the blockchain, we developed a Go language script that would run through the transactions of every block and collect information on the smart contracts that were created and run by those transactions. This included the Ether balance, the number of times each contract was called, and the bytecode instructions used. These were recorded by mapping each address of a unique contract with a balance, reference count, instruction count, and vulnerabilities found by Maian. We were able to do so in a straightforward manner by deciding whether the transaction we were analyzing was a creation transaction or a message transaction.

The creation transactions create new smart contracts. Included in these transactions are the contract's bytecode. This can be considered the latest version of the contract's code because it cannot be changed after creation. When our script encountered a creation transaction, it recorded the most recent balance of that account. Then, it ran the account's code through Maian's security tool to find out whether Maian flagged that contract for prodigal, suicidal, or greedy vulnerabilities. Finally, we counted all the instructions that made up the bytecode.

The only information we gathered from message transactions were the number of calls to each contract. This was done simply by tallying the reference map each time a contract appeared in the "to" field of one of these transactions.

After we have finished parsing through every block's transactions, we store each mapping to a CSV file. This includes the contract's hash, balance, reference count, and Maian results. We decided only to include the instruction counts for CALL, CALLCODE, and DELEGATECALL because these are the instructions we are primarily interested in. We wanted to use these instructions to determine how interdependent the contracts are, as they are the instructions contracts used to communicate with other contracts.

It is important to mention that additional functionality was implemented to run this script in parallel, analyzing several blocks at a time. This was done mostly because of the latency each Maian test posed. Because Maian takes on average 10 seconds to analyze each contract, and we were expecting to have to test nearly one million contracts, this script could have taken about two and a half weeks to run sequentially.

Furthermore, when we first tried to run Maian in parallel using 32 different threads, we found that the vulnerability checks slowed down significantly. Therefore, we implemented a timeout of one minute for each test. When we first ran Maian on the first 65,535 unique smart contracts on the Ethereum blockchain with this many threads and using this timeout, the timeout was killing a significant portion of our tests and rendered our data inconclusive.

To address this problem, we decided to only run the Maian tests for smart contracts with nonzero balances of Ether, which would significantly decrease the runtime of our collection. Because our primary interest is how effective these contract security tools are against contracts with substantial amounts of Ether and contracts that are traded with frequently, we believe this is a reasonable sacrifice to make.

4.4 Results

We began the last of our tests at approximately 19:30 EST on April 26, 2018. This test ended around 8:30 EST the next day. Our tool documented statistics for the first 970,892 smart contracts on the first 4,799,998 blocks of the Ethereum blockchain. This is the same population of smart contracts on which Maian's founders evaluated their tool.

The difference in our run is that we only ran Maian on contracts that had a non-zero balance of Ether. We found 98,720 of these smart contracts. However, we were able to map the Maian results of these contracts to all of the duplicates of that contract, which may have had zero balances. This being said, we obtained Maian results for 574,227 smart contracts.

Table 1: Vulnerabilities by Contract

Vulnerability	**Contracts**	**Percentage (%)**
No vulnerability	573125	99.808
Greedy	1007	.17537
Prodigal	48	.0083591
Suicidal	30	.0052244
Suicidal & Prodigal	17	.0029605

Our findings are depicted in *Table 1*. It is important to note that seventeen contracts, similar to the suicidal bank contract in *Listing 8* in Section 4.2, tested positive for both suicidal and prodigal vulnerabilities. However, there were no contracts that tested positive for greedy and suicidal, greedy and prodigal, or all three vulnerabilities.

The results in *Table 1* alone are enough to confirm our suspicions. However, we wanted further information on what contracts exactly are being diagnosed as vulnerable. Specifically, how wealthy are the contracts that are being flagged as prodigal, suicidal, or greedy?

As stated before, of the contracts we analyzed, 98,720 had nonzero balances of Ether. That leaves 872,172 contracts without Ether. However, the average balance per contract is still about 10.5 Ether. This means that the distribution of wealth among Ethereum smart contracts is completely erratic. It is because of this that we expect to see that most of the vulnerabilities Maian was documented to find were in contracts with a zero balance.

In *Table 2,* we look at the percentage of vulnerabilities for all the Ether circulating on the portion of the Ethereum blockchain that we sampled. In total, the smart contracts we analyzed held 4,487,770 Ether, which equates to just under $3.1 billion. The Ether to USD conversion (1 ETH = $687.39) was taken at 15:00 EST on April 28, 2018 [11].

Table 2: Vulnerability by Contract Balance

Vulnerability	**Ether**	**US Dollar ($)**	**Percentage (%)**
No vulnerability	4,283,690	2,944,565,669	95.453
Greedy	204,054	140,264,679	4.5469
Prodigal	7.8268	5,380	.00017440
Suicidal	12.6959	8,727	.00028290
Suicidal & Prodigal	7.92075×10^{-13}	0	0

These results serve as less confirmation than the results from before. Contracts that have been flagged as greedy make up 4.5% of total contract wealth. In particular, Maian indicated that one contract with 203,468 Ether was greedy. If this is true, that contract will have locked away almost $140 million worth of Ether. This is a significant find that is similar to the Parity bug mentioned earlier.

Next, we wanted to analyze the results of Maian against how many times each contract is being used. This is important because the more the Ethereum network is completing transactions with a smart contract is a good measure of how important it is. Because this is a new realm of computer science and e-commerce and has received quite a bit of attention just in the past year, it is fair to assume that several of these smart contracts are experimental and are not interacted with very often. Maian might flag these contracts as vulnerable, but that is not entirely relevant, as they are not going to be sent much Ether in the future.

Our smart contracts had a similar erratic distribution of contract references as it did for contract balances. More than half of these contracts (558,206) were never sent a transaction and thus had no references. However, the average was 44.4 references per contract, while the standard deviation was a whopping 7551.8. Like with balances, we expect that this dispersion could mean that many vulnerabilities Maian originally claimed to have found are with contracts that are not used very often.

Table 3: Vulnerability by Contract Reference

Vulnerability	**Contract References**	**Percentage (%)**
No vulnerability	13,746,333	99.875
Greedy	15,206	.11048
Prodigal	1,724	.012526
Suicidal	114	.00082828
Suicidal & Prodigal	30	.00021797

The results we gathered in *Table 3* reflect similar statistics that we found in *Table 1*. The contracts that Maian deems vulnerable by being greedy are referred to much more than any other vulnerability and even these contract references make up a fraction of a percent of the total contract interactions.

Finally, we wanted to acquire some measure of how much the contracts we tested for vulnerabilities interact with each other. We did this by counting the presence of the three opcodes Ethereum smart contracts use to call other contracts: DELEGATECALL, CALLCODE, and CALL. The results of this are shown in *Table 4*. Though CALL is present in almost ninety percent of our contracts, the other two instructions are used far less.

Table 4: Opcode Count

Opcode	**Total Count**	**Contract Count**	**Percentage (%)**
CALL	1,096,179	514,793	89.650%
DELEGATECALL	12,859	8,309	1.4470%
CALLCODE	3,958	3,378	.58827%

4.5 Discussion

After looking at Maian-found vulnerabilities only in contracts with Ether holdings, it is fair to say that the security tool is far less effective on nontrivial contracts than trivial ones.

When Maian was first engineered, it claimed to find 1,504 prodigal contracts, 1,495 suicidal contracts, and 31,201 greedy contracts on the Ethereum blockchain. These contracts, similar to our results, take into account *all* accounts; that is, these figures include duplicate contracts. However, when we ran Maian on just the contracts that had a non-zero balance of Ether, we only found 48 prodigal, 30 suicidal, and 1,007 greedy vulnerabilities. Considering our Maian tests took up 59.14% of the contracts, yet we found less than *one-thirtieth* of the vulnerabilities that were originally found, it is safe to conclude that Maian is far more successful in diagnosing bugs in contracts without Ether.

The same kinds of results were echoed when we counted the amount of times contracts flagged for vulnerabilities were referenced by other contracts. Though the accounts we tested were, in total, sent transactions almost 14 million times, flagged contracts were only referenced about 17,000 times.

Nevertheless, Maian is not incapable of flagging contracts with Ether. In fact, when we look at the amount of Ether controlled by the contracts Maian did flag, we found that a significant amount of the blockchain’s wealth (4.5%) was held by greedy contracts. This is alarming and runs counter to our original hypothesis. The rest of the vulnerable contracts only made up about $14,107 in Ether.

Nonetheless, we found one account in particular flagged as greedy that accounted for 99.7% of the cryptocurrency in this subset of smart contracts. When we took a look further into the greedy contract that, according to our Maian results, had effectively lock up almost $140 million worth of Ether, we were interested in what we found. Using Etherscan, an online database which stores information about these smart contracts, including Solidity source code for some of these smart contracts, we were able to look at what this contract was actually doing [12]. We have provided the source code for this contract in Appendix B.

Essentially, this contract, called Fundraiser, is a fundraising smart contract with two “signers” that together decide when to withdraw, how much to withdraw, and which account to withdraw to. Anyone can send Ether to the contract, which is handled in the payable fallback function. However, when the signers want to withdraw from the contract account, both signers must send a transaction that is converted into a proposal by the Withdraw function. If both proposals match up, the Ether amount is sent to the agreed upon address in the MaybePerformWithdraw function.

The problem, as Maian sees it, is that the *transfer* call at the end of the MaybePerformWithdraw function, which happens to be the only way for this contract to send Ether to another account, is not reachable by a single transaction. Therefore, the contract, by Maian’s definition, is greedy, because it can collect Ether but has no way to send it to other Ethereum accounts.

However, we know this not to be the case. The transfer call at the end of MaybePerformWithdraw is reachable, but it takes two transactions from the two signers to reach it. It is fair to say, then, that the flag Maian raised for this contract is a false positive, and thus can be scratched from our results posted in Table 2. When we do this, we find that contracts that are defined as greedy by Maian make up only about $402,650 worth of Ether, which makes up .013672% of all the Ether held by contracts we tested by Maian. It is fair to assume that if further analysis were done on the contracts that tested positive for greedy vulnerabilities, this percentage would continue to drop.

We were surprised to see so few uses of DELEGATECALL and CALLCODE. It appears library contracts are used far less often than we originally thought. This could be due to the fact that we are only counting instructions in contracts with an Ether balance. Often a library contract is used for transactions of tokens that are not Ether. It’s possible then that the purpose of contracts that interact with library contracts is solely to trade tokens, and not to trade Ether.

Nevertheless, we found that CALL was present in almost ninety percent of contracts with Ether. This is not surprising, as if these contracts have Ether, and do not have some way to send or transfer this Ether to other accounts, then they would have greedy vulnerabilities. This finding proves our theory that these contracts are interactive with one another. It is our opinion, then, that these security tools must treat them as such.

CHAPTER FIVE

Closing Remarks

5.1 Future Work

Though existing tools have proved effective on attacks that have already occurred, such as the theDAO contract leak and the Parity contract bug, it is only a matter of time before another theft of millions of US dollars worth of Ether leads to the development of tools which can detect similar exploits. This is not to say that the future of contract security should not keep past exploits in mind; rather, a comprehensive understanding of how contracts inter-communicate with one another as well as which instructions are being used to do so is necessary to define what it means to safely call another contract.

An interesting challenge for the future design of smart contract security tools will be to have a novel way to keep track of a contract's state through not just one transaction, but multiple. The example of the Fundraiser smart contract, which requires two transactions from both the signer addresses declared in the constructor, triggered a false positive from Maian. This happened because the transfer of funds could never be reached with only one transaction, but rather needed two before that statement was reachable.

Another challenge for these future tools will be to keep track of not just the world state of the Ethereum blockchain and the state of the contract under scrutiny, but also the state of any contract the original might call using a DELEGATECALL, CALLCODE, or CALL. Only then would these tools be able to properly model these calls to other contracts and truly analyze how a smart contract’s inter-contract behavior is potentially susceptible an Ether leak or lock.

These challenges, of course, will need to be met with runtime and memory usage flexibility. Being able to symbolically analyze the instructions of a stateful smart contract is non-trivial enough, but additionally tracking the states of other contracts through not one but multiple state-changing transactions could exponentially increase analysis time.

It is likely then that if a tool like this were to be created, that its purpose would not be to analyze the entire blockchain of smart contracts but perform analysis one contract at a time. In fact, the tool would best be used if it were provided to contract authors before they deploy their bytecode to the blockchain.

5.2 Conclusion

In this paper, we gathered data on the current ecosystem of the smart contracts in Ethereum. We hoped these statistics would shed light on the characteristics these contracts have so to suggest ways in which tools such as Oyente and Maian can be improved. We found that many of these contracts are not used and have no holdings of Ether. These contracts are much easier to find vulnerabilities in, as they are typically simpler, but this is not particularly helpful, as they cannot leak or lock up any Ether.

When the current tools are used against contracts that have Ether and are referenced multiple times, they catch far fewer vulnerabilities than before, and flags they do raise can very likely be false positives. Because these are the contracts that can cause the problems that we have seen before, there is clearly room for improvement.

As consumers continue to place their faith in cryptocurrencies, platforms such as Ethereum will have control over more wealth than ever before. Smart contracts will become more complex as will their interactions with each other. Unfortunately, these trends will only bring about larger economic incentives for malicious parties to target exploits in contracts that may have never been seen before. It is precisely because of this that contract authors and analyzers must continue to look for ways contracts can be misused before these malicious parties are given the chance.

Appendix A: Ethereum Bytecode Instruction Set [3]

Table 5: Stop and Arithmetic Operations

Value	String Representation	Description
0x00	STOP	Halts execution
0x01	ADD	Addition operation
0x02	MUL	Multiplication operation
0x03	SUB	Subtraction operation
0x04	DIV	Division operation
0x05	SDIV	Signed integer division operation
0x06	MOD	Modulo remainder operation
0x07	SMOD	Signed modulo remainder operation
0x08	ADDMOD	Modulo addition operation
0x09	MULMOD	Modulo multiplication operation
0x0A	EXP	Exponential operation
0x0B	SIGNEXTEND	Extend length of two’s complement signed integer

Table 6: Comparison & Bitwise Logic Operations

0x10	LT	Less-than comparison
0x11	GT	Greater-than comparison
0x12	SLT	Signed less-than comparison
0x13	SGT	Signed greater-than comparison
0x14	EQ	Equality comparison
0x15	ISZERO	Simple not operator
0x16	AND	Bitwise AND operation
0x17	OR	Bitwise OR operation

0x18	XOR	Bitwise XOR operation
0x19	NOT	Bitwise NOT operation
0x1A	BYTE	Retrieve single byte from word

Table 7: Hash Operations

0x20	SHA3	Compute Keccak-256 hash

Table 8: Environmental Information Operations

0x30	ADDRESS	Get address of currently executing account
0x31	BALANCE	Get balance of the given account
0x32	ORIGIN	Get execution origination
0x33	CALLER	Get caller address
0x34	CALLVALUE	Get deposited value by the instruction/transaction responsible for this execution
0x35	CALLDATALOAD	Get input data of current environment
0x36	CALLDATASIZE	Get size of input data in current environment
0x37	CALLDATACOPY	Copy input data in current environment to memory
0x38	CODESIZE	Get size of code running in current environment
0x39	CODECOPY	Copy code running in current environment to memory
0x3A	GASPRICE	Get price of gas in current environment
0x3B	EXTCODESIZE	Get size of an account’s code
0x3C	EXTCODECOPY	Copy an account’s code to memory
0x3D	RETURNDATASIZE	Get size of output data from the previous call from the current environment
0x3E	RETURNDATACOPY	Copy output data from the previous call to memory

Table 9: Block Information Operations

0x40	BLOCKHASH	Get the hash of one of the 256 most recent complete blocks

0x41	COINBASE	Get the block’s beneficiary address
0x42	TIMESTAMP	Get the block’s timestamp
0x43	NUMBER	Get the block’s number
0x44	DIFFICULTY	Get the block’s difficulty
0x45	GASLIMIT	Get the block’s gas limit

Table 10: Stack, Memory, Storage, and Flow Operations

0x50	POP	Remove item from stack
0x51	MLOAD	Load word from memory
0x52	MSTORE	Save word to memory
0x53	MSTORE8	Save byte to memory
0x54	SLOAD	Load word from storage
0x55	SSTORE	Store word to storage
0x56	JUMP	Alter the program counter
0x57	JUMPI	Conditionally alter the program counter
0x58	PC	Get the value of the program counter
0x59	MSIZE	Get the size of active memory in bytes
0x5A	GAS	Get the amount of available gas, including the corresponding reduction for the cost of this instruction
0x5B	JUMPDEST	Mark a valid destination for jumps

Table 11: Push Operations

0x60	PUSH1	Place 1-byte item on stack
0x61	PUSH2	Place 2-byte item on stack
…	…	…
0x7f	PUSH32	Place 32-byte item (one full word) item on stack

Table 12: Duplication Operations

0x80	DUP1	Duplicate first stack item

0x81	DUP2	Duplicate second stack item
…	…	…
0x8f	DUP16	Duplicate sixteenth stack item

Table 13: Exchange Operations

0x90	SWAP1	Exchange first and second stack item
0x91	SWAP2	Exchange first and third stack item
…	…	…
0x9f	SWAP16	Exchange first and seventeenth stack item

Table 14: Logging Operations

0xA0	LOG0	Append log record with no topics
0xA1	LOG1	Append log record with one topic
…	…	…
0xA4	LOG4	Append log record with four topics

Table 15: System Operations

0xF0	CREATE	Create a new account with associated code
0xF1	CALL	Message-call into an account
0xF2	CALLCODE	Message-call into this account with an alternative account's code
0xF3	RETURN	Halt execution returning output data
0xF4	DELEGATECALL	Message-call into this account with an alternative account's code, but persisting the current values for sender and value
0xFE	INVALID	Designated invalid instruction
0xFF	SELFDESTRUCT	Halt execution and register account for later deletion

Appendix B: Fundraiser Contract Solidity Code [12]

```
pragma solidity 0.4.11;

contract Fundraiser {

        /* State */

        address public signer1;
        address public signer2;

        enum Action {
                None,
                Withdraw
        }

        struct Proposal {
                Action action;
                address destination;
                uint256 amount;
        }

        Proposal public signer1_proposal;
        Proposal public signer2_proposal;

        /* Constructor, choose signers. Those cannot be changed */
        function Fundraiser(address init_signer1,
                                address init_signer2) {
                signer1 = init_signer1;
                signer2 = init_signer2;
                signer1_proposal.action = Action.None;
                signer2_proposal.action = Action.None;
        }

        /* allow simple send transactions */
        function () payable {
        }


        /* Entry points for signers */

        function Withdraw(address proposed_destination,
                                uint256 proposed_amount) {
                /* check amount */
                if (proposed_amount > this.balance) { throw; }
                /* update action */
                if (msg.sender == signer1) {
                        signer1_proposal.action = Action.Withdraw;
                        signer1_proposal.destination = proposed_destination;
                        signer1_proposal.amount = proposed_amount;
                } else if (msg.sender == signer2) {
                        signer2_proposal.action = Action.Withdraw;
                        signer2_proposal.destination = proposed_destination;
                        signer2_proposal.amount = proposed_amount;
                } else { throw; }
                /* perform action */
                MaybePerformWithdraw();
        }

        function MaybePerformWithdraw() internal {
                if (signer1_proposal.action == Action.Withdraw
                        && signer2_proposal.action == Action.Withdraw
                        && signer1_proposal.amount == signer2_proposal.amount
                        && signer1_proposal.destination == signer2_proposal.destination) {
                        signer1_proposal.action = Action.None;
                        signer2_proposal.action = Action.None;
                        signer1_proposal.destination.transfer(signer1_proposal.amount);
                }
        }

}
```

BIBLIOGRAPHY

[1] Nakamoto, Satoshi. “Bitcoin: A Peer-to-Peer Electronic Cash System.” *Bitcoin.org*, Jan. 2009, bitcoin.org/bitcoin.pdf.

[2] Buterin, Vitalik. “Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform.” *Ethereum.org*, Ethereum, 2014, www.ethereum.org/.

[3] Wood, Gavin. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Ethereum, 12 Apr. 2017, www.cryptopapers.net/papers/ethereum-yellowpaper.pdf.

[4] Atzei N., Bartoletti M., Cimoli T. (2017) A Survey of Attacks on Ethereum Smart Contracts (SoK). In: Maffei M., Ryan M. (eds) Principles of Security and Trust. POST 2017. Lecture Notes in Computer Science, vol 10204. Springer, Berlin, Heidelberg

[5] Luu, Loi, et al. “Making Smart Contracts Smarter.” *National University of Singapore Computing*, National University of Singapore, 2016, www.comp.nus.edu.sg/~loiluu/papers/oyente.pdf.

[6] Suiche, Matt. “Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode.” *Comae.io*, Comae Technologies, 7 July 2017, www.comae.io/reports/dc25-msuiche-Porosity-Decompiling-Ethereum-Smart-Contracts-wp.pdf.

[7] Dika, Ardit. "Ethereum Smart Contracts: Security Vulnerabilities and Security Tools." Norwegian University of Science and Technology, Dec. 2017.

[8] Nikolic, Ivica, et al. "Finding the Greedy, Prodigal, and Suicidal Contracts at Scale." *Finding the Greedy, Prodigal, and Suicidal Contracts at Scale*, ArXiv, 14 Mar. 2018.

[9] Parity Ethereum Client [Computer Software]. (2018). Parity Technologies. Retrieved from https://www.parity.io/.

[10] Higgins, Stan. "From $900 to $20,000: Bitcoin's Historic 2017 Price Run Revisited." *CoinDesk*, 30 Dec. 2017, www.coindesk.com/900-20000-bitcoins-historic-2017-price-run-revisited/.

[11] Ethereum Price. (n.d.). Retrieved April 28, 2018, from https://ethereumprice.org/

[12] Etherscan: The Ethereum Block Explorer. (n.d.). Retrieved April 29, 2018, from https://etherscan.io/

ACADEMIC VITA

Luke Gwaltney
lrg5180@psu.edu

EDUCATION

PENNSYLVANIA STATE UNIVERSITY'S SCHREYER HONORS COLLEGE *University Park, PA*
B.S. in Computer Engineering, Minor in Spanish *2014-2018*

- Awards and Distinctions: Dean's List, William A. Schreyer Scholarship
- Thesis: *Analysis of Safe Smart Contract Calling in the Ethereum Virtual Machine*
- Supervisor: Dr. Patrick McDaniel
- Advisor: Dr. John Hannan

WORK EXPERIENCE

HONEYWELL VOCOLLECT *Pittsburgh, PA*
Software Engineering Intern *May-August 2016 & 2017*

- Worked two summer internships helping develop a voice recognition mobile application designed to enhance retail store inventory management efficiency
- Completed projects including, but not limited to:
  - various cleanups of the user interface of mobile application, including internationalization of entire app and addition of confirmation screens
  - unit tests for settings portion of the app
  - continuous integration strategies for building pre-release versions of the app
- Assisted in team-wide regression testing prior to the 2.1 release of the app
- Participated in design meetings and daily standup alongside team of software engineers

COMMUNITY INVOLVEMENT

PENN STATE THON *University Park, PA*
Dancer Relations Committee Member *2016 & 2017*

- Participated in year-long awareness and fundraising efforts to support those impacted by childhood cancer, culminating in an annual dance marathon in February
- Partnered with a participant in the dance marathon to help keep them on their feet for all 46 hours

SKILLS

- Experience with C++, C#, C, Java, Python, and Go programming languages
- Experience with SourceTree, Git, and other version control systems
- Understanding of Microsoft, Macintosh, and Ubuntu operating systems
- Proficiency in Spanish Language (minor), adept in basic conversation and grammar