Project Summary

Project Title: Software Assurance for Blockchain Contracts

Document: Final Report

Date: May 4th, 2018

Investigators:

Ryan Shivers
Student Researcher
Tennessee Technological University
Rmshivers42@students.tntech.edu

Zachary Wallace
Student Researcher
Tennessee Technological University
Zawallace42@students.tntech.edu

Samuel Wehunt
Student Researcher
Tennessee Technological University
Swwehunt42@students.tntech.edu

Dr. Mohammad Ashiqur Rahman
Faculty Advisor
Tennessee Technological University
marahman@tntech.edu

Dr. Paul E. Black
Technical Director / Project Sponsor
National Institute of Standards and Technology
Paul.black@nist.gov

Problem Title: Software Assurance for Blockchain Contracts
Area Description: This project is investigating the area of blockchain smart contracts. Blockchain technology is used to create distributed, tamper resistant ledgers for use in finances, business transactions, and many other peer-to-peer applications. Smart contracts are a way for multiple parties to transact based on the application logic of a contract stored on a blockchain.

Keywords: Blockchain, Software Assurance, Ethereum, Smart Contract, Static Analysis
Project Description: Our team is interested in methods of ensuring the security of blockchain smart contracts. Our team is investigating tools that already to exist to solve this issue. So far, several promising tools have been discovered; however, due to licensing and closed-source code it has been difficult to perform an in-depth analysis of some of these tools.

Executive Summary

Blockchain technology is rapidly developing and, with proper execution, is on its way to widespread adoption in many areas outside of the technology sphere. Blockchains have many use cases and one function that can be provided is the idea of “smart contracts”. Smart contracts are applications that are stored on the blockchain that allow trustless transactions between peers [1]. This is possible through smart contracts because once a contract is written to the blockchain it is immutable and will execute according to a consensus protocol between miners. Smart contracts are executed when a transaction on the blockchain meets specific requirements, and then the contract is executed by all miners on the network so that a consensus on the result is achieved; this makes it so that the result of a smart contract cannot be manipulated by any single party. Blockchains are transparent to all interacting nodes so this entire process is visible to all involved parties from beginning to end.

Smart contracts seem like a very good application within blockchain technology; however, there are risks involved with utilizing these contracts because they can be developed and deployed by anyone with a working knowledge of common programming languages. This leads to issues because once these contracts are pushed to the blockchain it is impossible to change them without a hard fork. A hard fork in a blockchain is when 51% or more nodes agree to fundamentally change the blockchain design and update their software to a new version. When this happens all nodes that do not cooperate are left mining the old version of the blockchain that is incompatible with the new version. The blockchain community is staunchly against utilizing hard forks arbitrarily as it splits the community and goes against one of the core values of this technology: immutability. This means that vulnerable smart contracts must remain on the blockchain indefinitely, where they are open to exploitation by malicious parties. One suggestion to mitigate this issue is to have a clause that invalidates old contracts if a signed update is pushed. This solution is workable, but this project’s goal is to prevent this problem entirely by developing a testing suite that will allow smart contract developers to assure that their software is as error-free as possible before deploying it to the blockchain.

Project Title: Software Assurance for Blockchain Contracts
Date: May 4th, 2018
Authors and Affiliations:

1. Ryan Shivers – Tennessee Tech University
2. Zachary Wallace – Tennessee Tech University
3. Samuel Wehunt – Tennessee Tech University
4. Dr. Ashiqur Rahman – Tennessee Tech University
5. Dr. Paul E. Black – National Institute of Standards and Technology (NIST)

Table of Contents

Project Summary 1
Executive Summary 2
1. Introduction 4
1.1. Problem Statement 4
1.2. Purpose Statement 4
1.3. Motivation 4
2. Literature Review 5
2.1. Bugs and Vulnerabilities 5
2.2. Smart Contract Analysis Tools 6
2.3. Motivation 7
2.4. Alternate Views 8
2.5. Justification 8
3. Methods and Procedures 8
3.1. Method Characterization 8
3.2. Plan Overview 9
3.3. Schedule 9
3.4. Deliverables 10
3.5. Limitations and Delimitations 10
4. Findings 11
4.1. Overview 11
4.2. Detailed Findings 11
Analysis Discrepancies between Tools 13
5. Issues 17
6. Conclusions and Recommendations 17
6.1. Interpretation of Findings 17
6.2. Future Work 18
7. References 18
8. Team 19
8.1. Biographical Sketches 19
8.2. Tasking 20
9. Appendix 20

1. Introduction

1.1. Problem Statement

Blockchains are a very useful tool, and smart contracts make blockchain technology even more powerful. With this utility come some pretty big risks as well, as seen with the DAO hack which caused an estimated $60 million in losses. This incident was caused by a vulnerability in a smart contract which allowed attackers to move an arbitrary amount of money. This vulnerability would have been caught early if the developers had good software assurance practices. The goal of this project is to develop a solution which allows blockchain developers to prevent simple vulnerabilities such as these from ever entering the blockchain.

1.2. Purpose Statement

The purpose of this project is to research existing tools and techniques which can detect vulnerable smart contract code. This research will then be used to develop a framework which blockchain developers can follow which will allow them to develop provably secure smart contracts. Currently there are tools that exist which can perform automatic auditing of source code, but as evidenced by the DAO hack, these tools are not being integrated into the development cycle of production code. It is our goal to make integrating these automated tools into a development workflow easy and painless for blockchain developers, so that the number of bugs introduced into the blockchain is greatly reduced.

1.3. Motivation

Each researcher on our team sought this project out because we have a strong background in software development and an interest in blockchain technologies. We want to see blockchain technology thrive and smart contracts offer a function that could have widespread use in society. Contractual agreements have existed for millennia and will always have a place in society. Removing the need for a trusted third party opens up new avenues for businesses to conduct business without the need to worry about their transactions and with much cheaper fees than a professional mediator. Smart contracts could even lead to a restructuring of basic organizational hierarchies as seen attempted by the Decentralized Autonomous Organization (DAO).

The concept of The DAO was to create a framework for leaderless, true democratic organizations. All members of The DAO were in possession of coins that they could use to vote on propositions made by other members of the organization. All of the functions in this organization were designed as smart contracts which users contacted by sending certain amounts of Ether (the internal currency of the Ethereum blockchain) to a certain contract with parameters attached. The contract then executes its programmed instructions and performs one of many tasks such as starting a proposition or executing a trade [2]. This framework could have strong applications in the business world as it enacts true, transparent democracy. This would be great if contracts written for purposes of this sort had a way of

assuring that their contracts had no bugs / vulnerabilities. The DAO failed quickly after its inception due to a discrepancy of this nature. There was a vulnerability in one of the contracts that allowed a member to recursively execute a fork proposition and allowed the attacker to claim his portion of the DAO funds repeatedly which quickly drained the DAO of its finances and its support from the community. After this vulnerability was exploited, and over $55 million dollars was stolen, the members of the DAO pressured the Ethereum blockchain to perform a hard fork which would allow them to reclaim their stolen property [3]. The Ethereum community split over this decision because the whole concept of blockchains is that if a user makes a contract and follows the rules that have been put in place then the transaction is seen as valid. The technical details of the attack on the DAO will be discussed in a further section; however, this is a perfect example of why it is critically important that contracts be heavily scrutinized before they enter the blockchain ecosystem.

2. Literature Review

2.1. Bugs and Vulnerabilities

This section focuses on common vulnerabilities that we have discovered that are either platform agnostic or were applicable in an Ethereum environment and could also be applicable in a Hyperledger context. The first four bugs and vulnerabilities listed in this section are discussed in more detail in Appendix A.

2.1.1. Timestamp Dependence

A timestamp dependent contract is a contract that relies on the timestamp of some block in the blockchain to execute a critical operation, such as transferring Ether. The timestamp of a block can be changed by the miner of the block and can therefore be manipulated for some sort of malicious activity (i.e. stealing Ether) [8].

2.1.2. Mishandled Exceptions

Smart contracts have the ability to throw exceptions because of the language they are written in. Lots of times these exceptions are not explicitly checked for, resulting in the failure of a transference of Ether, roles of users, etc. [8]

2.1.3. Transaction Ordering Dependence

An important thing to consider when dealing with smart contracts is that there can be many users invoking transactions on the same contract. The order that these transactions are executed could play a vital role in the outcome of a contract. For example, updating the cost of an item at the same time another user is buying that item could result in the buyer paying much more for something than expected. [8]

2.1.4. Reentrancy Vulnerability

Reentrancy is a well-known vulnerability that the DAO attack exploited, as mentioned above. When one contract calls another, the called contract is halted in its execution until the call is

finished. This leaves the contract in an intermediate state that a malicious attacker could take advantage of [8]

2.1.5. Integer Overflow and Underflow

This is a very simple vulnerability that is equally simple to mitigate. In solidity (the programming language used to develop Ethereum contracts) if a value goes above the predefined limit it resets to zero and if a value goes below 0 it resets to the maximum value [9]. This attack is simple and easy to check for but if a developer misses this while allowing users to withdraw or send money via a contract then it could allow an attacker to withdraw past their limit, get the max allowable value, and then withdraw that amount from the contract.

2.1.6. Denial of Service

This is a broader vulnerability that is common in regular software. An example of a denial of service vulnerability is when an auction is taking place in a smart contract there is a leader that receives money from the highest bidder when a time limit is reached. If a refund is taking place and the refund fails then the system reverts. An attacker could take advantage of this system by becoming the leader and then setting their code up such that their refunds always fail which will effectively block more auctions from taking place [9].

2.2. Smart Contract Analysis Tools

This section details the analysis tools we have discovered and have been studying more in depth. Some of these tools are not open-source but these tools will be noted as such in their independent section.

2.2.1. Solcover

SolCover is a tool that utilizes the idea of Code Coverage to test the source code of a Solidity contract. Code Coverage is the process of determining how much of a program can actually run given a set of inputs that is designed to traverse every possible execution path in the code. The inputs are generated automatically and then tested to determine whether the result produced from running the program with said input was expected. SolCover simply takes this idea and applies it specifically to Solidity contracts, which by nature are different from regular software [10].

2.2.2. Manticore

Manticore is a symbolic execution tool made specifically for Ethereum Virtual Machine bytecode which is the decompiled representation of Solidity contracts. This program also generates inputs automatically but rather than try to cover all execution paths it focuses on generating unique code paths that could be maliciously used. Manticore also attempts to generate inputs which will result in memory violations that could be exploitable. Once an execution path has been generated by Manticore it is possible to trace through the program's execution with that specific input and see variable values at each step. Manticore also offers an extendable programmatic interface which can be used to create custom analysis processes which could prove very useful for the automation of our framework [11].

2.2.3. Mythril

Mythril is a disassembler / blockchain exploration and analysis tool that can be used to not only check a specific contract for vulnerabilities but can also parse the entire Ethereum blockchain and find contracts which likely have a vulnerability and mark them for further investigation. This is very useful because we can use it to find test contracts that are vulnerable to certain types of vulnerabilities. It does this scan by taking a snapshot of all of the contracts on the Ethereum mainnet and then analyzes function signatures and opcodes to search for contracts that are likely to contain a vulnerability [12].

2.2.4. Oyente

Oyente is another symbolic execution tool that looks for common vulnerabilities. Some of the vulnerabilities it claims to be able to find are the ones detailed in 2.1.1. – 2.1.4. Oyente is open-source and is made specifically for analyzing Ethereum Virtual Machine Bytecode. At one time a test was run against 19,366 contracts on the Ethereum mainnet and Oyente marked 8,833 of them as vulnerable. Oyente is designed specifically to have a lower rate of false positives when compared to other tools [13].

2.2.5. Solgraph

Solgraph is a static analysis tool that visualizes the control flow of a Solidity contract by generating a DOT graph. Potential vulnerabilities that arise from specific portions of the code taking control are highlighted in red to make it easier for someone analyzing the code to see the structural problem. This tool is very different than other tools that are listed in this section and cannot be easily automated; however, it could serve as a very useful tool in our comprehensive framework [14].

2.2.6. SmartCheck

There is not a lot of publicly available information about SmartCheck because it is not an open-source tool. It is an online Solidity auditing tool that claims to check for vulnerabilities and bad practices but it is difficult to verify because the process of doing this is obfuscated from the developer [15].

2.2.7. Securify

Securify is also a non-open-source project but there was more information about its inner workings available online. Securify is the opposite of Oyente in terms of false positives. Securify overestimates the behaviors of a contract so it can prove the non-existence of certain vulnerabilities within a program but tends to signal more false positives. Securify also attempts to traverse all paths in a program when checking for vulnerabilities so it has that in common with SolCover [16].

2.3. Motivation

The big event that triggered this area of research was the attack on the DAO that has been discussed in this paper. This was the first significant time where large amounts of money were stolen due to programmer negligence when developing the smart contracts used within a system. Since then different tools have been developed to attack this problem from different angles and the more research that has been done the more obvious it has become that this is a problem.

More vulnerabilities are being discovered constantly but the most dangerous vulnerabilities tend to be the simplest ones that are easy to overlook. Developing a system that makes it easier to spot careless bugs in the code make it less likely that once a contract is deployed that it will need to be reworked. If a contract is deployed with a vulnerability, the process of fixing the issue is very difficult and in the eyes of many blockchain proponents goes against the inherent nature of smart contracts and blockchains. That is, once something is deployed to the blockchain it should not be changed and transactions that follow the rules originally in place should be allowed to proceed. This means that it is very important to ensure that contracts are flawless upon deployment.

2.4. Alternate Views

We have been researching tools to secure contracts but there seems to be little to no effort in the way of combining tools to create one framework that developers can use to ensure the validity of their contracts. As detailed in Section 2.2. there are differences in each tool's specific implementation but there does not seem to be any discord regarding the need for these tools to ensure that contracts do not contain vulnerabilities when deployed.

2.5. Justification

The focus of our project is to aggregate all current information regarding common vulnerabilities and tools that can be used to avoid these vulnerabilities. As shown in 2.2 there are many tools available that use one approach to find issues in contracts but there are none that attempt to aggregate this information into one common framework that can be used to exhaustively search for vulnerabilities. This is even more obvious when observed in a Hyperledger context because there are essentially no tools available made specifically to analyze Hyperledger contracts.

Hyperledger is more geared to business, implementing private blockchains and supporting them with smart contracts for their internal transactions. Under this assumption it may seem unlikely for the need to check for vulnerabilities in the smart contracts because the users of this blockchain are all authorized and there is a communal trust between parties unlike the Ethereum blockchain. This does not excuse the lack of software assurance because if someone were to gain access to the private blockchain or the business is faced with an insider threat then they leave themselves open for attack. This is our goal is to make Hyperledger contracts as secure as possible because Hyperledger is more business orientated and could see widespread use in less-technology centered circles and they need to be protected with the same vigor as other mainstream business applications.

3. Methods and Procedures

3.1. Method Characterization

Our team used two methods for researching this topic. The first method utilized is exploratory research. Our goal was the explore the research space and find out what classes of vulnerabilities effect smart contracts specifically, and we also needed to research what tools currently exist to perform software

assurance analysis against smart contract code. After exploring what research currently exists, we then performed some empirical research by testing the discovered tools against smart contracts which contain the researched vulnerabilities.

3.2. Plan Overview

To accomplish the task ahead of us, we came up with a plan which breaks down the project into easily solved pieces. First, we researched what smart contract platforms exist and what programming languages they use. This gave us a good overview of the ecosystem we were working in. Once our team had a good idea of what platforms exist, we moved on to the second step of researching the vulnerabilities that are common to smart contract platforms. The third step of the process was to then research the tools that currently exist which can detect these vulnerabilities. Now that most of the preliminary research was done, our team moved on to step four which was the empirical analysis of the tools and vulnerabilities. We collected several smart contract code examples which were known to contain vulnerabilities, and some which contained no (known) vulnerabilities. We then tested these contracts against a suite of tools which claimed to be able to detect these vulnerabilities. The final step in this plan was to collect these results and analyze them to evaluate the effectiveness of the tools which were tested.

3.3. Schedule

	Time Period					
	2/14-2/28	3/1 - 3/15	3/16 - 3/30	3/31 - 4/13	4/14 - 4/28	4/29 - 5/11
Tasks						
1. Research Smart Contract Platforms						
1.1. Determine what programming languages are used						
1.2. Learn common issues with these languages						
1.3. Determine most used platforms						
2. Research vulnerabilities of all smart contract platforms						
2.1. Determine what vulnerabilities are common to all platforms						
2.2. Document how to catch these vulnerabilities						
3. Explore existing smart contract assurance tools						
3.1. Document all existing assurance tools						
3.2. Narrow down to most relevant tools						
4. Test smart contract assurance tools						
4.1. Collect known vulnerable contracts						
4.2. Test each tool against each contract						
4.3. Determine discrepencies between results						
4.4. Research why results differ						
5. Prepare final report and presentation						

Above is the Gantt chart that matches the exact tasks that were completed throughout the semester. This schedule has been altered since our previous iterations and the reasons for this will be discussed in the issues section below.

3.4. Deliverables

The deliverables for our project are listed below:
1. Research on Smart Contract Platforms

The purpose of this deliverable is to give an overview of what smart contract technologies currently exist and their usefulness.

2. Identification of general smart contract vulnerabilities
This gives detailed information on what vulnerabilities are specific to smart contract platforms and which are unrelated to other software.

3. Research on cutting edge smart contract assurance tools and frameworks

Some tools already exist to detect the vulnerabilities which plague smart contracts, and this is a report on which ones are out there.

4. Test results from vulnerable contracts

Our method of testing the tools involves running each one against smart contracts which are known to contain vulnerabilities, and then collecting those results in to one dataset.

5. Final analysis of tool effectiveness

This is the analysis of the above deliverable.

Each of these deliverables are contained in this report.

3.5. Limitations and Delimitations

There were several circumstances which limited our investigation into this research topic. One of the major hurdles came during the testing of vulnerable smart contracts. We found that contracts which were written for outdated versions of solidity (Ethereum's smart contract language) could not be analyzed by all of the tools. This means that many contracts were simply not able to be tested without rewriting them for the most up-to-date version. We also chose to delimit the scope of our testing to only tools which are open source. We chose to do this because proprietary software gives little value to the research community if the methods and techniques employed by the code cannot be known to the general public. While these closed-source tools may be very effective, it is more useful to know how these tools work at detecting vulnerabilities.

4. Findings

4.1. Overview

Testing and analyzing the three tools that have been chosen has led to some interesting discoveries about how the tools operate. Mythril is the most likely tool to raise flags when given a contract because it generates warnings for a lot of cases that could possibly cause undefined or unwanted behavior. It is also smart enough to know not to classify functions that utilize the transfer function as reentrancy vulnerabilities because this function limits the amount of gas that is provided to the external call which eliminates the possibility of a reentrancy attack. It still generates a warning about calling an external contract to let the tester know to be careful of the function but not telling him that it specifically has a vulnerability. Oyente is more likely to classify a function as a reentrancy vulnerability even if it utilizes the correct procedures. It searches for all commands that contact external contracts and marks them as transaction ordering dependencies because two transactions could possibly be within the same block and could alter the state before the next transaction executes. Manticore is much slower than the other two tools tested and performs a less comprehensive scan that is more prone to errors. This is explained in more depth in the following section.

4.2. Detailed Findings

We have collected 12 Solidity contracts that are known to have vulnerabilities in their code and tested the three static analysis tools against them to see which vulnerabilities are detected. Our three specified tools were installed on an Ubuntu VM and all of the tools were collected on this VM and then copied for each member of the team. In the following section we have listed each contract that contained a discrepancy between the results of testing the tools against the contract. If a tool produced results that were different from another tool's results we have listed it in the following section and described why the results were different and what we learned about each tool from this. The full results are shown below:

Mythril Test Results

	Type of Vulnerability			
Contract Name	Unprotected Function	Reentrancy	External call to untrusted contract	Integer overflow/underflow
Alice	FALSE	FALSE	FALSE	FALSE
Bob	FALSE	TRUE	TRUE	FALSE
RaceCondition	FALSE	TRUE	TRUE	FALSE
ReentrancyExploit	TRUE	TRUE	TRUE	FALSE
Reentrancy	TRUE	TRUE	TRUE	FALSE
Unprotected	FALSE	FALSE	FALSE	FALSE
GiftBox	FALSE	FALSE	TRUE	TRUE
Ownable (KOTH.sol)	FALSE	FALSE	FALSE	FALSE
CEOThrone (KOTH.sol)	FALSE	FALSE	FALSE	FALSE
Lottery	FALSE	FALSE	FALSE	TRUE
VarLoop	FALSE	FALSE	TRUE	FALSE
PrivateBank	FALSE	FALSE	TRUE	FALSE

Oyente Test Results

	Type of Vulnerability			
Contract Name	Code Coverage	Integer Underflow	Transaction Ordering Dependence	Reentrancy
Alice	99.3	FALSE	FALSE	FALSE
Bob	96.6	FALSE	FALSE	FALSE
RaceCondition	98.7	FALSE	FALSE	FALSE
ReentrancyExploit	97.8	FALSE	TRUE	TRUE
Reentrancy	97.7	FALSE	TRUE	TRUE
Unprotected	99.4	FALSE	FALSE	FALSE
GiftBox.sol	96.4	TRUE	TRUE	FALSE
Ownable (KOTH.sol)	98.4	FALSE	FALSE	FALSE
CEOThrone (KOTH.sol)	98.6	FALSE	FALSE	FALSE
Lottery.sol	86.2	TRUE	TRUE	FALSE
VarLoop.sol	96	FALSE	TRUE	FALSE
PrivateBank.sol	98.6	FALSE	FALSE	TRUE

Manticore Test Results

		Type of Vulnerability	
Contract Name	Coverage	Integer Overflow	Uninitialized Storage
Alice	100%	FALSE	FALSE
Bob	100%	FALSE	FALSE
RaceCondition	0% (Failed)	FAILED	FAILED
ReentrancyExploit	0% (Failed)	FAILED	FAILED
Reentrancy	72%	FALSE	TRUE
Unprotected.sol	100%	FALSE	FALSE
GiftBox.sol	44%	TRUE	FALSE
Ownable (KOTH.sol)	100%	FALSE	FALSE
CEOThrone (KOTH.sol)	88%	FALSE	FALSE
Lottery.sol	31%	FALSE	FALSE
VarLoop.sol	35%	FALSE	FALSE
PrivateBank.sol	14%	TRUE	FALSE

4.2.1. Analysis Discrepancies between Tools

This section describes each contract that generated results from each tool that differed from one another and described what caused the discrepancy and what we learned from it.

Contract – Bob

Mythril found a reentrancy vulnerability and an external call to an untrusted contract. Manticore and Oyente did not find any vulnerabilities in the contract.

Bob.sol declares two contracts within itself each which have the same two functions (set and set_fixed). The first contract declared within Bob.sol is “Alice” and its two functions do not do anything when called. The other contract declared within Bob.sol is “Bob” which has the same two functions with the difference being the input parameter for the Bob functions is an instance of an Alice contract and when one of the functions are called, the function then calls the same function but within the Alice instance. The Alice functions are not declared as View or Pure functions which means they have the potential to modify the state of the program and this is a vulnerability because the calling contract could produce unexpected behavior by using a specialized call-stack. When this Solidity file was scanned by Mythril it recognized this issue and informed the tester. This shows that Mythril takes into account the functions that are being called within a contract and if they are referencing another contract that the functions must be either View or Pure functions for the call to be safe because the attacker could utilize the called function to change the state of the program and produce unexpected results. Manticore and Oyente produced no vulnerability results for this contract which shows that they do not check to the functions that are being called to see if they could modify the state.

This check could be extended to locate the contract that is being called (which is not hard in this case because the contract is defined within the same Solidity file) and see if it modifies the state in a way which could be exploited or if it is benign. If this was implemented this contract would be classified as non-vulnerable because technically the Alice functions do not do anything once they are called so they cannot be used maliciously. This would be difficult to implement because it would have to check for

exactly what the contract is doing and it would be hard to determine if it could be used maliciously in any case. For now, it is better to be cautious and mark the contract as a potential threat.

Contract – RaceCondition

Mythril found a reentrancy vulnerability and an external call to an untrusted contract. Oyente did not find any vulnerabilities in the contract. Manticore was unable to run this contract.

In the file RaceCondition.sol there are two contracts defined. The first is called "ERC20" and it complies with the known ERC20 standard to define which functions can be called through this contract via the underlying token. The second contract defined in RaceCondition.sol is "RaceCondition" where two functions are defined. A buy function is implemented that allows someone to call it and become the new owner as well as change the price to a new price that they specify in the input. The second function is a changePrice function that can only be called by the current owner. If the owner calls the changePrice function once someone has already called the buy function they can force them to spend more Ether to purchase ownership. This is a race condition / transaction ordering dependence. Mythril determined that the transfer of money from the buyer to the owner was an untrusted external call and advised to make sure the buyer is a trusted contract. It also specified that the changing of state after the external call could result in business logic failure or a reentrancy vulnerability if the buyer recalled the function when it is called to transfer the money.

Mythril did not specifically determine that there was a transaction ordering dependence but it did specify that there was an external call and it needed to be treated carefully.

Oyente did not find any vulnerabilities and it does actively search for transaction ordering dependence and reentrancy vulnerabilities. The reentrancy vulnerability is not dangerous in this case so it could be that Oyente determined this and did not report it for this case but this is unlikely and it would be more useful to notify the tester of the external call for them to check on their own. It seems as if Oyente does not handle token transfers well because they are abstracted through the ERC20 functions rather than being implemented as direct transactions sent to external contracts. It also did not identify (the actually existent) transaction ordering dependence which is attributed again to the fact that it does not handle token transfers the same as direct transactions.

Contract – ReentrancyExploit

Mythril found an unprotected function, a reentrancy vulnerability, and an external call to an untrusted contract. Oyente found a Transaction Ordering Dependence and a Reentrancy vulnerability. Manticore was unable to run this contract.

This contract is actually designed to exploit the Reentrancy contract that is discussed next. We thought it would be interesting to examine this contract as well to see if vulnerabilities are found within it. Mythril gave a warning each time the call() function was called which is to be expected as shown in the results above. An interesting warning that was also given by Mythril was an unchecked suicide warning. This lets the tester know that the suicide function is used within the contract and appears to be callable without restriction. The EVM suicide function sends all Ether stored at the contract's address to a

specified address and frees the bytes associated with the contract effectively erasing it from the Ethereum blockchain. If this contract was actually deployed to the Ethereum blockchain it could be destroyed by calling the function containing the suicide function.

Oyente also made note of the suicide function being used within the contract but labeled it as a transaction ordering dependence. This makes sense because this function could be called at opportune times to try and deny the sending of owed either in some cases but in this contract it is not actually a vulnerability. Oyente only took issue with one of the call commands issued in the contract and it was within the function that sets the address of the vulnerable contract to be exploited. It labeled this line as both a reentrancy vulnerability and a transaction ordering dependence. This makes sense because it sets the address of the vulnerable contract so if it was to be called again during the attack it would mess up the call stack and thwart the attack. It is also a reentrancy vulnerability because this function can be called by anyone and it sets the receiver of the Ether within the function so someone could call this function and name their own contract as the vulnerable contract and Ether would be sent to them where they could call the function again in their own fallback function.

Contract – Reentrancy

Reentrance.sol is a small banking type contract where there are three main functions: getbalance, addToBalance, and withdrawBalance. There are also two more functions within Reentrance.sol and they are withdrawBalance_fixed and withdrawBalance_fixed_2 which attempt to show how the vulnerable withdrawBalance function could be altered to fix the vulnerability. withdrawBalance_fixed attempts to fix the reentrancy bug by moving the reassignment of the funds available to that account before the Ether is sent to the withdrawer. withdrawBalance_fixed_2 fixes the bug by using the transfer function rather than the call function to send the funds because transfer and send are safe against reentrancy vulnerabilities.

Mythril gives a warning for each line in the contract that sends Ether whether it is using the transfer function or the call function which shows it does not make a distinction between the two and would rather the tester be aware of the Ether transfer either way.

Oyente does not mark withdrawBalance_fixed_2 as vulnerable which shows that it makes the distinction that the transfer function cannot be exploited for reentrancy purposes because it does not pass along extra gas to perform more instructions. It does mark withdrawBalance as a transaction ordering dependency as well as a Reentrancy vulnerability. It seems that it is simply marking it as a transaction ordering dependence due to the call to another contract even though there is no vulnerability present because there is no way the state could be modified during the block executions to take advantage of this contract. It also marks withdrawBalance_fixed as a transaction ordering dependency but not as a reentrancy vulnerability and it declares withdrawBalance_fixed_2 as safe which shows that it is advanced enough to determine that the remapping affects the flow enough to fix the reentrancy vulnerability.

Contract – GiftBox

All of the tools found the integer underflow, Mythril kept giving warnings for every single send that existed, and Oyente continued giving transaction ordering dependence results for transfers.

Contract – Lottery

Mythril and Oyente found an integer underflow. Oyente also found a transaction ordering dependence. Manticore did not find any vulnerabilities.

Mythril only found two issues with the contract and they were both integer underflow possibilities. It was correct in finding that there was a possibility for an integer underflow in two instances; however, both instances were during the process of reseeding the random number generation. This would not have affected the execution of the contract unless the attacker could have used this information to figure out the seed and then find the next lottery number. It would be interesting to try and find a way to automatically determine if the vulnerabilities being reported are actually dangerous in the context they are present within but this could lead to false negatives so this would be difficult to program.

Oyente also found the same integer underflow lines but also marked all transfer functions as transaction ordering dependencies. It is obvious that Oyente marks all transfer of Ether as transaction ordering dependencies even if they are following the correct procedures which is less useful than a more intuitive analysis.

Manticore did not find any vulnerabilities or errors in the contract which is interesting because it seems that it does not consider integer underflows if the results are not being written to storage which makes sense but could miss some edge cases that could be exploited.

Contract – VarLoop

Mythril found an external call to an unprotected contract. Oyente found a transaction ordering dependence. Manticore did not find any vulnerabilities. Mythril and Oyente both consider calls dangerous, Mythril does not take issue with transfers and Manticore does not support finding results about sending Ether. Mythril and Oyente are just warning that it might be weird but not declaring a reentrancy vulnerability.

Contract – PrivateBank

Mythril and Oyente both perform as expected on this contract with regards to the results above but Manticore reports two integer overflows within the contract “Log” within PrivateBank.sol. There are no addition operations contained within the Log contract but there is a function named addMessage which pushes the message onto a History of messages. This leads to the conclusion that Manticore also looks for functions with the keyword add contained in it which could prove useful but can also lead to false positives with regards to integer overflow.

5. Issues

Progress thus far has seen very little in the way of roadblocks. Blockchain is an up-and-coming technology that has the tech world in a buzz, which means that information on the topic is widely available. One slight issue that our team has encountered is that several of the tools that could be useful for developing a framework are commercial products and closed-source. This means that an in-depth audit of some of the tools will be difficult to accomplish. This is only a minor setback because even without source code there are methods to determine the effectiveness of these tools.

One larger setback that we encountered was with the scope of our project. Our initial goal was to expand on some of the already existing frameworks in order to integrate multiple kinds of smart contracts. After speaking with our technical director, who has been working on that same sort of project for several years, he advised that we compare different frameworks and analyzation tools rather than trying to expand on one. To account for this change, we had to reexamine our timeline and tweak it. In our initial Gantt chart, we only did research on some smart contract development tools and then wanted to expand on the one that we thought was best from our research. After changing the scope, this was the main focus of our project. Rather than just doing research on some development and analysis tools, we researched all of the tools that we could find, narrowed it down to a few that we thought were the best, then did an in depth analysis and comparison between these tools, which was a big change from our original plan.

Other setbacks that weren't quite as big of an issue, but still something that we had to deal with was the actual analysis of the tests produced by the tools as mentioned above. The way in which each tool analyzed a contract was different than the next. For example, one tool used code coverage to test how much contract code could be reached, while another tested the contract using symbolic execution trying to find exploitable vulnerabilities. There was also a large difference in time between some of the tools. This made it hard to compare the tools because a big question that was asked was whether or not we would rather have a faster tool, or a more in depth tool.

6. Conclusions and Recommendations

6.1. Interpretation of Findings

The impact of this research on the larger community of smart contract developers will be that smart contracts will be secured before they are deployed to this blockchain. This work was focused on Ethereum smart contracts but the overarching goal is to bring this security to other platforms such as Hyperledger. The effect on public blockchains will have the most immediate benefit because the contracts hosted on these platforms are completely public and as such extremely vulnerable to exploitation. Ensuring that contracts are safe will result in less malicious usage of contracts which will lead to less damages to users of these contracts. This will increase the public opinion of the safety of blockchain smart contracts which is an important step towards adoption of this technology to new sectors.

Expanding this security to Hyperledger and other private blockchains will allow businesses to implement smart contract solutions into their core business design without the worry of exploitation. These contracts are less vulnerable because they are not publicly exposed but they are just as vulnerable when

someone with malicious intent has access to the system. This situation arises when there are insider threats such as disgruntled employees. This is a problem in many businesses and is very important in situations where the underlying data is very sensitive as is the case for many government entities.

6.2. Future Work

The overarching goal of this project is to develop a framework for testing all smart contracts for the most popular blockchains that support smart contracts. This was our goal at the beginning of the semester but after discussing it with Dr. Black we determined that this was not possible in one semester. Our contribution to the overall goal is documentation of the types of vulnerabilities that need to be removed from smart contracts before they are deployed and an analysis of three tools that we determined as the most advanced and utilized in the smart contract community. The next step that needs to be completed is the porting of a tool to Hyperledger or creating a format for one of these tools that current smart contracts can be converted to so that the analysis of the contracts can be more universal across different languages / platforms. Before doing this it is important that improvements be made to the existing tools that were analyzed in this project. Manticore does not check for enough vulnerabilities to be a viable solution and its execution time is orders of magnitude larger than Oyente and Mythril. Oyente is very good at detecting transaction ordering dependence so this portion of the codebase could be extended to Mythril. There were some cases as described in the report above where Oyente was able to determine that a vulnerability was not present even though Mythril reported a warning. This is because Mythril reported all circumstances where there could be an issue but does not expand these portions of the code and follow the execution path to see if it is actually a vulnerability. This portion of Oyente could be ported into Mythril and the warnings could still be generated but it could allow the tester to know if the vulnerability was specifically determined to not be an issue. The end goal is to have a framework where all smart contracts could be scanned before they are deployed and it can be determined whether they are vulnerable to attack or not.

7. References

[1] E. Mik, "Smart Contracts: Terminology, Technical Limitations and Real World Complexity", SSRN Electronic Journal, vol. 9, no. 2, pp. 269-300, 2017.

[2] P. Daian, "Analysis of the DAO exploit", Hacking Distributed, 2018. [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/. [Accessed: 09- Feb- 2018].

[3] D. Siegel, "Understanding The DAO Attack - CoinDesk", CoinDesk, 2018. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists/. [Accessed: 09- Feb- 2018].

[4] Anastasia Mavridou, Aron Laszka, Designing Secure Ethereum Smart Contracts: A finite State Machine Based Approach, International Conference on Financial Cryptography and Data Security (FC 2018)

[5] A Decentralized Exchange Built on Ethereum, https://github.com/etherex/docs/blob/master/paper.md [3] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, Aquinas Hobor, Making Smart Contracts Smarter, ACM, 2016

[6] Ethereum Block Protocol, https://github.com/ethereum/wiki/blob/master/Block-Protocol2.0.md

[7] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, A Survey of Attacks on Ethereum Smart Contracts, Proceeding of the 6th International Conference on Principles of Security and trust – Volume 10204, p. 164-186, 2017

[8]L. Luu, D. Chu, H. Olickel, P. Saxena and A. Hobor, "Making Smart Contracts Smarter", in 23rd ACM Conference on Computer and Communications Security, Vienna, Austria, 2018, pp. 254-269.

[9]"Known Attacks - Ethereum Smart Contract Best Practices", Consensys.github.io, 2018. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/. [Accessed: 9- Mar- 2018].

[10]"Code Coverage for Solidity – Colony", Colony, 2018. [Online]. Available: https://blog.colony.io/code-coverage-for-solidity-eecfa88668c2. [Accessed: 9- Mar- 2018].

[11]"ConsenSys/mythril", GitHub, 2018. [Online]. Available: https://github.com/ConsenSys/mythril. [Accessed: 9- Mar- 2018].

[12]2018. [Online]. Available: https://github.com/trailofbits/manticore/blob/master/. [Accessed: 9- Mar- 2018].

[13]"melonproject/oyente", GitHub, 2018. [Online]. Available: https://github.com/melonproject/oyente. [Accessed: 09- Mar- 2018].

[14]"raineorshine/solgraph", GitHub, 2018. [Online]. Available: https://github.com/raineorshine/solgraph. [Accessed: 09- Mar- 2018].

[15]"SmartCheck", Tool.smartdec.net, 2018. [Online]. Available: https://tool.smartdec.net/. [Accessed: 09- Mar- 2018].

[16]"Securify ◆ Formal Verification of Ethereum Smart Contracts", Securify.ch, 2018. [Online]. Available: https://securify.ch/. [Accessed: 09- Mar- 2018].

[17]"Welcome to Hyperledger Fabric — hyperledger-fabricdocs master documentation", Hyperledger-fabric.readthedocs.io, 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.0/. [Accessed: 09- Mar- 2018].

[18]"The Go Project - The Go Programming Language", Golang.org, 2018. [Online]. Available: https://golang.org/project/. [Accessed: 09- Mar- 2018].

[19]"Frequently Asked Questions (FAQ) - The Go Programming Language", Golang.org, 2018. [Online]. Available: https://golang.org/doc/faq#implements_interface. [Accessed: 09- Mar- 2018].

8. Team

8.1. Biographical Sketches

Ryan Shivers is pursuing a Master's of Science degree in Computer Science with a focus in Cybersecurity. He was awarded a Bachelor's of Science in Computer Science with a focus in Scientific and Software Applications in Spring 2017. He has experience in C++, Java, C#, and Python and has worked as a software development intern for three consecutive summers. This is beneficial because it makes

researching tools written for smart contracts easier by providing an understanding of the languages being used.

Zachary Wallace is pursuing a Master's of Science degree in Computer Science with a focus in Cybersecurity. He was awarded a Bachelor's of Science in Computer Science with a focus in Scientific and Software Applications in Spring 2017. He has experience with a wide variety of programming languages as well as a solid foundation in Cybersecurity concepts and practices.

Samuel Wehunt is a Senior Computer science major at Tennessee Tech pursuing a B.S. / M.S. in cyber security. He is part of the CyberCorp program at Tennessee Tech and thus is pursuing a cyber security career in the government. He has experience programming in C/C++ and Python, which is beneficial in auditing the source code of smart contracts and related code.

8.2. Tasking

All team members have been involved in some way during each phase of this project. Listed below are the tasks each researcher worked on more heavily but all researchers have worked together during each phase.

Samuel focused on Task 1.1, 1.2, and 1.3 of the Gantt chart which included general research regarding Hyperledger and identifying all of the blockchain platforms that support smart contracts and which ones are most prevalent.

Zachary has primarily focused on Tasks 2.1 and 2.2 of the Gantt chart. He was the primary researcher involving vulnerabilities, how they can be exploited, and ways to make code more secure.

Ryan has been the primary researcher regarding Tasks 3.1 and 3.2 which involved determining what tools are available regarding smart contract assurance and determining their usefulness. Ryan also did a deeper analysis of the vulnerability that led to the demise of the DAO which relates to Task 2.1.

The whole team tested different contracts using different tools as the tools and contracts were divided evenly and then tested. The results had to be pared down due to incompatibility issues between Solidity versions which meant some tools did not work with some contracts. The team all recorded their results in a spreadsheet and then the researchers performed individual analysis of all of the results and then compared with each other to generate the final results.

9. Appendix

A. Featured Vulnerabilities

A.1. Timestamp Dependence

There are many contracts that rely on the timestamp of a block in the blockchain in order to execute some sort of operation. These smart contracts are known as timestamp-dependent contracts. A good example to consider when discussing timestamp-depended contract is theRun

contract shown in the figure below [5]. This contract uses a random number generator in order to pick a winner for a lottery sort of game. In order to generate random numbers, the generator needs some sort of seed. This contract uses the hash of a previous block as the seed to select the winner (Line 9-10). The way that it decides which block to use is by looking at the timestamp of the current block (Line 5-7). The miner of the block is the person that sets the timestamp for that block. Most of the time, this is set to the time of the miner's system, but this value can also be changed by about 900 seconds while still being an acceptable block [6]. With the ability to manipulate the timestamp, a user could potentially change the outcome of these timestamp-dependent contracts.

```
1 contract theRun {
2   uint private Last_Payout = 0;
3   uint256 salt =  block.timestamp;
4   function random returns (uint256 result){
5     uint256 y = salt * block.number/(salt%5);
6     uint256 seed = block.number/3 + (salt%300)
7                  + Last_Payout +y;
8     //h = the blockhash of the seed-th last block
9     uint256 h = uint256(block.blockhash(seed));
10    //random number between 1 and 100
11    return uint256(h % 100) + 1;
12  }}
```

Figure 5: A real contract which depends on block timestamp to send out money [22]. This code is simplified from the original code to save space.

Attack Scenario: As mentioned and shown above in theRun, since a miner can change the value of the timestamp, he can make the random number generator choose a number that favors him. The hash of the previous block, the block number, and last payout are all known. Therefore a miner can go ahead and compute and select the timestamp so that either he or whoever he chooses has a favorable chance of being chosen by the function random. There are many other contracts that are dependent on the timestamp of blocks in the blockchain. In one paper, out of the first 19,366 contracts they examined, 83 of them depend on the timestamp to transfer Ether to different addresses [5].

A.2. Mishandled Exception Disorder

Smart contracts in Ethereum are written in a language called Solidity. Solidity has the power to throw exceptions when certain events happen including the following: call stack limit reached, execution of a contract runs out of gas, or the throw command is called. One bad thing about Solidity though is that the way it handles exceptions are not uniform and can be seen in the lines of code below [7].

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { uint x=0;
                 function pong(Alice c){ x=1; c.ping(42); x=2; } }
```

If a user invokes Bob's pong function, but Alice's ping throws an exception then the entire execution stops and everything is reverted back to the way it was before the call was made. On the other hand, suppose that Bob invokes the ping function by another call. In this scenario, only the side effects will be reverted. False is returned by the call, but the execution would continue and set x to be 2. Not having these exceptions handled in a uniform manner could easily affect the security of some contracts, especially when a contract is in charge of transferring ether.

Attack Scenario: In Ethereum there is a contract call the King of the Ether Throne or KoET for short. This is a game that players or users compete to be the King of Ether. In order for someone to be the new king, he or she must pay what the current king is asking to get the contract – this is knows as compensation. The profit the king makes is calculated by subtracting what he paid for the contract from the amount that he sold it for. This contract has a vulnerability though. The contract contains a send function that sends the compensation to the current king. The send function, though, doesn't check for exceptions. Therefore, if for some reason the compensation fails to make it to the king, he simply does not receive the compensation but still gets removed from the throne and the contract keeps the amount of the compensation.

A.3. Transaction Ordering Dependence

As previously mentioned, a block in the blockchain consists of sets of transactions which change the state of the blockchain every so often. Since multiple people can invoke a transaction on the same contract at the same time, this can cause discrepancy in the state of the contract between the users. For example, if the blockchain is in state X and the new block that is added contains two transactions then the users don't know for certain what state the contract is at when their individual invocation is executed [4]. The users that invoke those transactions on the contract can't control the order in which they will be executed and updated, because that relies solely on the miner of the block. Therefore, the final state of that contract will depend on the order the miner chooses to invoke those transactions.

You may not see the immediate impact of this, but there are at least two scenarios that can happen when there is a dependence on the transaction ordering of a contract. The first scenario isn't really a harmful one, rather it produces unexpected results to the users that invoked the transactions. The second scenario on the other hand is a much more malicious scenario where the user actually exploits this vulnerability in order to steal another user's money.

Scenario 1: Consider two transactions (T1 and T2) that invoke the same contract at the same time. Imagine that T1 is the owner of the contract, and he is trying to update the reward for the puzzle. T2 is submitting the answer for the puzzle and is expecting a certain reward for his

solution. The order that the miner invokes the transactions could make a big difference on how much T2 gets as his reward. If the owner of the puzzle sends a transaction to double the reward due to the difficulty of solving the puzzle, but T2 gets invoked first, then he will only receive half of the reward that he should. In the opposite scenario, if the owner decreased the amount and his contract gets executed first, then T2 will receive much less than he is expecting. This isn't a very significant problem in things such as puzzles, but imagine the case of someone is trying to buy something from some store or market place. The buyer could end up paying way more for something than he or she is supposed to. These contracts are known as a decentralized exchange (i.e., EtherEx) [5].

Scenario 2: Consider the same sort of scenario, but instead of it being accidental a user could purposefully exploit this vulnerability in order to obtain free solutions to his problems. There is a time gap between when a transaction is broadcasted and when it actually appears in a block, and in Ethereum this gap is about 12 seconds. If the owner of the contact is listening to the network until he sees a solution transaction being broadcasted, he could quickly change the reward for the solution to 0 and therefore get a free solution to his puzzle. The owner could also influence the choice of which transaction would take place first by participating in the mining himself and setting a higher gasPrice for his transaction.

A.4. Reentrancy Vulnerability

The reentrancy vulnerability is one of the most well-known vulnerabilities in smart contracts. This is what was exploited during TheDao hack where one user managed to get about 60 million US dollars. In Ethereum, contracts can call other contracts, as discussed before. When a contract makes the call to another, the current execution of that contract is halted until the call is finished. The intermediate state of the calling contract can then be exploited by a malicious user.

Attack Scenario: For this scenario we will consider the SendBalance contract shown below [5]. The contract that is being called by SendBalance can simply call it again making use of this intermediate state. On line 11, you can see that the balance of the userBalance is sent to whatever the contract address of the contract asking to withdraw its balance. Since the value of the userBalance variable is only zeroed after the call is completed, the contract could withdraw the balance over and over again until all of the ether for that contract has been exhausted or until all of the gas is used up.

```
1 contract SendBalance {
2  mapping (address => uint) userBalances;
3  bool withdrawn = false;
4  function getBalance(address u) constant returns(uint){
5   return userBalances[u];
6  }
7  function addToBalance() {
8   userBalances[msg.sender] += msg.value;
9  }
10 function withdrawBalance(){
11  if (!(msg.sender.call.value(
12    userBalances[msg.sender])())) { throw; }
13  userBalances[msg.sender] = 0;
14  }}
```

Figure 7: An example of the reentrancy bug. The contract implements a simple bank account.