Portland State University

PDXScholar

Dissertations and Theses Dissertations and Theses

5-5-2020

Smart Contract Vulnerabilities on the Ethereum Blockchain: a Current Perspective

Daniel Steven Connelly
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Computer Sciences Commons

Let us know how access to this document benefits you.

Recommended Citation

Connelly, Daniel Steven, "Smart Contract Vulnerabilities on the Ethereum Blockchain: a Current Perspective" (2020). *Dissertations and Theses.* Paper 5440.
https://doi.org/10.15760/etd.7313

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Smart Contract Vulnerabilities on the Ethereum Blockchain: A Current Perspective

by

Daniel Steven Connelly

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Wu-chang Feng, Chair
Nirupama Bulusu
Charles Wright

Portland State University
2020

Abstract

Ethereum is a unique offshoot of blockchain technologies that incorporates the use of what are called *smart contracts* or *DApps* – small-sized programs that orchestrate financial transactions on the Ethereum blockchain. With this fairly new paradigm in blockchain, however, comes a host of security concerns and a track record that reveals a history of losses in the range of millions of dollars. Since Ethereum is a decentralized entity, these concerns are not allayed as they are in typical financial institutions. For example, there is no Federal Deposit Insurance Corporation (FDIC) to back the investors of these contracts from financial loss as there is with bank depositors. Furthermore, there is also no Better Business Bureau (BBB) or Consumer Reports organization to offer any sort of ratings on these contracts.

However, there exists a well-known method for verifying a program's integrity; a method called *symbolic execution*. Such an examination promises to give not only a perspective on the security of Ethereum, but also highlight areas where security experts may need to target to more quickly improve upon the security of this blockchain.

This paper proposes a solution to ensuring security and increasing end user confidence -- a digital registry of smart contracts that have security flaws in them. A rating system for contracts is proposed and the capabilities one has with knowledge of these vulnerabilities is examined. This research attempts to give a picture of the current state of security of

Ethereum Smart Contracts by employing symbolic analysis on a portion of the Smart Contracts up until approximately the 8.4 millionth block.

Vulnerabilities in Smart Contracts may be prevalent and, if they are, a registry for enumerating which ones are can be built and potentially used to easily enumerate them.

Table of Contents

Abstract i

List of Figures v

List of Tables vi

Acknowledgements vii

1. Introduction 1

1.1. What Is a Blockchain? 2

1.2. Ethereum Smart Contracts: A Different Kind of Blockchain 3

1.3. Application Binary Interface (ABI) 5

1.4. Security of Smart Contracts, Historical Losses 7

1.5 A Survey of The Security of The Ethereum Blockchain 9

1.6 Symbolic Execution 11

1.7. Mythril 12

1.8. Symbolic Execution Specifics 13

1.9. Related Work 16

2. Mythril Vulnerability Type Examples 17

2.1 Vulnerability Walkthrough 17

2.2 Vulnerability Types 21

3. Methodology 24

3.1. Google Cloud Platform, Machine Type 24

3.2. Contract Address Obtainment 26

3.3. Mythril Data Collection 27

3.4. Data Parsing, Storing, Retrieving 29

4. Results 30

4.1. Number of Contract Addresses 30

4.2. Vulnerable Ethereum Contracts 32

4.3. Amount of Vulnerabilities 33

4.4. Amount of Ether at Risk 36

4.5 Coverage Amount 37

5. Discussion 38

5.1. Ethical concerns 39
5.2. Veracity of Results 40
6. Exploitation 42
6.1. Interacting with Contracts Via Partial ABI 42
6.2. Decompiling Contracts to Exploit 45
7. The Data Registry: haveibeenexploited 46
8. Future Work 49
8.1. Automating Analysis 49
8.2. Rating System 50
8.3. Mythril Results as Guide for Further Execution 50
8.4. Registry for Other Blockchains, Registry Built-in to Blockchains 51
9. Conclusion 53
10. References 55

List of Figures

Figure 1 - ABI Contract Example [4] 6
Figure 2 - Symbolic Execution Example [9] 12
Figure 3 - Self Destruct Example 18
Figure 4 - Unprotected Ether Withdrawal Example 19
Figure 5 - Integer Overflow/Underflow Example 20
Figure 6 - High-Level Overview of Data Collection Program [26] 28
Figure 7 - Contract Addresses in 9 Million Blocks 31
Figure 8 - Live Contract Addresses in 9 Million Blocks 32
Figure 9 – Number of Vulnerabilities by Type 34
Figure 11 - Exploits Per Millionth Block 35
Figure 12 - ETH at Risk Per Millionth Block 36
Figure 13 Pt. 1- Exploit Contract With Partial ABI 43
Figure 14 Pt. 2 - Exploit Contract with Partial ABI 44
Figure 15 – www.haveibeenexploited.com 46

List of Tables

Table 1 - Summary of Vulnerabilities Mythril Catches [23] 23

Acknowledgements

I owe special thanks to Professor of Computer Science Dr. Wu-chang Feng. Without Dr. Feng, I would not have had any knowledge to work with in Blockchain or in Cloud Computing, both of whose subject knowledge this thesis relied heavily on. Furthermore, without his additional guidance in crafting this thesis, I would have gotten into many sticky situations, which were ultimately avoided.

I owe another thanks to the Ethereum Foundation for connecting me with a community ---- the ETHSecurity Community Telegram channel -- to ask questions and gain informational resources.

Finally, thanks must go to the two funders of this project. The first, The National Science Foundation (NSF) grant *Curricula and CTF Exercises for Teaching Smart Fuzzing and Symbolic Execution* (#1821841). The second source, the Google Faculty Research Grant, which generously gave $5,000 worth of cloud research credits for Google Cloud Platform use. Without the NSF grant and compute resources afforded by these Google Cloud credits, this project would not exist at its current level of maturity.

1. Introduction

Blockchain technologies are quickly becoming the rage and have reached proportions of popularity similar to other relatively newer fields such as Artificial Intelligence and Machine Learning. LinkedIn, for example, lists blockchain as the most in-demand hard skill in 2020 [1]. Other corroborating data suggests that it would behoove developers, companies, institutions, and governments to adopt blockchain as the technology claims to be a panacea for many problems.

However, using blockchain for financial use cases can be a dangerous gamble. For example, there are many instances where financial losses have been incurred due to insecure programming. These mistakes are often more serious than traditional software system bugs due to the financial consequences (which are uninsured) and the often-permanent bugs on an immutable blockchain.

However, there exists methods, such as *Symbolic Execution*, to identify insecure programming which yield reliable results for up to 50% of programs [2]. These results *could* be aggregated into a data registry where users could search before using buggy software; for example, in a blockchain such as Ethereum. This is the subject matter of this paper. However, no registry currently exists in the Ethereum ecosystem. The following chapter and subsections describe these topics in more depth as well as describe fundamentals of the Ethereum blockchain.

1.1. What Is a Blockchain?

A blockchain, at its simplest, is an anonymous, immutable, distributed, append-only *ledger* shared among a peer-to-peer network. In this ledger, a number of transactions between users of the network are recorded and each transaction is given a timestamp and put into a set, called a *block*. Each block corresponds with an interval of time that the transactions were made in.

The first ever mainstream blockchain, and still the most popular, is Bitcoin. Bitcoin has its own currency, BTC, that users can buy and then trade amongst one another. Sending and receiving BTC with other peers in the network is what makes up the transactions within the blocks of Bitcoin. However, since sensitive financial information of this kind is stored amongst peers and shared, there must be a way to verify that the information stored and then disseminated to other peers has not been tampered with.

This is where *miners* come in. Miners are the verifiers of most blockchains. Verification is computed by each miner through a process where all the transactions in a block are hashed separately and each hash is part of the input to another hash, creating a tree-like structure of hashes, called a *Merkle Tree*. A single root hash is produced by the miners through this operation. The miners whose root hash match a mathematically determined fraction of other miners[1] in the network get their block immortalized into the ledger and added to a

[1] *For more information, see The Byzantine Generals Problem.*

"chain" of verified blocks (the origin of where blockchain receives its name). This process is designed to prevent miners from being untruthful as to the number of transactions and/or the content that was generated within each transaction in a respective block.

1.2. Ethereum Smart Contracts: A Different Kind of Blockchain

Ethereum is another similar, yet unique, creation of blockchain. Like Bitcoin, it has its own currency, called ETH. But the Ethereum network is a unique blockchain in that, unlike Bitcoin, it has a stack based, big-endian Ethereum Virtual Machine (EVM). The EVM is a similar construct to the Java Virtual Machine (JVM) and is used to run programs called *Smart Contracts,* which are decomposed into byte-level EVM instructions. Much like the JVM, there exists opcodes that enable a program to perform calculations and move a state machine forward such as ADD, SHL, LT, etc. [3]. In a similar fashion to the Java language and the JVM, these byte-level instructions are created by compiling a programming language, named Solidity, down to EVM-compatible bytecode instructions where these contracts may be run.

Contracts possess unique characteristics in comparison to other programs in other languages, however. When a contract is created, it is typically sent to the network where its EVM instructions can be ran by members of the network; this sending action is called *deploying* a contract. In the process of deployment, a contract's constructor is called once, and the bytecode minus the constructor code is posted to the immutable blockchain where its bytecode is unalterable. The only way a program is no longer available is if its bytecode

is destroyed through the use of an optionally programmed function that invokes the EVM *self destruct* instruction. This function is left up to the developer to include or not when writing the contract.

Contracts also have similar characteristics to regular programs that enable them to be used in a variety of ways. Like other programs, Ethereum contracts have a *name*. Once submitted for deployment on the Ethereum network, contracts are given a unique 42-character hex address consisting of numbers and upper and/or lower-case letters (e.g., 0x12a34C55...). This allows users to quickly lookup transactions in the ledger that this address was involved with (through an online website that tracks the ledger, for example). It also allows users and other contracts to *call* functions within the program, given that they have additional information on the contract (see ABI section 1.3).

Secondly, contracts can *contain state*. This enables a contract to hold *ETH*, the main form of currency for Ethereum. The contract can be loaded with ETH at deploy time, but also sent ETH at any time by specifying the address (i.e., name) that belongs to the contract. The contract can also send ETH to other contracts or send ETH straight to users, who have their own addresses called *wallet addresses*, depending on the contract's business logic.

With the technology to run these programs and keep track of money going in and out, a contract can do many things that would otherwise require intentional attention and effort on the part of a human agent, such as acting as a middleman between two parties. For

example, if buying a house, a person may put money into a contract and be transferred the deed electronically. When setting up a trust fund, a contract could release the funds at a given date(s). There are also many use-cases beyond these such as investing, gambling, and building other blockchains on top of this system.

1.3. Application Binary Interface (ABI)

However, even though a contract may be located on the Ethereum blockchain, by default, not everyone may interact with the bytecode on the blockchain. A mechanism of each contract, named the *contract dispatcher* (or *function selector*) must be told which function is being called by a user or contract. Once the dispatcher knows this, the appropriate bytecode is run. This is primarily because sections of bytecode are associated with the hash of a function's signature. If this dispatcher did not exist, there would be no system to tell what bytecode should be run as the bytecode hash is one-way.

An Application Binary Interface (ABI) takes care of this translation between bytecode and a user's desire to call functions at a human-readable level. As a contract is turned into bytecode, each function's signature (made up of name, parameter types...) is hashed with a Keccak-256 hash (SHA-3) and the leftmost four bytes are included in the bytecode and associated with the bytecode of the function body.

Each specific function call to a contract from a user or contract is located in the Javascript Object Notation (JSON) format called *calldata*. The hex value making up the calldata

begins with the respective 4-byte Keccak-256 hash of that function's signature. When a function call is made to a contract, the relevant bytecode is executed by deriving and matching up the bytecode with the corresponding 4-byte address using the aforementioned contract dispatcher. Any bytes in the *calldata* after this 4-byte value represent the function's parameters and are represented with 32 bytes (padded if necessary).

For a concrete example, take the following contract:

```
1 pragma solidity >=0.4.16 <0.6.0;
2
3 contract Foo {
4   function bar(bytes3[2] memory) public pure {}
5   function baz(uint32 x, bool y) public pure returns (bool r) { r = x > 32 || y; }
6   function sam(bytes memory, bool, uint[] memory) public pure {}
7 }
```

Figure 1 - ABI Contract Example [4]

A method call to the function baz would create the 4-byte hash *0xcdcd77c0*. If we chose to pass 69 into the place of *x*, it would be represented in hex as:

0x0000000000000000000000000000000000000000000000000000000000000045.

If we passed true into the value of *y* the hex value passed int becomes:

0x0000000000000000000000000000000000000000000000000000000000000001.

In total, the call data would be:

0xcdcd77c000000000000000000000000000000000000000000000000000000000000000

4500000000000000000000000000000000000000000000000000000000000000001.

The contract dispatcher would strip the first 4-bytes to know which function to interact within a contract and then take each 32-byte segment and pass into that function as parameters.

This ABI potentially shared with users of that contract. By having the ABI, one can interact with a deployed contract as any capable library (e.g., Web3) or website (e.g., remix.ethereum.org) can easily convert the desired function into its 4-byte equivalent with any parameters and call the relevant function that matches a section of deployed bytecode.

1.4. Security of Smart Contracts, Historical Losses

Since contracts can support financial transactions in variety of use cases, it is well worth asking the question of how secure these contracts really are and if there are any examples of security snafus. With a small amount of research, one can conclude that there are many documented issues with Smart Contracts.

Firstly, contracts are often programmed in Solidity. This is a potential problem as Solidity is a historically insecure language that does not by default guarantee safety from programming flaws, such as integer underflows or overflows that other languages may

protect against. There are indeed entire papers and researchers that study and categorize ways that Solidity-based contracts can be vulnerable [5].

Developers of Smart Contracts may intentionally add in code that takes advantage of the insecure language constructs of Solidity. For example, developers may write functions that allow for the developer to have total control over the funds users' put in a contract, pocketing them at a time of his or her choosing. Other developers may make errors (that aren't caught by Solidity compilers) that also play on insecure language constructs.

Secondly, malevolent parties of varying sizes may search out contracts which contain errors in their programming and exploit vulnerabilities to extract ETH from contracts. The access may be unfettered as long as an attacker knows the first 4-bytes of the keccak256-hash of a vulnerable function signature and the method is public. Some of these vulnerabilities are easy enough to be exploited by bots where others are more complicated and require more analysis to exploit.

Historically, there are well documented cases of vulnerabilities from different sources. A recent example that demonstrates the gravity of this situation was seen in a contract named the Fairwin contract. Multiple vulnerabilities were discovered by security researchers in the popular gambling contract which once contained a peak of $10.5 million in funds and now contains $0 [6]. One of the most concerning of the vulnerabilities discovered allowed the owner of the contract to drain the contracts funds completely (among a few other

security issues) -- a loophole that made many call this contract a Ponzi Scheme [7]. Since there is no FDIC insurance or equivalent institutions to insure members of insecure contracts, investors only option and hope were to pull their money out. Likely most investors did manage to pull their money out, but had there been no warning from experts, many people may have lost all of their funds.

A less recent example was the DAO contract. The DAO contract contained the equivalent of $150 million USD in it and was partially drained of funds by exploiting what is called a reentrancy bug. This caused millions to be initially lost [8]. In an uncharacteristic move of a decentralized and immutable technology, the Ethereum community decided to roll back to an earlier block before the exploit occurred, essentially undoing the exploit and replacing the vulnerable code. This, however, is not characteristic of the Ethereum Blockchain and may never happen again.

1.5 A Survey of The Security of The Ethereum Blockchain

The story of this ‘almost’ loss as well as the Fairwin contract are important as they reveal the severe consequences these vulnerabilities can have on community users. With literally millions of dollars having been siphoned out of insecure contracts through purposeful intent or clueless error, a survey and system to analyze the fortitude of contracts can help improve the security of Ethereum.

Such an examination promises to give not only a perspective of how bad the current situation is globally, but also highlight local areas where leading security experts may need to target to more quickly improve upon the security of Ethereum. Furthermore, results garnered could also be released publicly so developers and users can take advantage of this information. This thesis proposes that creating a registry containing addresses of insecure contracts would improve the ecosystem by helping developers be aware of poorly written code and by helping users avoid insecure contracts.

Having a registry promises to provide a number of benefits to users and developers. Users of contracts can search this registry preemptively before using a contract. This would, in theory, lower the number of investments in insecure contracts. For contracts that exist currently, users may be able to pull their funds if they discover a contract is insecure and avoid a vulnerability from being taken advantage of.

Developers may decide to redeploy contracts where bugs have been discovered in their code and alert the users of their contracts that a vulnerability has been identified that puts their ETH at risk so they can withdraw any investment. If a developer programmed a self destruct function, they could also remove this bytecode completely from the network, never to be used again.

Furthermore, both users and developers may not be aware of symbolic execution engines and other static and dynamic analysis tools that reveal vulnerabilities in Smart Contracts.

By allowing a research project such as this to research the best tools and run them on all the contract addresses on Ethereum, users and developers can receive information about the security of contracts without needing to know a list of symbolic analyzers. Additionally, some of these tools require careful study of documentation and background programming knowledge that may be a barrier to entry for some individuals.

In addition, getting research results takes time and money. Having a single effort handle the scanning and parsing out of millions of lines of output regarding millions of contracts is advantageous and would save an exponential amount of time for the users of this proposed registry. Electricity costs can also be quite high, especially when considering a large-scale effort to assess the safety of multiple contracts. By sharing the results of analysis and allowing multiple people to benefit, less computational power is being wasted.

1.6 Symbolic Execution

The primary method used in this paper for creating such a registry is called *symbolic execution*. Symbolic execution works like algebra for computer programs. In a nutshell, any variable, let's call it λ, is declared that can take on any unknown value. This unknown value may trigger a condition in the program that sets off a control flow that may or may not be desired. Take for example the below figure. If λ = "hunter2", then the if condition prints out "good job" is executed.

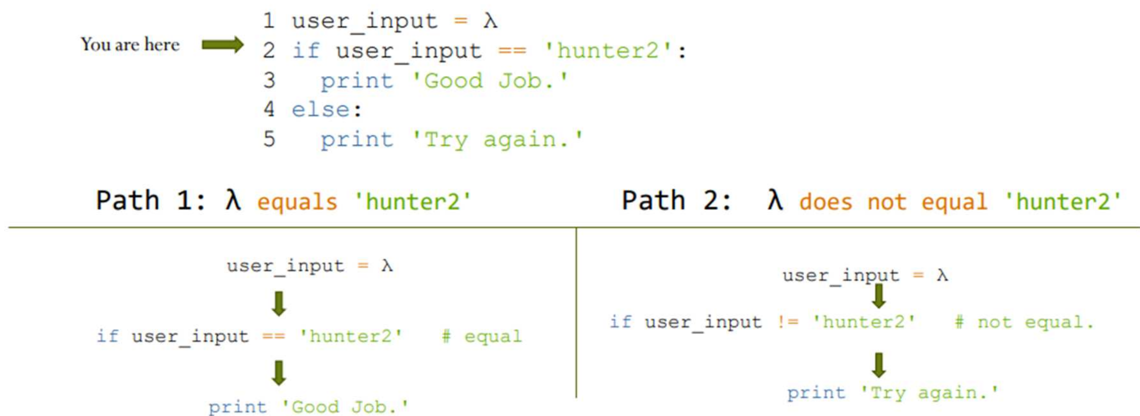

Figure 2 - Symbolic Execution Example [9]

The job of a symbolic execution engine is to discover the state in which a particular engine can reach a certain state. For example, on line 2 have λ == “hunter2”. This typically involves the use of a constraint solver to determine the inputs and branching/path logic that would lead to such a state.

But what if, instead of that print statement, the logic was “send all ETH to the (perhaps malicious) person who called this program”. That would be good information to know for investors of Smart Contracts! Fortunately, there are symbolic execution engines built to work on EVM bytecode as well that could detect this and other disastrous programming logic.

1.7. Mythril

Mythril [10], is the exclusive symbolic execution engine used in this project to determine vulnerabilities in Smart Contracts. By covering all possible states of a contract as a result

of a function(s) call, Mythril claims to discover potential ways that a contract, programmed in Solidity or another EVM compatible programming language, may contain vulnerabilities. It should be noted that Mythril detects vulnerabilities, but not exploits.

A distinction between *vulnerabilities* and *exploits* should be made here. Vulnerabilities refers to code that contains some insecurity (e.g., Integer Overflow). An exploit refers to the process of abusing a vulnerability (or multiple vulnerabilities) that leads to some payout for the exploiter. Therefore, vulnerabilities can exist, but this does not guarantee that an exploit can occur as a result of the vulnerable programming. The contract could contain no ETH, or the vulnerability may not bear any real consequence on the programming logic, for example. However, some vulnerabilities, operated on alone or in combination with other vulnerabilities, may lead to a successful exploit.

1.8. Symbolic Execution Specifics

Mythril analysis takes the bytecode of a contract and decompiles it into EVM opcode instructions where all possible program states are explored over *n* transactions (two by default). These transactions represent how many times a contract is called. However, Mythril must have an environment to execute this opcode in to reveal vulnerabilities. This is what LASER [11], a symbolic virtual machine (SVM) created for Mythril, smart contracts, is used for.

Within LASER, all possible program states are explored and Mythril makes use of a

number of analysis modules to determine if vulnerabilities exist. This type of symbolic execution is more possible compared to other technologies as each contract consists of a finite number of program states and we know that a call(s) to a program will terminate. Over the span of states explored, if a vulnerable state is encountered (e.g., a Self Destruct Opcode) Mythril then uses Z3, an automated theorem prover, to prove or disprove the reachability of that state.

Most symbolic execution engines work alongside an automated theorem prover, also known as a constraint solver. Z3 is a theorem prover released by Microsoft Research under the MIT license [12]. With a high enough verbosity level (indicated by a flag one can set in Mythril), Z3 produces the input into a function(s) needed to reach that state of vulnerability; a step useful for ultimately exploiting a contract.

But the vulnerabilities Mythril senses cannot always be exploited. In some cases, the Z3 solver employed returns an input that satisfies some path to trigger that exploit, but it does not always return an input. Additionally, in the results obtained in this research, there were multiple bugs encountered that showed some results were faulty. An in-depth analysis of the *Unprotected Self Destruct* vulnerability, for example, is worthy of particular attention. When looking over disassembled code, it was discovered that there was additional logic in most contracts that prevented this vulnerability from being exploited, even when Z3 returned an input to trigger it. This researcher is currently working with Mythril on a volunteer basis to fix these bugs.

Another limitation is the contract state that Mythril operates within – an entirely symbolic one. That is, all variables are symbolic and no real data from the network except for immutable bytecode is used when running Mythril analysis modules. So, in our example of an *Unprotected Self Destruct,* if a flag is set to false on the network which prevents a self destruct opcode from executing and it can never be set again, Mythril is unaware of this and finds a state where that variable may be true [13].

Another limitation is Mythril's default multi-transactional setting. The multi-transactional setting indicates how many calls deep the symbolic execution analyzes in order to find an exploit. By default, Mythril uses two calls/transactions (from a user or contract) to determine if a contract contains a vulnerability and potential exploit. For example, to exploit a contract one may need to leverage an Integer Overflow vulnerability in one call/transaction within one function of the insecure contract and then make another call/transaction in another function of the insecure contract to finally receive a payout. Mythril runs execution, by default, with a level of multi-transactional setting of two. Anything else must be set by the user. However, adding more transactions causes the state space to grow exponentially as each transaction can have multiple valid final states [14], which also increases the time to discover vulnerabilities.

1.9. Related Work

There are many research articles about symbolic execution engines and related topics meant to spread knowledge on Smart Contract vulnerabilities [15], [16], [17]. Additionally, some researchers specialize in studying groups of attacks rather than what each smart contract's level of security may be [18].

There are also projects that seek to apply automatic analysis towards Smart Contracts. For example, www.contract-library.com/ is attempting to automatically run analysis on every new block with their own symbolic execution engine. *Karl* is a software that uses Mythril to gain real-time insights into new contracts on the blockchain [19]. Other tools exist to automatically detect *and* exploit contracts [20], [21], [22].

However, while prior work has sought to analyze varying vulnerabilities, there has been no published work that considers analyzing the major state of security on the Ethereum blockchain for smart contract vulnerabilities. This work is unique in its approach to scanning a large set of contracts and expositing on the results.

2. Mythril Vulnerability Type Examples

Mythril is capable of detecting a range of vulnerabilities. Each vulnerability, in theory, could have some negative consequence on the contract that contains a vulnerability. A chief consequence that can occur is if a vulnerability is able to be exploited. That is, to cause a loss of assets for users, either within its own contract or that of another contract. As of this paper's date, Mythril is able to detect 16 types of vulnerabilities.

Though each of these vulnerabilities are important, there are some that carry more deleterious effects than others. Mythril uses what it calls *severity* ratings to indicate how dangerous a vulnerability may be. Below is an enumeration of possible vulnerabilities as well as select source code examples that demonstrate how they may look like in the wild. A summary table of vulnerabilities is included in table 1.

2.1 Vulnerability Walkthrough

Unprotected Functions

Unprotected functions are identical to what they are in other programming languages: a function that anyone can call and use. Instead of a *private* keyword which Solidity does not have, the use of a modifier that restricts access to the owner of the contract can nullify this vulnerability. Mythril considers a contract high-risk if this class of vulnerability are inside a contract's function(s).

1). Self Destruct

Perhaps the easiest vulnerability to understand is called Self Destruct. The Self Destruct vulnerability is named after the *Selfdestruct* EVM opcode that can send all the ETH inside of a contract to a specified address before permanently deleting a contracts bytecode on the network.

The problem in this vulnerability is not that a contract has the ability to do this operation, but rather that *anybody* may call a contract's function that has a Selfdestruct command inside. For example, the below code is a minimal example of an Unprotected SelfDestruct:

```
1 function close() public {
2  selfdestruct(msg.sender);  // send available ETH to the contract invoker
3 }
```

Figure 3 - Self Destruct Example

In the above code, the method has been declared public, which means anybody can call it (line 1). Furthermore, the person who called it is sent any available ETH in the contract before the bytecode is self-destructed (line 2).

2). Unprotected Ether Withdrawal

The problem with this vulnerability is the same as the one before it, but without the occurrence of a contract being self-destructed. This vulnerability is typically seen in *payable* functions, those that are able to receive ETH from other contracts and wallets. For example:

```
1 function transfer(address _to, uint256 _value) public payable {
2   _to.call.value(_value);
3    balances[msg.sender] -= _value;
4 }
```

Figure 4 - Unprotected Ether Withdrawal Example

In the above example, anyone can call this contract function and pass any value to be transferred from the current contract's balance, so long as the amount actually exists in the contract, to any address (e.g., _to).

Overflows

Integer Overflows/Underflows are cases where variables that hold a certain maximum/minimum value (e.g., 2^256 -1 or a number < 0) are filled with a number greater than or lesser than their maximum or minimum respectively. This type of vulnerability can be allayed by using the types from the SafeMath library, rather than the default Solidity types for numbers, as the SafeMath library will check for overflows and underflows. Mythril considers a contract high-risk if this class of vulnerabilities are inside a contract's function(s).

1). Integer Overflow

A fairly normal operation in programming is to check if a certain condition is true or false and carry on a control flow if the condition is met. However, if default solidity types are used, this can lead to dangerous consequences.

Take for example the following code:

```
1 function transfer(address _to, uint256 _value) public {
2  require(balanceOf[msg.sender] >= _value);
3    balanceOf[msg.sender] -= _value;
4    balanceOf[_to] += _value;
5}
```

Figure 5 - Integer Overflow/Underflow Example

If a caller puts any number greater than the maximum for the type variable uint256 _value (e.g., 2^256), then the variable will overflow, and equal 0. In the above case with an overflowed value, the require statement (much like an if or try statement) will equal true and the transfer will be the same as a null operation, with the *_to* address receiving 0 ETH.

2). <u>Integer Underflow</u>

Integer Underflow works precisely the same way, but in reverse. What if a value, such as -1, was passed into the uint256 in Figure 4? *_value* would then equal the maximum value of an uint256. This would cause the require statement on line 2 to never execute in most cases and the transfer operation to fail. Or, in a worse case, the contract actually *does* have

that amount inside and the contract transfers 2^256 worth of the denomination WEI (the default currency used when ETH is not specified) to the address specified at *_to*.

2.2 Vulnerability Types

The below table represents all the vulnerabilities that Mythril catches, for the reader's reference. The Smart Contract Weakness Classification (SWC), a recent library of vulnerabilities, name, description, and link also are included in this table. Mythril detects 16 vulnerabilities in total.

Mythril Vulnerabilities (SWC Assertion and Link)	**Quick Vulnerability Explanations**
Integer Overflow and Integer Underflow	An overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of a type. E.g., uint8 = 2^8. Overflow could cause if statements to be false when true.
Exception State (Assert violation)	Flow control reaches a failing *Assert()* statement.
External Call To User-Supplied Address (Reentrancy)	A contract calls an external contract that the callee of the contract provides, opening up the possibility for a reentrancy bug.
External Call To Fixed Address (Reentrancy)	A contract calls an external contract that the contract has hardcoded, opening the possibility for a reentrancy bug.
Delegatecall Proxy To User-Supplied Address (DelegateCALL to untrusted Contract/Callee)	A contract uses [address].delegatecall() where outside contract can change local storage or drain contract of balance.

Mythril Vulnerabilities (SWC Assertion and Link)	Quick Vulnerability Explanations
Dependence on predictable environment variable • Detects Weak Randomness • And Timestamp Dependence	Numbers controlled by miners are a bad source of randomness as miners can control the output, and by association a variable that is using that number.
Use of tx.origin (Use of Deprecated Functions)	A deprecated function. May lead to unintended side effects.
Unprotected Ether Withdrawal (Ether Thief)	Function(s) is not protected with the potential net effect being any party may withdraw ETH from the contract.
Multiple Calls in a Single Transaction (DOS With Failed Call)	If an external call fails accidentally or deliberately, a DoS condition can result in the contract as a contract is waiting for a call to return.
State change after external call (Reentrancy)	A contract may call back into the calling contract before the first invocation finishes. This could result in undesirable consequences.
Unprotected Selfdestruct (Unprotected Selfdestruct)	Any party can call the function that has a self-destruct in its contract.
Unchecked Call Return Value (Unchecked Call Return Value)	Return values of a message call must be checked to see if an exception was thrown. Otherwise, the program will continue despite a failed call.
Use of callcode (Use of Deprecated Functions)	A deprecated function. May lead to unintended side effects.

Mythril Vulnerabilities (SWC Assertion and Link)	**Quick Vulnerability Explanations**
Jump to an arbitrary instruction (Arbitrary Jump with Function Type Variable)	A developer may use assembly instruction *mstore* or the *assign* operator, an attacker may point a function type variable to any code instruction.
Jump to an arbitrary line (Arbitrary Jump with Function Type Variable)	A developer may use assembly instruction *mstore* or the *assign* operator, an attacker may point a function type variable to any code instruction.
Write to an arbitrary storage slot (SWC: Write to an Arbitrary Storage Location)	A contract may write to an arbitrary storage location, which could house the contract owners address…an attacker could be renamed the contract owner.

Table 1 - Summary of Vulnerabilities Mythril Catches [23]

3. Methodology

An iterative design of how to obtain contract addresses, symbolically execute on these addresses, and parse the results occurred over a few stages. Generally, a single machine was dedicated to collect the contract addresses, many machines symbolically executed on a subset of these addresses, and the parsing occurred at the end of this research project. Below is the process, in more detail.

3.1. Google Cloud Platform, Machine Type

To measure contracts, VMs on the Google Cloud Platform (GCP) were employed due to the generosity of the Google Cloud Research program. Reasons beyond this generosity, however, made GCP an ideal source for computational power. GCP obviates the need for physically setting up machines, quickly brings up and tears down instances, requires no maintenance, and offers great flexibility in machines rented.

The machine which housed the GETH node required to attain contract addresses had requirements beyond those machines which ran symbolic execution. Therefore, an n1-standard-4 (4 vCPUs, 15 GB memory) was used with 600 GB of Solid-State Drive (SSD) storage.

The machines which symbolically executed on the contract code belonging to the contract addresses obtained were a n1-highcpu-8 (8 vCPUs, 7.2 GB memory) machine type with approximately 50 GB of SSD storage. Early on, it was known that the bottleneck for this

workload was the CPU, not the network or storage I/O. Making a call, for example, across the network API for the bytecode to execute on in Mythril or saving the output of the result was trivial, but actually symbolically executing over a range of possibilities of contract code was what took the longest in Mythril. This machine type was chosen primarily due to its ability to compute and its memory to handle multiple threads.

Another reason for using this type of machine was due to the free-tier API, Infura, that Mythril uses. Infura denies requests that are too close to each other and too frequent. Therefore, a computer with many CPUs could not compute everything in a small timeframe as the API would repeatedly reject requests. Instead, several computers that ran a reduced number of threads to avoid the IPC/RPC 429 Errors were employed.

Each machine ran a Linux Operating System (OS); specifically, Ubuntu 18.04 (Bionic Beaver). This version was chosen as it is a stable version that does not contain bloatware, but has the minimal services needed to run networking requests and other services needed for this project. Another reason this OS was chosen was its useful dependencies. Idle machines were a waste of credits, but there exists no infrastructure to determine when a process has finished and Mythril has an unpredictable timeline for finishing execution. Through collaboration, this research also generated a free text and email notification system for long running processes [25].

3.2. Contract Address Obtainment

There are many ways to attain information about contract addresses and other information from the Ethereum blockchain. One way is to build a webscraper to scrape data from a website that has blockchain information (called a block explorer). Another choice is to use an Application Programming Interface (API) and work within rate limits afforded by the tier your API key belongs to.

The third way is by running an Ethereum full node. This is typically the suggested approach as this gives access to the full ledger of the blockchain. The research conducted here made use of the implementation using the Go programming language, named a Go Ethereum (GETH) node [24]. The primary purpose of using a GETH node for this project was to determine the contract addresses that were needed to feed to Mythril.

Within each block there is a JSON structure that indicates whether an address is the name of a contract. Using this information gained from the GETH node, it was easy to create a program that broke down the structure of each block to find how many contract addresses existed on the blockchain. Each contract was identified whether it was a live contract (had bytecode to operate on) or a dead contract (had no bytecode due to being self-destructed). This meant that the final number of contract addresses received represented all contracts created from the history of the Ethereum network (the “genesis” block) to the 9 millionth block. This research also was able to determine how many contracts were destroyed by determining how many contracts’ bytecode still existed on the network.

3.3. Mythril Data Collection

Approximately 10-12 threads ran Mythril version 0.21.21 in each machine where each thread was staggered by six seconds to avoid rejected requests to the API that Mythril uses. These threads, spawned by a Python program, launched a new bash shell which then used the Docker version of Mythril to run in an isolated environment. Each environment was given one contract address to execute on and store the output in a variable that is local to the main thread which is then written out to files/bins depending on the content (Exceptions, Errors, or Output) -- see below figure for a high level design of this program. The separation of output made parsing possible and also enabled efficient bug reporting.

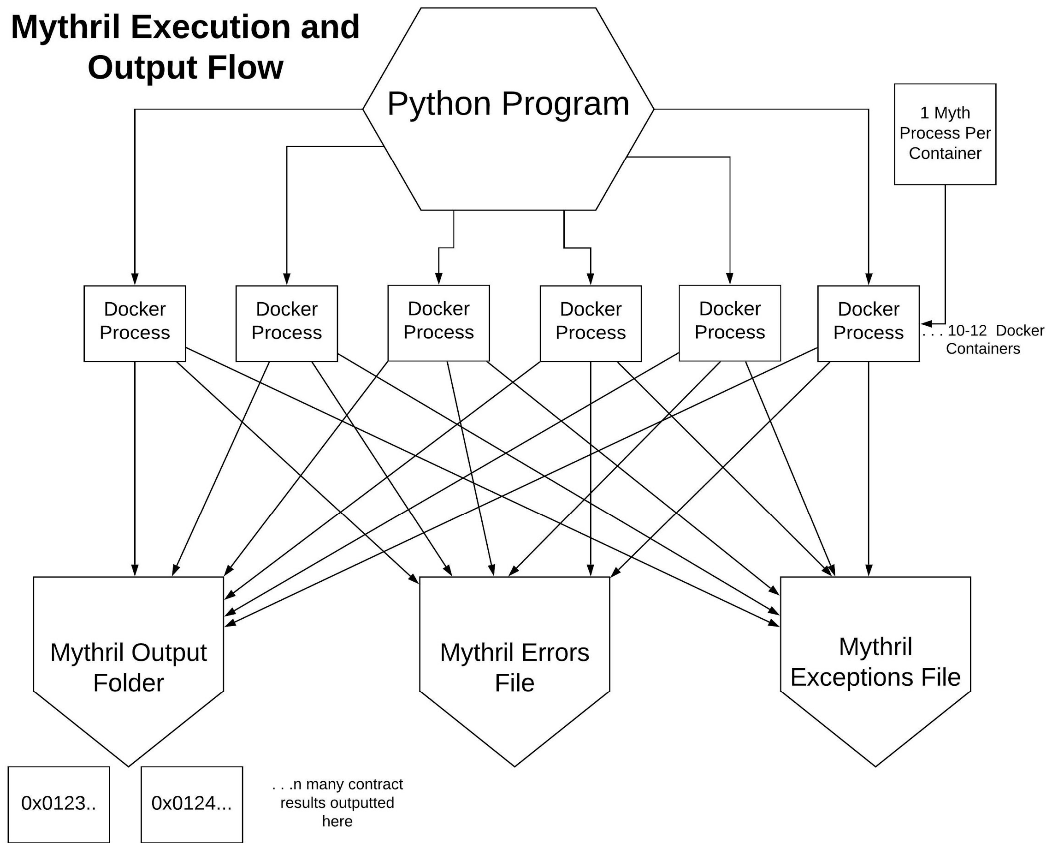

Figure 6 - High-Level Overview of Data Collection Program [26]

Each docker container waits for precisely one-hour before terminating the execution. This was important as a small subset of contracts never complete and thus would have caused the entire program to halt as the available random access memory (RAM) and main memory would fill with symbolic execution data and produce an out of memory (OOM) error.

3.4. Data Parsing, Storing, Retrieving

Once the output of multiple contracts was written out to files, this information was parsed to extract relevant data for the registry; specifically contract address, type of vulnerability, and severity of that vulnerability into a .csv file. A hand-coded parser was written to run through the output files, which contained millions of lines of output – including vulnerability type, severity, the name of contract to which each output belonged to, and the coverage obtained by the engine. This information was manipulated into a .csv file, which is an ideal format for uploading to a database or simply parsing to gain insight by row and/or column.

4. Results

The resulting dataset from the GETH node and program yielded all the contract addresses created from the 0 - 9 millionth block, though only a majority of the 0 - ~8.4 millionth block was analyzed for this research project. The results yielded here should be taken with a grain of salt as an amount of contract addresses weren't able to be completely scanned within this range as the project ran out of research credits, time to complete analysis, and time to look for more funding. In other words, this project is a proof of concept.

4.1. Number of Contract Addresses

The total number of contract addresses created, since the beginning of the Ethereum blockchain, up to the 9 millionth block, totaled **3,046,140.** This data was used by the symbolic execution engine, Mythril, which yielded results on what contracts were compromised. Below is a histogram distribution of how many contract addresses were found to be within each millionth block (inclusive, exclusive).

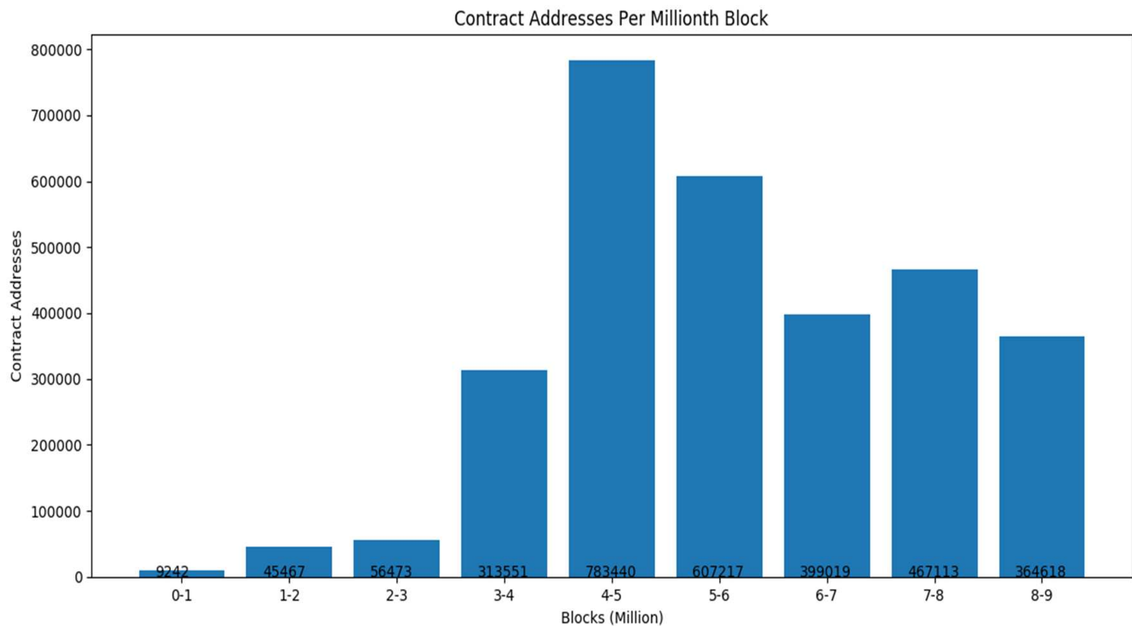

Figure 7 - Contract Addresses in 9 Million Blocks

One thing to keep in mind is that some addresses were a facsimile of others. For example, Etherscan.io has a portion of their website dedicated to listing these contracts [27]. Therefore, the amount of *unique* addresses with and without vulnerabilities is actually lower than the numbers given in this paper.

Furthermore, some contract addresses no longer contain any bytecode as they have been destroyed through a Selfdestruct method call or were created without any init code, so there is no deploy code either. As a result, this means that symbolic execution tools have no bytecode to execute on. The number of ***live*** contract addresses on the Ethereum blockchain from the 0-9 millionth block are **2,927,521**. Approximately 119,000 addresses have been

destroyed over the lifetime of the blockchain. The below is a graph of the live contract allocation per millionth block range:

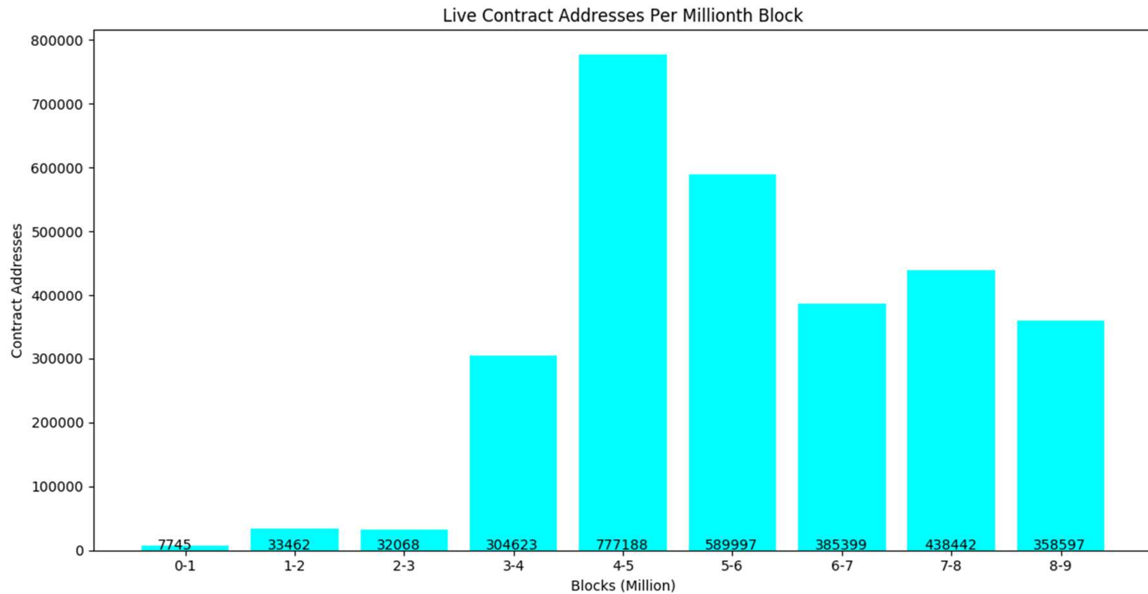

Figure 8 - Live Contract Addresses in 9 Million Blocks

4.2. Vulnerable Ethereum Contracts

The number of contracts with vulnerabilities on the Ethereum blockchain from the 0-~8.4 millionth block was **797,384 contracts**. This number is surprisingly large. As mentioned previously, there is a total of **2,927,521** live contracts up to the 9 millionth block. This means that more than 27% of Ethereum smart contracts have one or more vulnerabilities in them (*more,* since this research completed only a partial analysis up to the 8.4 millionth block). Furthermore, this percentage is higher still considering that Mythril could not execute on all addresses due to bugs in its engine or simply a lack of infinite compute time to process each contract beyond a 1-hour limit. Additionally, if more symbolic execution

engines were used on the dataset of addresses, then it is very likely that this percentage would be even higher as different symbolic execution engines would find different vulnerabilities.

4.3. Amount of Vulnerabilities

The number of vulnerabilities discovered during this project summed to the number **1,224,486**. This number does not reflect duplication, for example, when a contract may have repeated vulnerabilities of the same type within a function(s). Rather, the following data should be taken to mean that, for each number in a type of vulnerability, that a contract contained at least one vulnerability of that type (e.g., Integer Overflow >= 1). Thus, the number of ways a user may potentially exploit a vulnerable contract is one or more. Below

is the number of non-repeating vulnerabilities discovered during this research, by vulnerability type, in a graphical format:

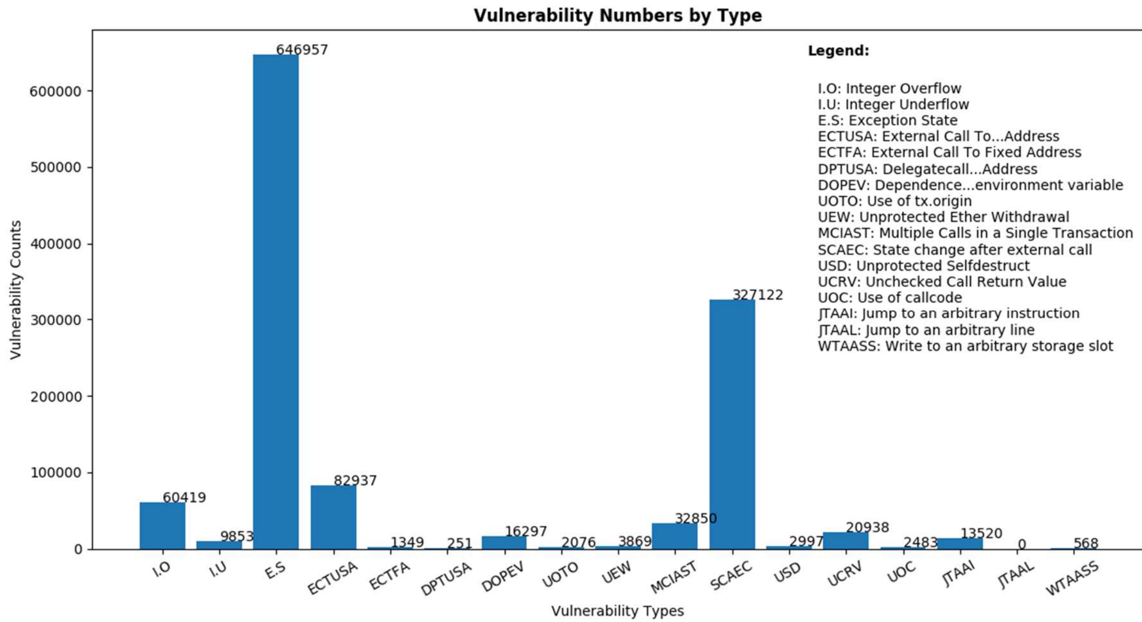

Figure 9 – Number of Vulnerabilities by Type

Seeing the data in this way helps us realize the potential severity of these vulnerabilities by using Mythril's severity ratings. The most common vulnerabilities are *Exception State* (ES; 646,957) and *State Change After External Call* (SCAEC; 327,122). An ES vulnerability has a severity rating of *low* according to Mythril. A SCAEC vulnerability has a severity rating of *low* or *medium* depending on if the external address is a user-supplied address (*medium severity*) or a hardcoded one by the developer (*low severity*). The results here indicate that the majority of vulnerabilities present in the contracts in this research are not likely to cause much damage to users as the majority of vulnerabilities are of a mostly low severity rating. However, even a few contracts with higher severity vulnerabilities could contain a large quantity of ETH and a single potential vulnerability could lead the way to

an exploit that could cause damages on multiple users of a contract in the millions, as history has shown. Furthermore, if this research continues and finishes analysis, this ratio may change, though it is unlikely that it would change drastically.

Another informative way to take our data is to analyze what blocks have the most vulnerabilities inside of them.

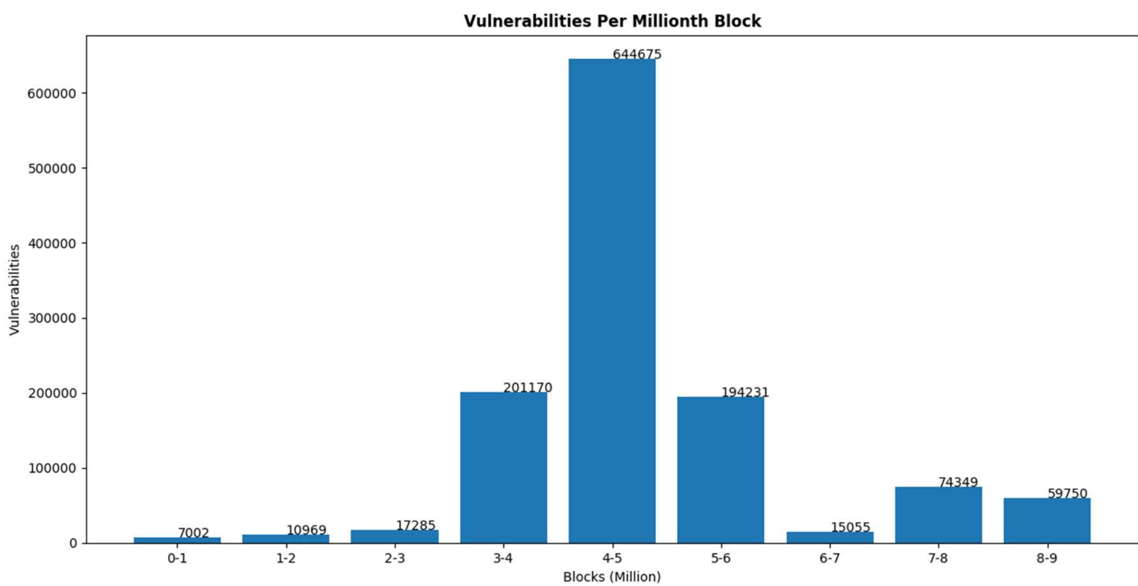

Figure 10 - Exploits Per Millionth Block

As can be seen, among the results obtained, most of the contracts that contain insecure code were created within the 4-5 millionth block. As time has progressed, most vulnerabilities have decreased in number. Though this is only a partial result, this could mean that there are factors at work that are making Ethereum contracts more secure over time. Examples include: the Solidity language becoming safer as time goes on, developers using symbolic

execution tools, etc. More research is needed to come to a conclusive analysis on the state of Ethereum vulnerabilities per millionth block.

4.4. Amount of Ether at Risk

'At risk' employs some hyperbole. The true amount of ether that could be taken out or destroyed forever through the manipulation of a vulnerability is much lower. However, if one defines 'At Risk' as meaning a contracts available ETH is potentially able to be siphoned off due to the nature of a contract having a vulnerability, then the ETH in contracts potentially at-risk totals ***2,580,565* ETH**. Or, in USD as of 01/19/2020, ***$430,128,605***.

One can see the amount of ETH potentially at risk per millionth block in the following figure:

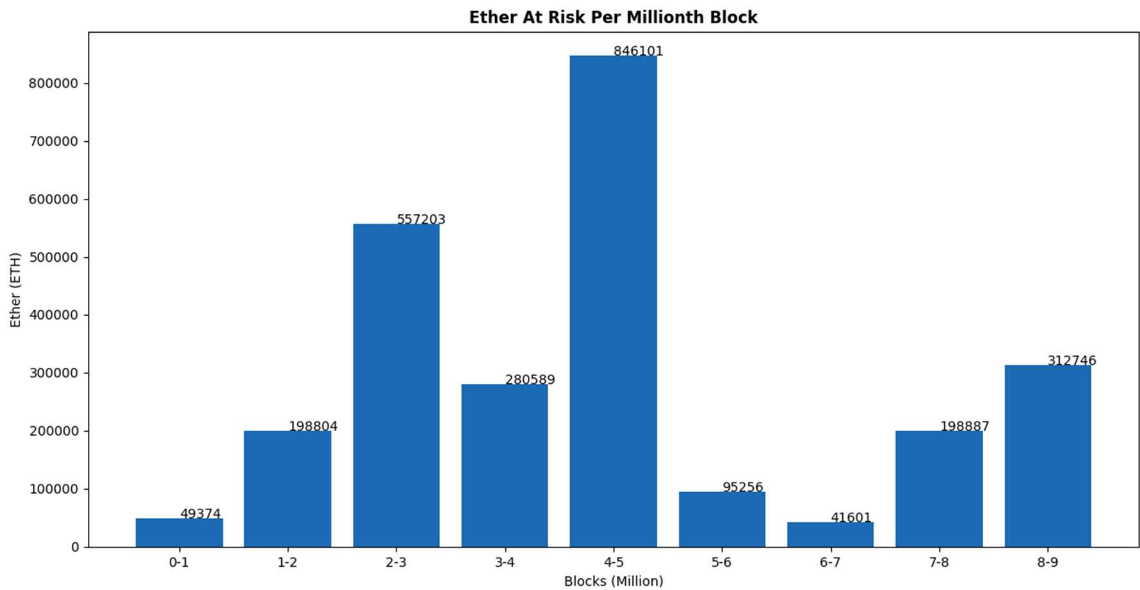

Figure 11 - ETH at Risk Per Millionth Block

4.5 Coverage Amount

The coverage that Mythril was able to obtain from each contract was also recorded and inserted into the data registry. As the symbolic execution was constrained to 1-hour for performance and cost reasons, this was necessary. Mythril also does not achieve 100% coverage sometimes due to bugs in its tooling or other reasons.

Oftentimes Mythril achieves 99.80% coverage, but this is in actuality 100% as the .20% coverage cannot be achieved since the instructions are merely sanity checks inserted by Solidity code to prevent out of bounds access and are not reachable with any input.

Overall, the average code coverage achieved for this proof of concept was **76%**. In the future, it may also be useful to instilling confidence in users to separate the amount of code coverage per millionth block. However, since this is only a proof-of-concept, this task will wait until a further point in time.

5. Discussion

Researchers Perez and Livshits ask an important question in their 2019 research article titled *Smart Contract Vulnerabilities: Does Anyone Care?*. As researchers discovered, pooling results from 5 different academic research articles, there were "at most 504 out of 21,270 contracts...subjected to exploits" (pg. 1). However, this was *only* "9,094 ETH (1 million USD)...0.30% of the 3 million ETH (350 million USD)" (pg. 1) [28].

These results need to be framed in a similar context to see if they are worth caring about. The study by Perez and Livshits can act as a sample in a larger population and be generalized to the larger subset discussed in this research, which encompasses a much larger number of contracts. If 504 out of every 21,270 contracts can be exploited -- roughly 2% -- how many contracts could be exploited out of 797,384 vulnerable contracts found by this project? 2% of 797,384 contracts is approximately 15,948 contracts. If 504 contracts housed 1 million USD, then that means, given the same ratio of contracts likely to be exploited (2%), 31 million USD (~15948/504) would likely be at risk in this dataset. Assuming the same USD:ETH (the value of ETH has risen since Perez and Livshits' research so would be a higher dollar amount), this would mean that, at minimum, 8% of the total Ethereum supply chain is likely to be exploited. Though not a giant percentage, it still is *not* a satisfying number in this context.

Furthermore, hackers DO care about Ethereum vulnerabilities; they just may be lacking in sophistication. For example, the internal transaction logs (the history of functions called

within a contract) can show multiple transactions trying to exploit the *Selfdestruct* vulnerability of an **empty contract** [29]. This is brute force, automatic, and bot-like behavior. But what if individual hackers devised a more complicated, targeted approach; one informed by a dataset such as the one presented in this thesis? They may have more luck, and the amount of exploitation may rise in the Ethereum blockchain.

5.1. Ethical concerns

With these results it is important to discuss a variety of issues. A chief ethical concern comes to mind: What if people use this type of information to exploit vulnerable contracts? This project is an attempt at being proactive with exposing and alerting users of vulnerabilities. Though malevolent parties may use this data as a stepping stone to taking advantage of a vulnerability, the fact is that these exploits would, in all likelihood, happen with enough passing of time as certain parties become more savvy and the use of symbolic execution engines or smart fuzzers (another method of detecting vulnerabilities) become part of the norm.

Due to the nature of blockchains decentralization and anonymization, a system for alerting contract owners and users of vulnerabilities is non-existent. Therefore, users cannot be alerted directly of vulnerabilities in their contract code. Furthermore, in attempting to abide by ethical guidelines, no vulnerabilities can be taken advantage of by researchers, even for the purpose of setting aside the ETH for owners to claim later.

However, since contract owners and users cannot be alerted directly, at least they can be knowledgeable if they make use of this dataset and, with that knowledge, may attempt to pull money out and/or close the contract. For the newest contracts that contain vulnerabilities, if developers and users utilize this service, they will not fill a vulnerable contract full of money and setup a malevolent user for future success.

5.2. Veracity of Results

A small subset of addresses never achieved output results due to bugs in Mythril. Furthermore, the API tier that Mythril uses limits the number of requests one can make. In some cases, this prevents the execution on some addresses with large bytecode sizes, and it is unknown when or if these bugs will ever be fixed. In other cases, bug fixes are in the works, but have not been pushed out yet to the general public, so were unable to be included in the dataset, although these contracts were documented so as to be included at a later date.

There were also contracts for which no symbolic execution will likely ever return results. For one, Mythril is not able to complete execution on 100% of a contract's code for 100% of all Ethereum addresses. As a result, contracts deemed to have no vulnerability, may very well indeed have one or more uncaught vulnerabilities. Unless technology improves in some dramatic way, there is no way to deem a contract *truly* safe and the results in this research conclusively true for all output data. This is part of the reason why this dataset

includes only contracts deemed unsafe, rather than explicitly declaring other contracts safe. If a contract is not within this dataset, a user should not assume it is free of vulnerabilities or safe.

Furthermore, it should be noted that the results here are the opinion of one engine and, as it is sometimes with medical opinions, it is always best practice to get a second opinion before deciding whether to use a contract. Multiple perspectives from engines is, hopefully, a goal that this project can enable in some future point in time.

6. Exploitation

The data being released explicitly lists contracts that contain vulnerabilities, which could potentially lead to exploitation of Smart Contracts. As mentioned before, an ABI is needed to interact with the source code. However, even though there are a variety of ways to see if a contract's code/ABI has been *verified/published* [30], [31], [32], [33], [34], [34] not everyone releases their code or ABI to the public. The following sub-sections describe how an attacker may gain access to a contract's vulnerable function(s) and potentially exploit it without a full ABI.

6.1. Interacting with Contracts Via Partial ABI

A partial ABI may be all that is necessary to leverage a vulnerability(ies) inside a contract. On a small scale, anyone may interact successfully with a contract that is created on Ethereum with a minimal ABI specific to that function. Take, for example, this contract:

```
1 pragma solidity ^0.5.10;
2
3 contract MyContract {
4
5 string name = "Vitalik";
6
7 function getString() public view returns (string memory) {
8     return name;
```

```
 9 }
10
11 function close() public {
12   selfdestruct(msg.sender);  // `owner` is the owners address
13 }
14 }
```

Figure 12 Pt. 1- Exploit Contract With Partial ABI

This particular contract has a programmed *Selfdestruct* vulnerability inside (line 11-13). With a full source code, a full contract ABI can be derived and access any function inside this contract, including the vulnerable function *close()*. But, as mentioned, a user may not have access to a full ABI. However, with a partial source code, a partial ABI can also be used to interact with a contract function(s). Simply by knowing this generic self destruct, and often used function signature, one can write a minimal contract interface with a single empty function and receive an ABI that is able to interact with that contract's vulnerable function! For example, from the below source code a partial ABI can be obtained to interact with the *close* function:

```
1 pragma solidity ^0.5.10;
2 contract MyContract {
3 Function close() public {
4 }
```

```
5 }
```

Figure 13 Pt. 2 - Exploit Contract with Partial ABI

Though this is a small example, this example demonstrates that anyone can call an entire function without any body of that function, within any contract, given that they have the signature of the vulnerable function or are able to guess that functions signature. If there are no extra safety checks or logic within the body of a vulnerable function, then it is very likely that the vulnerability being tried upon will result in a successful payout for an attacker.

Why is this important to the dataset this research has generated? This feature of solidity is a tool in the attacker's arsenal. If an attacker knows a particular function has a vulnerability (which this dataset reveals) and then also has the source code or partial ABI of a contract, then, so long as that function is public, that attacker has unfettered access to call the contract function at will. Any attacker may use this dataset to go through the vulnerabilities listed here, discover which function it is referring to (perhaps by running Mythril on that specific function for more detailed output) and write generic contract interfaces to try and interact with contracts (for example, a Selfdestruct vulnerability). If the programmer of a contract has not done a good job in using checks in the body of that function (e.g., to check if the caller of the contract is the owner), then whatever business logic a developer intended that function to have for a selected party (e.g., himself) may be opened up to anyone.

6.2. Decompiling Contracts to Exploit

But how does one get a function signature of a function in which the vulnerability is not generic as a self destruct or the source code is not public? Novice users and developers of Ethereum may be under the impression that a contract is secure as long as the source code or ABI is not public to anyone. This is an old thought known as “Security Through Obscurity”. Many experts agree, however, that this is a security method that should only ever be used in tandem with other security methods.

In fact, it is still possible to discover a contract’s (vulnerable) function(s) signature(s) with the use of decompilers [35], [36]. Decompilers return the machine-readable bytecode to partially human readable source code. Though not 100% human readable, it may reconstruct enough of the source code to reconstruct a vulnerable function’s signature, which in turn may give a partial ABI to interact with a contract whose ABI and/or source code is not public.

7. The Data Registry: haveibeenexploited

www.haveibeenpwned.com is a website that allows users to see if their information has been leaked in a security breach by entering any email address into a simple search bar. A prototype website of a similar concept for Ethereum was created through inspiration of this popular idea, located at the address *www.haveibeenexploited.com*. This website acts as a proof-of-concept registry where users can type in a contract's address into a search bar and determine if a contract is safe before use.

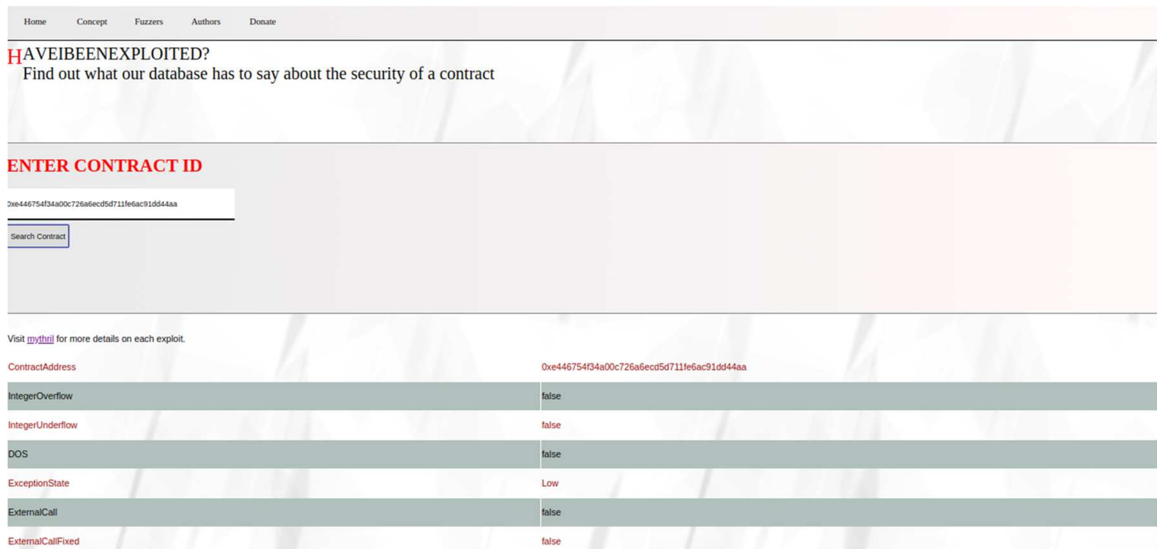

Figure 14 – www.haveibeenexploited.com

A small handful of technologies were used to implement this website. haveibeenexploited.com is hosted on a small micro-f1 instance on Google Compute Engine for $0/day, since its size qualifies it for the *always free tier*. The website frontend was

entirely supported through a *React* app that enabled display of the static website pages needed to explain and present the concept. The popular language *Go* handles incoming server requests that query a MySQL server that runs in the background and contains the list of contract addresses and their vulnerabilities, collected in this study.

This website needs a massive overhaul in order to get users of the Ethereum chain to adopt it, as can be seen in the previous figure. Design is a foremost concern as it is very minimal, but also other contingency plans need to be made if the website's popularity expands such as upgrading the machine type it is running on. Also, as other symbolic execution engines and/or smart fuzzers are added in, the website needs to be modified to reflect multiple tool's results.

As it is now, a user may query a contract at the website and see if it contains vulnerabilities. This original implementation was by design, as it limits the ability for individuals to access the entirety of the database and see all contracts which have vulnerabilities. It was thought that this way, honest users who wish to check on the safety of contracts they are concerned with may receive this information. On the other hand, those who wish to obtain a large listing of insecure contract addresses for exploitation purposes may not do so, short of the possibility of hacking into the database holding the vulnerable contracts.

This implementation may change as it becomes clear how many vulnerabilities are actually able to be exploited. It may also be beneficial for contracts to be released to the public so

more hobbyists can explore hacking Smart Contracts with vulnerabilities baked into them. This kind of ability to view vulnerable contracts in the wild does not widely exist. The dataset uses in this research is currently active on the website.

8. Future Work

Future goals include finishing the symbolic execution process on the remaining contract addresses on the Ethereum blockchain. Another goal would be including engines in addition to Mythril and repeating the process. This would be more informative and beneficial for users of smart contracts to gain multiple opinions on the safety of smart contracts from multiple symbolic execution engines [37], [38]. Below are more ideas besides these that will potentially be explored.

8.1. Automating Analysis

A further implementation could be the automation of smart contract analysis, parsing, and updates to a decentralized database. The process of contract obtainment, symbolic execution, and parsing can be automated. This would be trivial to implement and maintain so long as stable versions of symbolic execution engines were used (to abolish the need to update a parsing agent with each new version) and runaway processes were handled appropriately. The benefit this would provide would be close to real-time feedback on contracts by a variety of symbolic execution tools. This is an idea similar to Consumer Reports [39] or what VirusTotal [40] provides to its users for malware analysis and detection.

www.contract-library.com/ is a forerunner in automated analysis and contains a variety of features. New contracts are automatically scanned, and the results uploaded to the website. The UI allows for selecting contracts by vulnerability type. The website has a disassembler

baked into it. This data and these functionalities are useful for security researchers as well as newcomers wanting to learn more of what vulnerabilities look like in the wild. Using this website as an example or merging this research's results with the website would create a powerful library of insecure addresses. Automating analysis has not been implemented in this project until the interest grows and/or funding appears for this add on.

8.2. Rating System

As a part of this project, a rating system is also proposed. Mythril already uses *severity* as a keyword to indicate a vulnerability's potential level of harm for a contract. This rating system would either be based off number of vulnerabilities and/or the severity levels of the respective vulnerabilities. As more symbolic execution engines are added to the mix, the rating system may be updated to reflect a safety level from each symbolic execution engine (e.g., Mythril — low severity; [symbolic_engine2] - medium severity), with perhaps a total score which takes into account multiple results from varying engines.

8.3. Mythril Results as Guide for Further Execution

As mentioned previously, Mythril provides information about possible vulnerabilities and is not proficient at finding states where an exploit may exist. However, there are other symbolic execution engines that may be able to find exploits more successfully with hints from Mythril as to what vulnerabilities exist. Manticore [37], for example, works best when specifying the exploit that it is looking for. By configuring Manticore to detect a specific

exploit(s) (e.g., Integer Overflow), Manticore is able to cover a larger section of the code for possible states where an exploit may be achieved through a specific vulnerability.

Furthermore, since Mythril, by default, keeps track of state with a maximum of two separate transactions, certain contracts could be run again with a higher number of multi-transactional settings. This would reveal harder to find vulnerabilities or exploits. The criteria for running this secondary in-depth analysis on contracts could be guided by the amount of ETH inside each contract and/or specific vulnerabilities. In addition, other symbolic execution engines could be used with more rigorous settings to reveal these types of tough to find vulnerabilities or exploits.

8.4. Registry for Other Blockchains, Registry Built-in to Blockchains

This type of research and idea for a registry is not germane to a single blockchain or language such as Solidity. For example, Mythril may run on any EVM bytecode from whichever high-level language it was once constructed in. Furthermore, symbolic execution and other static and dynamic analysis tools can be used on other blockchains which have DApps and a similar registry may be made.

The research here may even be done by nodes in a network and reported to a central registry. This would indeed give a level of security to the blockchain that does not exist currently as each contract would be vetted automatically by these engines. However,

moving a project forward such as this would require structural changes in a blockchain (such as Ethereum) and agreement in leading developers and the majority of users.

9. Conclusion

Ethereum is one example of an increasing attention to blockchain and decentralized applications to replace and/or improve upon current financial infrastructure. But with each new technological invention comes security concerns; it is a truism that there always exists a group of individuals who wish to exploit vulnerable loopholes in new technologies.

Smart Contract vulnerabilities are not rare either, as evidenced by this thesis' results, even though their consequences of them may be unknown. These vulnerabilities pose a different risk than do traditional software systems bugs. Mainly, that these bugs can carry financial consequences and, once deployed to the network, are immutable programs which live on the network unless the bytecode is destroyed. Until such tools as symbolic execution engines become more commonplace, or another paradigm is invented to declare a contract reasonably secure, there will always be doubt as to whether a Smart Contracts or the Ethereum blockchain is truly fit for financial use cases.

This thesis is an attempt to bring attention to the vulnerabilities of this technology and offer a model similar to a Consumer Reports or VirusTotal for users, developers, and Ethereum enthusiasts. It is the hope of this research that the results laid out here increase further improvement and, in response, adoption of Ethereum, and blockchain generally. If users feel that new technology is safe and trustworthy, only then will they give up older technology in favor of the new. By analyzing the state of Ethereum and providing a system that creates transparency in specific contracts which contain vulnerabilities, the number of

users of the network will likely increase and adoption of Ethereum for financial use cases will grow.

[1] B. Anderson, "The Most In-Demand Hard and Soft Skills of 2020," 9 Jan 2020. [Online]. Available: https://business.linkedin.com/talent-solutions/blog/trends-and-research/2020/most-in-demand-hard-and-soft-skills. [Accessed 23 02 2020].

[2] A. Groce, "246 Findings From Our Smart Contract Audits: An Executive Summary," 8 08 2019. [Online]. Available: https://blog.trailofbits.com/2019/08/08/246-findings-from-our-smart-contract-audits-an-executive-summary/.

[3] Ethervm, 10 Feb 2019. [Online]. Available: https://ethervm.io/. [Accessed 01 01 2020].

[4] Solidity, "Solidity: ReadTheDocs," 10 Feb 2019. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.3/abi-spec.html#examples. [Accessed 01 01 2020].

[5] A. Manning, "Solidity Security: Comprehensive List of Known Attack Vectors and Common Anti-Patterns," 30 05 2018. [Online]. Available: https://github.com/sigp/solidity-security-blog. [Accessed 01 01 2020].

[6] R. Stevens, "Fairwin: The $125 million alleged Ponzi scheme eating Ethereum," 3 October 2019. [Online]. Available: https://finance.yahoo.com/news/fairwin-125-million-ponzi-scheme-143621138.html. [Accessed 01 01 2020].

[7] S. Zheng, "Ethereum developers find 'critical vulnerabilities' in 'Ponzi scheme' FairWin," 27 09 2019. [Online]. Available: https://www.theblockcrypto.com/linked/41307/ethereum-developers-find-critical-vulnerabilities-in-ponzi-scheme-fairwin. [Accessed 01 01 2020].

[8] P. Daian, "Analysis of the DAO exploit," 18 06 2016. [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/. [Accessed 01 01 2020].

[9] W.-c. Feng, 2019. [Online]. Available: https://www.thefengs.com/wuchang/courses/cs410b/slides/05a_SymbolicExecution.pdf. [Accessed 01 02 2020].

[10] ConsenSys, "Mythril," 2019. [Online]. Available: https://github.com/ConsenSys/mythril. [Accessed 01 02 2020].

[11] B. Mueller, 29 05 2018. [Online]. Available: https://github.com/ConsenSys/mythril/wiki/LASER-Module. [Accessed 01 02 2020].

[12] Microsoft, 01 01 2019. [Online]. Available: https://github.com/Z3Prover/z3. [Accessed 01 02 2020].

[13] B. Mueller, 2018. [Online]. Available: https://github.com/b-mueller/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf.

[14] B. Mueller, "Practical Smart Contract Security Analysis and Exploitation Part 1," 13 November 2018. [Online]. Available: https://hackernoon.com/practical-smart-contract-security-analysis-and-exploitation-part-1-6c2f2320b0c. [Accessed 01 02 2020].

[15] J. Feist, G. Gustavo and G. Alex, "Slither: A Static Analysis Framework for Smart Contracts," *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB),* 2019.

[16] B. Jiang, L. Y. and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection.," in *The 33rd ACM/IEEE International Conference on Automated Software Engineering*, New York, 2018.

[17] L. Loi, C. Duc-Hiep, O. Hrishi, S. Prateek and H. Aquinas, "Making Smart Contracts Smarter," in *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, New York, 2016.

[18] N. Atzei, M. Bartoletti and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts," 2017.

[19] D. Luca, "Karl," 01 01 2019. [Online]. Available: https://github.com/cleanunicorn/karl. [Accessed 01 01 2020].

[20] J. Krupp and R. Christian, "teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts," in *USENIX Security Symposium*, 2018.

[21] D. Luca, "Theo," 1 01 2019. [Online]. Available: https://github.com/cleanunicorn/theo. [Accessed 01 01 2019].

[22] B. Mueller, "Scrooge McEtherface," 2019. [Online]. Available: https://github.com/b-mueller/scrooge-mcetherface/. [Accessed 01 02 2020].

[23] "SWC Registry," 2019. [Online]. Available: https://swcregistry.io/. [Accessed 01 02 2020].

[24] "go-ethereum," 2019. [Online]. Available: https://github.com/ethereum/go-ethereum/wiki/geth. [Accessed 01 01 2020].

[25] T. Dulcet, "Send-Msg-CLI," 31 01 2020. [Online]. Available: https://github.com/tdulcet/Send-Msg-CLI. [Accessed 31 01 2020].

[26] "LucidChart," 15 01 2020. [Online]. Available: https://www.lucidchart.com/. [Accessed 15 01 2020].

[27] Etherscan, "Similar Contracts," 2019. [Online]. Available: https://etherscan.io/find-similar-contracts. [Accessed 01 01 2020].

[28] D. a. L. B. Perez, "Smart Contract Vulnerabilities: Does Anyone Care?," *arXiv,* 18 02 2019.

[29] Etherscan, "0xbF9D83a2019a3Daad8E9D37a577ff02E6592463b," [Online]. Available: https://etherscan.io/txsInternal?a=0xbF9D83a2019a3Daad8E9D37a577ff02E6592463b&p=1. [Accessed 01 02 2020].

[30] Swarm, "Swarm," [Online]. Available: https://swarm-guide.readthedocs.io/en/latest/introduction.html. [Accessed 01 02 2020].

[31] IPFS, "IPFS," [Online]. Available: https://gateway.ipfs.io/ipfs/QmTeW79w7QQ6Npa3b1d5tANreCDxF2iDaAPsDvW6KtLmfB/. [Accessed 01 01 2020].

[32] Etherscan. [Online]. Available: https://www.etherscan.io. [Accessed 01 01 2020].

[33] Etherchain, "Etherchain," [Online]. Available: Etherchain.org. [Accessed 01 01 2020].

[34] Verification.komputing. [Online]. Available: http://verification.komputing.org/. [Accessed 01 02 2020].

[35] EVEEM. [Online]. Available: https://www.eveem.org/. [Accessed 01 02 2020].

[36] Contract-Library, "Decompiler," [Online]. Available: https://contract-library.com. [Accessed 01` 02 2020].

[37] Manticore, "Manticore," [Online]. Available: https://github.com/trailofbits/manticore. [Accessed 01 02 2020].

[38] Echidna, "Echidna," [Online]. Available: https://github.com/crytic/echidna. [Accessed 01 02 2020].

[39] "Consumer Reports," [Online]. Available: https://www.consumerreports.org/cro/index.htm. [Accessed 01 02 2020].

[40] "VirusTotal," [Online]. Available: https://www.virustotal.com/. [Accessed 15 01 2020].