Reverse Engineering Cryptocurrency Smart Contracts

1st Davis Beilue
Computer Science and Engineering
Texas A&M University
davis.beilue@tamu.edu

2nd Liam Bessell
Computer Science and Engineering
Texas A&M University
lbessell@tamu.edu

3rd Matthew Ho
Computer Science and Engineering
Texas A&M University
mattho@tamu.edu

***Abstract*—In this project we will reverse engineer Ethereum smart contracts, programs that run on the Ethereum platform. We aim to analyze the overall binary structure of smart contracts and identify different security vulnerabilities in them. Upon completion of the project we hope to have a better understanding of reverse engineering and smart contracts as a whole.**

***Index Terms*—Blockchain, Smart contract, Reverse engineering, Cryptocurrency**

I. Introduction

Bitcoin launched in 2009 and was the first successful decentralized, digital currency - or cryptocurrency. Now there are thousands of cryptocurrencies, and while each has their own unique properties, they all rely on blockchain technology. Simply put, a blockchain is a growing list of records linked together via cryptography. It is impossible to modify a block without changing its subsequent blocks. Thus on a distributed network, one node can't change a block without the others knowing about it and consenting to the change. This creates a secure and decentralized database system which make up the benefits of cryptocurrencies.

In this project we will be focusing on the Ethereum network, which is a programmable blockchain that runs on the Ether cryptocurrency. One of the key differences between Ethereum and Bitcoin is that the Ethereum blockchain can hold and execute programs called smart contracts. A smart contract is simply a program that runs on the Ethereum blockchain. It is useful to think of a smart contract like a vending machine; it takes some input in the form of Ethereum tokens and a selection, then outputs or fulfills the product or service.

Since smart contracts exist on the Ethereum blockchain, their bytecode is publicly accessible and can be reverse engineered [1]. There are a number of reasons for wanting to reverse engineer a smart contract. First and foremost is ensuring that it is actually doing what is advertised. It is trivial to ensure that given a certain input, the advertised output is fulfilled. However, it could be argued that reverse engineering the contract before a transaction takes place would ensure the expected output is given. There is further motivation though: what if the smart contract does something unexpected behind the scenes?

It is important to note that many smart contract authors do, in fact, publish the source code of their contract along with the bytecode. This has the advantage of allowing users to be fully confident in their interaction, but comes with the disadvantage of allowing competitors to view your work. Those that do not have such transparancy are designated as "opaque" or "proprietary" contracts [1].

II. Goals

Our first goal is to analyze high-level code that we will acquire by using the Erays [1] software to reverse engineer Ethereum bytecodes on the blockchain. The purpose of our analysis shall be to examine the structure of said high-level code to determine what, if any, vulnerabilities authors could feasibly leave in their contracts. Whether intentional or not, such vulnerabilities could have devastating effects for Ethereum users should they interact with a malicious contract.

Similarly, we will examine the structure and patterns of Ethereum bytecodes themselves. By comparing both the bytecodes and high-level codes of different contract instances, we hope to determine if any contract traits can be inferred from the bytecodes themselves without any sort of reverse engineering process being performed. This could allow us to draw conclusions about how much knowledge Ethereum users need to have in order to feel confident that a given contract on the blockchain is indeed performing the advertised actions.

As our final goal, we plan to gain insight on current initiatives to keep Ethereum smart contracts protected from being reverse engineered. We would like to discuss the advantages and disadvantages of this idea (from the perspectives of both contract authors and customers), as well as present the feasibility of such action based on current research [2].

III. Topic

We will now introduce the main topics of our research, which include the Ethereum Virtual Machine, the process of reverse engineering smart contracts, and the purposes of

doing so.

A. The Ethereum Virtual Machine

As one of the Ethereum Virtual Machine (EVM) authors writes, "[the EVM *exists*] as one single entity maintained by thousands of connected computers running an Ethereum client" [3]. While other blockchains such as Bitcoin are thought of as a distributed ledger, the Ethereum blockchain has to keep track of a machine state as well as accounts and balances. This is where the EVM comes in, which specifies rules for this machine state and how it can change. The EVM is necessary for the Ethereum blockchain to be programmable - to run smart contracts.

While the EVM does keep track of a machine state, the authors describe the transactions to be more broadly changing a "world state" [3]. As one cryptographic blog puts it, "the World State of Ethereum consists of a mapping of 160-bit address identifiers and account [states]" [4]. Thus, each set of transactions takes the entire Ethereum landscape from one "world state" to another (seen in Figure 1), further implying that this landscape as a whole can actually be represented "as a stack of transactions" [5]: if one were to pop transactions off of the stack, thus "undoing" them, one would theoretically be able to represent the state of Ethereum at any desired point in time.

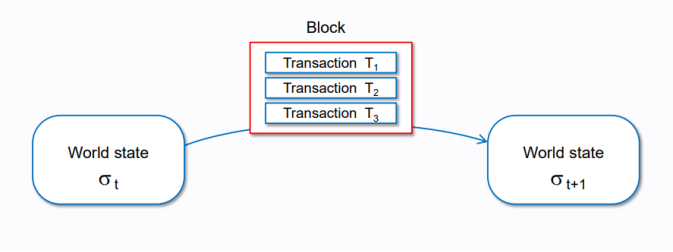

Fig. 1. Representation of one world state being mapped to another through a transaction [5].

B. Smart Contracts

If the EVM is the heart of the Ethereum body, then smart contracts are the veins; they regulate the spending of Ethereum coins by customers, and ensure those customers get exactly what they asked for. They are the embodiment of the state machine aspect of Ethereum: given a certain input, they will provide a deterministic output.

To deploy or interact with a smart contract, users have to pay a "Gas cost," which is essentially using Ethereum tokens to pay for the computational power used to perform the desired action with the given contract. Our investigation will not require us to interact with the contracts in such a direct way, so this cost will not be a concern of ours.

As we have mentioned, there are many reasons to reverse engineer a smart contract on the Ethereum blockchain. The team believes the main advantage to doing so lies in the security aspect: if a customer could possibly be left vulnerable to a malicious vendor (or, perhaps, a clueless one), then there is much at risk, especially at higher levels of trading. This malice could be expressed in a multitude of ways: remember that smart contracts are deterministic, thus customers can, with great confidence, expect what they will receive from a given contract. If this were to ever not be the case, it would be desirable for there to be a path to ensure that all contracts can be efficiently reversed and confirmed. Another possible path to harm would be in some form of code vulnerability that exists in the contract itself: perhaps the author implemented some sort of overflow attack, or something of the like, that could harm the data of their customers.

Regardless of our finding any sort of vulnerabilities (which, according to Hildenbrant et al. [6], are very real and dangerous), we believe this study will be a valuable look into how to reverse engineer binaries of a unique language and architecture.

IV. Methods of Investigation

We will conduct our investigation on this topic by first setting criteria for smart contracts to ensure they are relevant to our research. Then, a set of Ethereum smart contract bytecodes which fit our criteria will be acquired from the public blockchain. These bytecodes will be compared to find any patterns within its structure which may help indicate its traits and purposes. Additionally, each bytecodes' source code will be analyzed and compared to gain a deeper understanding on each contracts' structure. To accomplish this, these contract bytecodes will be decompiled into high level code with the use of tools and observed for any other noticeable patterns which may exist.

Tools exist to investigate the Ethereum blockchain and decompile smart contracts. We will examine some of these in more detail in section V, but broadly speaking they will allow us to view deployed smart contracts and decompile their bytecode into a high level language. Many smart contracts also have their source code publicly available on this platform, thus allowing use to the directly compare the decompiled code to the source code.

Figure 2 shows an example of a smart contract's source code and the code decompiled from its deployed bytecode. In this small function there are already many interesting insights and questions. First, the the function *to_little_endian_64* has a different name in the decompiled code. This is because it is a private function to the contract, it is not accessible to those interacting with the contract. By contrast *get_deposit_count* retains its name, because it is apart of the contract's interface. Another interesting part of the code is the *storage* access.

Smart contracts store persistent data in *storage*, so we can gather from looking at this that *deposit_count* is some piece of data that needs to be persistent between calls to the contract. There are also portions of the decompiled code that are not yet clear to us. For example, why is it that we lose the variable types, and what do *var0* and *var1* do?

```
// Source Code
function get_deposit_count() override external view returns (bytes memory) {
    return to_little_endian_64(uint64(deposit_count));
}

// Decompiled Code from EVM Bytecode via ethervm.io
function get_deposit_count() returns (var r0) {
    var var0 = 0x60;
    var var1 = 0x10c2;
    var var2 = storage[0x20];
    return func_14BA(var2);
}
```

Fig. 2. Side-by-side comparison of source code from a smart contract to the code decompiled from its bytecode. Credit: Eth2 Deposit Contract

Figure 3 displays the translation from EVM bytecode to human readable opcode. The EVM is a stack-based machine, akin to other virtual machines like the JVM. This means that operations are coupled with the stack, both getting values and storing results onto it. An obvious exception to this rule in the EVM *PUSH* opcode, which is unique for the fact that it must have a specified operand. In other words one can't just call *PUSH*, it must be *PUSH 0x**. It should be noted that bytecode to opcode has a 1:1 translation, much like assembly and binary. The ability to directly translate bytecode to opcode could be integral in our reverse engineering efforts. As we've learned, all tools have limitations. This gives us the ability to directly verify what is happening under-the-hood without purely relying on tools others have developed.

```
# First 11 bytes of the ETH2 Stake Contract bytecode
0x6080604052348015610010...

0x6080: PUSH 0x80               # Push 0x80 onto the stack
0x6040: PUSH 0x60               # Push 0x60 onto the stack
0x52  : MSTORE                  # memory[0x60] = 0x80 & 0xFF
0x34  : CALLVALUE               # Push the message funds in Wei onto the stack
0x80  : DUP1                    # Clone and the last value (message funds) onto the stack
0x15  : ISZERO                  # Is the top value equal to 0? i.e. message funds == 0
0x6100: PUSH 0x00               # Push 0x00 onto the stack
0x10  : LT                      # Is the top value less than the second to top value?
```

Fig. 3. Translation from EVM bytecode to human readable opcode. Credit: Eth2 Deposit Contract

V. Investigation Focus

This investigation will focus on finding vulnerabilities or traits within Ethereum smart contract bytecodes and source codes to determine whether reverse engineering an Ethereum contract is truly required for Ethereum users' assurance that the contract does what it is advertised to do. To decompile smart contract bytecodes, an open-source software "Erays" [1] will be used. Another tool called etherscan.io will be used to look up various attributes of deployed contracts and view their source code if it's been uploaded.

VI. Members' Roles

The following is the reiteration of the list of our team members, as well as what role they will mainly serve throughout the course of our project.

A. Davis Beilue

Research into smart contract mechanics and analyzing high-level code structure. Mainly responsible for presentation video.

B. Liam Bessell

Research into the Ethereum platform, using reverse engineering tools (e.g. decompiler), examining EVM bytecode.

C. Matthew Ho

Research into legality issues and data management. Mainly responsible for PowerPoint presentation.

VII. Data Plan

Ethereum was founded by Vitalik Buterin on July, 2015. The Erays software was developed and published by Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey in "Erays: Reverse Engineering Ethereum's Opaque Smart Contracts" [1] on 2018. The contributors of this particular github respository which contains a copy of the Erays software [7] we will utilize are James Levy and Deepak Kumar. This Erays software repository is licensed under MIT license [7], which grants us permission to private use and modification of any kind. Countless Ethereum smart contracts created by various individuals exist in the blockchain, and most contracts have an unlicense, or a license with no conditions and dedicates works to the public domain. For this reason, having a permissive license will be an important criteria for our selection of smart contracts. Thus, we will only focus on smart contracts under a license that grants us permission of free use.

The technical goal of this project is to analyze and study Ethereum smart contracts and to determine whether the contracts themselves have the possibility to cause adverse affects to its users through intentional or unintentional vulnerabilities. Learning about this topic will help gain knowledge that can be used in the future to ensure users' safety when performing transactions through Ethereum smart contracts.

Before dissemination, we will disclose any vulnerabilities that we find to our academic supervisor and disseminate only after approval is given by said supervisor. We will only include findings which are approved and nonsensitive

in our report to avoid any negative impacts to the vendor. Additionally, should any critical vulnerability found, we will inform the supervisor and the vendor of such vulnerabilities as soon as possible.

VIII. Demo Plan

Our final demonstration will consist, first and foremost, of meaningful and appropriate results gathered through reverse engineering Ethereum smart contracts. We will also give an overview of blockchain technology and the Ethereum platform. We plan on presenting said results and overview with the use of a PowerPoint presentation and a demonstrative video that will lay out our process of both reversing and data gathering.

The following subsections further detail our plans for each of these vessels.

A. PowerPoint Presentation

We will begin the presentation with some brief background information about Ethereum which are necessary to understand its process and inner workings. This will include explanations on the concept of decentralized currencies and how blockchains make them possible. Once sufficient background information on Ethereum is presented, we will lay out our investigation process on this topic by showing the smart contracts' bytecodes that we have decompiled along with comparisons. To finalize, we will present our findings that have been approved by our supervisor for dissemination, as well as draw conclusions on the topic based on our findings.

B. Demo Video

We will choose a smart contract and go through the process of reversing it from bytecode to a high level language. We will utilize both tools (e.g. decompilers) and examine the bytecode and opcode manually. With the original source code available, we will compare both the opcode and reversed high level code to the source code. We will also note common patterns of EVM bytecode and inner-workings of the EVM.

IX. Results and Analysis

We will now present our results and findings, along with relevant analysis, after utilizing the given methods to pursue our research goals.

A. Erays Analysis

In order to fully understand what Erays was accomplishing, we used its reversing capabilities on non-opaque contracts. What we found is that the output from Erays is closer to a detailed pseudocode than a true representation of the flow and structure of the original source code. This is partially due to the method in which the authors have chosen to reverse the bytecode, where the stack-based operations of the Ethereum opcode are rewritten as register-based operations.

Now to display an example of Erays pseudocode in a side-by-side with Solidity source code. This example is taken from the GeneScience contract found at https://tinyurl.com/RE451. Note that Etherium generates a separate instance of pseudocode (an instance being a unique .pdf file) for each function it detects during the reversing process, so the following images do not represent a contract in its entirety:

```
function _sliceNumber(uint256 _n, uint256 _nbits, uint256 _offset) private pure returns (uint256) {
    // mask is made by shifting left an offset number of times
    uint256 mask = uint256((2**_nbits) - 1) << _offset;
    // AND n with mask, and trim to max of _nbits bits
    return uint256((_n & mask) >> _offset);
}
```

Fig. 4. Source code from an example contract. Credit: GeneScience contract at https://tinyurl.com/RE451

```
args $s20 $s19 $s18 $s17

rets $s17

0x668
--------
$t = $s19
$s19 = 0x2 ** $s20
$s17 = (($s19 * ((0x2 ** $t) - 0x1)) & $s18) / $s19
intret()
```

Fig. 5. Pseudocode obtained from Erays using the bytecode generated from the sliceNumber function in the GeneScience contract [1].

Our initial analysis suggests that the pseudocode generated by Erays is quite unique from the original source code in terms of structure. The conversion of stack-based operations to register-based operations, combined with the use of more optimized and higher-level opcodes by the authors of Erays during the reversing process likely contributes to this [1]. Still, we are able to match the source code to the pseudocode based on the operations being done, as well as the return value type.

This is further illustrated by another example retrieved from a different function in the GeneScience contract:

```
function encode(uint8[] _traits) public pure returns (uint256 _genes) {
    _genes = 0;
    for(uint256 i = 0; i < 48; i++) {
        _genes = _genes << 5;
        // bitwise OR trait with _genes
        _genes = _genes | _traits[47 - i];
    }
    return _genes;
}
```

Fig. 6. Source code from an example contract. Credit: GeneScience contract at https://tinyurl.com/RE451

Once again, these two different representations of the same function look nothing alike on the surface. Take, for instance, the way the loop is handled. Besides the pseudocode using

```
                    args $s18 $s17

                    rets $s17
```

```
0x55a
--------
$s19 = 0x0
$s20 = 0x0
while (0x1) {
  if ($s20 >= 0x30)
        break
  $t = $s19
  $s19 = $s20
  $s20 = 0x20 * $t
  $t = $s19
  $s22 = 0x2f - $t
  assert($s22 < m[$s18])
  $s19 = (0xff & m[(0x20 * $s22) + (0x20 + $s18)]) | $s20
  $s20 = 0x1 + $t
}
$s17 = $s19
intret()
```

Fig. 7. Pseudocode obtained from Erays using the bytecode generated from the encode function in the GeneScience contract [1].

hexadecimal representation for the numbers, the structure of the source code's for-loop is almost inverted by the pseudocode's of an eternal while-loop and an if-statement checking for the break condition. Just as in the first instance, we are able to relate this pseudocode to the appropriate source code by analyzing the operations being done by the two side-by-side. In the context of a pure structural analysis however, the two seem to bear no resemblance.

B. Bytecode Analysis

Analysis on bytecodes of Ethereum smart contracts were accomplished by utilizing etherscan.io's decompiler tool which allows quick decompliation of smart contracts' contract creation code into bytecode. Erays was also used to further inspect bytecodes of smart contracts. Upon brief analysis of contract bytecodes, we found some patterns in its structure which seem to exist on all smart contract bytecodes.

A smart contract's bytecode is identical to its creation code aside for a certain number of characters at the head of the creation code. The contract creation code contains some amount of characters which are excluded from the bytecode, and seem to possess structure completely identical to its bytecode after the described discrepancy. An example of the characters exclusive to the contract creation code can be seen in Figure 8.

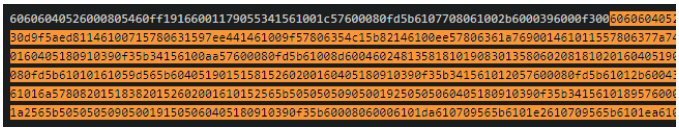

Fig. 8. Unhighlighted section of string showing characters excluded from contract bytecode Credit: GeneScience contract at https://tinyurl.com/RE451

To better understand the purpose and functionality of these bytecode characters which are exclusive to the contract creation code, we decompiled the aforementioned characters with Erays and obtained some high-level code. Interestingly, Erays outputs code which does not resemble the source code of the contract. Furthermore, the code contains functions which are not seen in the original bytecode itself, such as "codecopy". codecopy(t,f,s) copies s bytes while f is the source and t is the destination. The function is also always called at the end of a code, as shown in figure 9.

```
0x0
--------
$m = 0x60
s[0x0] = 0x1 | (0xffffffffffffffffffffff
assert(0 == msg.value)
codecopy(0x0, 0x2b, 0x770)
return(0x0, 0x770)
```

Fig. 9. High-level representation of the bytecodes of GeneScience contract that are exclusive to the contract creation code produced by Erays which shows the usage of the codecopy function [1]

These findings imply that the bytecodes that are only exclusively found in contract creation codes serve as a header which sets up and finalizes a smart contract to be deployed in the blockchain since this pattern holds true for other smart contracts. For these reasons, it may help to utilize these bytecode headers when analyzing ethereum smart contracts as they could contain important information regarding the smart contract's functionality.

C. Malicious Contracts and Counter Initiatives

Smart contract security and detection of malicious contracts is a large and growing area. In this subsection we examine malicious smart contracts and attempt to reverse engineer them such that we can learn their true purpose.

The general concept of a honeypot is to lure people into taking some action via a supposed reward. During or after the action the person pursuing the honeypot hits a trap and takes a loss. On the Ethereum blockchain there exist honeypot smart contracts which seek to fool people into interacting with them. In this way the contract (and thus owner of the contract) can get away with some of their Ether.

The King of the Hill smart contract [8] is an example of a real honeypot that existed on the Ethereum blockchain. The contract advertised that whoever had the largest stake in the contract was the owner. In actuality, the owner was immutable, and set to the user who deployed the contract. We performed a static analysis of the contract to ascertain this fact. First, we took the EVM runtime bytecode of the contract and put it into a Solidity decompiler (https://ethervm.io/). Solidity is the most popular high level language to write Ethereum smart contracts. Next we analyzed the decompiled Solidity code to search for the exploit. We found the exploit

by viewing the decompiled functions and observing the discrepancy between the *Stake* and *Withdraw* owner state variables. They both reference different owner variables. The latter is the pseudo-owner, while the former is the actual owner set at contract deployment.

See this demo video for a deep dive analysis we made of the King of the Hill honeypot: https://youtu.be/m7SsdFWLtRo.

X. Conclusions

We will now present the conclusions we have drawn based on the results and analysis given above. Overall we learned a great deal more about reverse engineering, blockchains, and the Ethereum ecosystem. One motivation for reverse engineering smart contracts is the fact that the Ethereum network is decentralized, there is no central entity that has to help you if something goes wrong.

A. Conclusions on Erays

After comparing various instances of Erays pseudocode to the respective original source code functions, it is our conclusion that, while the same operations are being represented, the structural differences of the two are enough to entirely obscure the structure of the source code. Note that, while we ran our tests with transparent contracts, this means that the code structure of opaque contracts will remain hidden. In terms of our original research goals, this implies that structural vulnerabilities in source code cannot be detected simply by using the open-source software of Erays. What it further implies is that those who wish to attack a smart contract through structural methods (buffer overflow attacks, for instance) will find no use from this resource, which we deem as a positive for the creators and those who wish to use Erays in earnest.

Our results led us to some interesting questions about the structure, however: in the case of the loop highlighted in Figure 6 and Figure 7, what if the same loop were stated different ways in the source code? Would Erays generate the same pseudocode, or would it be different? The results of such a study might have impacts on our current conclusion, but due to certain limitations on what contracts Erays can reverse (a contract has to have been published at the time the most recent Erays version is released for it to be seen as valid by the software), we were unable to conduct such tests at this time.

B. Conclusions on Bytecode Analysis

Regarding analysis of Ethereum smart contract bytecodes, we have concluded that there are recurring patterns for any and all sets of smart contracts. The header, which appears only exclusively in the contract creation code, is an additional part of smart contracts' bytecodes which finalizes and deploys them to the Ethereum blockchain. We were able to reach this conclusion by utilizing the Erays software and analyzing the special bytecode headers taken from various contracts in their high-level structure. This essentially implies that all smart contracts have a recurring, common pattern and this could be a cause for concern regarding their security. Recurring patterns within contracts like these may help attackers find vulnerabilities within them and, eventually, exploit them. Although it is unlikely that the specific header bytecodes we analyzed themselves can become a vulnerability for smart contracts, the same concept can be applied to the contracts' source bytecodes.

C. Conclusions on Malicious Contracts

In this project we confirmed our suspicions that there at least have existed malicious smart contracts on the Ethereum blockchain. We saw this in the King of the Hill honeypot smart contract, where users could be fooled into staking their Ether without knowing they wouldn't get it back. It seems safe to assume there exist more of these types of contracts on the Ethereum blockchain, thus it is important for members of the Ethereum community to be educated about the risk posed by them. While we hold it is prudent to invest in more smart contract reverse engineering tools, we also believe it's unreasonable to expect a lay user of the ecosystem to reverse engineer these contracts. Hence, we assert community efforts to (1) make contract source code open-source or (2) reverse engineer smart contracts for the public to view is a good step towards making people more confident in the Ethereum ecosystem as a whole.

References

[1] Y. Zhou, D. Kumar, S. Bakshi, A. M. Joshua Mason, and M. Bailey, "Erays: Reverse engineering ethereum's opaque smart contracts," *USENIX The Advanced Computing Systems Association*, 2018.

[2] T. L. O. of Eric Lewin. (2017) Reverse engineering smart contracts: Let's not kill all the lawyers (or anyone else) just yet. [Online]. Available: https://www.ericlewin.net/wp-content/uploads/2018/06/Thought-Leadership-Smart-Contracts.pdf

[3] axic: https://github.com/axic. (2021) Ethereum virtual machine (evm). [Online]. Available: https://ethereum.org/en/developers/docs/evm/

[4] T. Jain. (2020) Ethereum virtual machine tutorial — ethereum blockchain. [Online]. Available: https://www.cryptoknowmics.com/news/ethereum-virtual-machine-tutorial-ethereum-blockchain

[5] T. T. (2018) Ethereum evm illustrated. [Online]. Available: https://takenobu-hs.github.io

[6] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 204–217.

[7] teamnsrg: https://github.com/teamnsrg/erays. (2018) Ethereum smart contract reverse engineering. [Online]. Available: https://github.com/teamnsrg/erays/blob/master/LICENSE

[8] T. of Bits. (2020) Not so smart contracts. [Online]. Available: https://github.com/crytic/not-so-smart-contracts