CLUSTERING SMART CONTRACTS BY THEIR VULNERABILITIES

Bachelor Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

by

Abhinav Dutta
(Roll No. 1901CS02)

under the guidance of

Dr. Raju Halder

to

DEPARTMENT OF COMPUTER SCIENCE
INDIAN INSTITUTE OF TECHNOLOGY PATNA
PATNA = 801103, INDIA

May 2023

DECLARATION

I, **Abhinav Dutta (Roll No: 1901CS02)**, hereby declare that, this report entitled **Clustering Smart Contracts by their Defects** submitted to Indian Institute of Technology Patna towards the partial requirement of **Bachelor of Technology** in **Computer Science and Engineering**, is an original work carried out by me under the supervision of **Dr. Raju Halder** and has not formed the basis for the award of any degree or diploma, in this or any other institution or university. I have sincerely tried to uphold academic ethics and honesty. Whenever a piece of external information or statement or result is used then, that has been duly acknowledged and cited.

Patna - 801103 **Abhinav Dutta**
May 2023

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Clustering Smart Contracts By Their Vulnerabilities**" is a bonafide work of **Abhinav Dutta (Roll No. 1901CS02)**, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Patna under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Dr. Raju Halder**

Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology Patna, Bihar.

May, 2023
Patna.

ACKNOWLEDGEMENT

I would like to thank my advisor Dr.Raju Halder for his continuous help and support during this entire journey. His precious advice and suggestions have helped improve this work significantly. He has given me significant independence in shaping out this project and has always been kind and considerate to me. My colleagues at the Programming Languages and Robotics Lab have always been of immense help as well.

I have been very fortunate to have found the best group of friends at IIT Patna, some at the department of CSE and some at my hostel, who made have my stay memorable and supported me through the thick and thin. I cannot thank them enough for their support and companionship.

I would also like to thank Prof. Ron Shamir from Tel Aviv University. I did my first research project under him and he has been a guiding light ever since. I would also like to thank Dr. Xujie Si from McGill University since I did my first significant project on program analysis under him and it has been a real pleasure having worked under him. Finally and most importantly, I convey my regards and gratitude to all the members of my mother, father and brother who has been a constant source of inspiration in my life.

Patna - 801103 **Abhinav Dutta**

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

In the project, I have focused on clustering smart contracts together based on their vulnerabilities found in them. The motivation behind this project was to speed up the process of validating large batches of smart contracts. Let us consider a situation where we are required to validate a large number of contracts- it is very likely that a large fraction of them will not have any bugs. Thus, validating all such contracts one-by-one seems to waste a lot of computational resources. I feel it's better to pass all such contracts through a preliminary filter that can approximately identify suspicious contracts (which would hopefully be a small fraction of the total). Then we can use our heavy machinery of verification (formal or otherwise) on this smaller set of suspicious contracts. Such a methodology works if we can identify a suitable distance metric between contracts such that this distance correlates well with the type of vulnerabilities found in them.

Once, such a distance metric is identified, we can use it to modify standard clustering algorithms(k-means, hierarchical) or use the cheaper locality-sensitive hashing alternatives to cluster the smart contracts. My work mainly has been on trying various distance measures and comparing their suitability in this problem setting.

1.2 Contributions

My main contributions in the study were :

1. Design various methods of measuring the distance between smart contracts and compare their suitability based on their correlation with vulnerabilities

found in such contracts

2. Use the above distance metrics to cluster smart contracts and analyze their effectiveness

Chapter 2

Literature Survey

Clustering smart contracts based on their vulnerabilities is an entirely new problem and as such, I could not find any work on this topic. There are however papers that are related to clustering smart contracts or finding vulnerabilities in smart contracts in general. They have served as inspiration in this work.

2.1 N. Jia et. al

This is the only paper that describes the usage of LSH (locality sensitive hashing) in order to cluster smart contracts. This paper however has not mentioned what distance metric their proposed LSH actually preserves so it's hard to evaluate their scheme. Also, its unclear what their objective is (or in other words, on what basis were the smart contracts clustered), though it seems that they wanted to cluster based on semantic content. The authors have claimed that ERC related smart contracts, Gambling Related smart contracts, and game-related smart contracts were clustered separately. The method has two main steps :

1. **Collecting features** - First, they parse an abstract syntax tree and obtain the syntactic tokens of each code line. These syntactic tokens are handcrafted (there is a group of 90 such token) - like IfStatement, BinaryExpression, and CallExpression,etc. As their code has been kept private, it is difficult to know what exact features they use and thus not possible to reproduce their results. Each code line may contain multiple types of syntax tokens. The syntax tokens contained in each code line are regarded as a token unit, and the token units contained in a contract are used as features to measure the similarity between contracts. A feature matrix is built for all the contracts based on the token units contained in them. Two vector matrices are built for

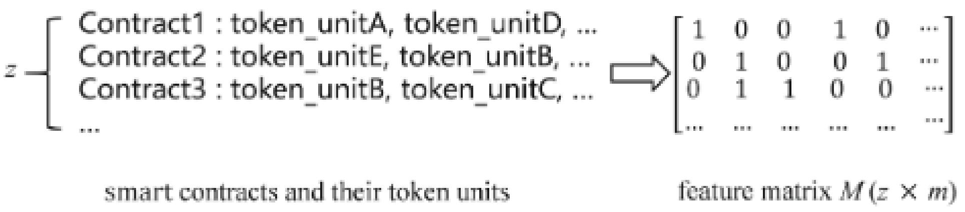

z
Contract1 : token_unitA, token_unitD, ...
Contract2 : token_unitE, token_unitB, ...
Contract3 : token_unitB, token_unitC, ...
...
smart contracts and their token units
feature matrix M (z × m)

Figure 2.1: Converting Smart Contracts to Vector[1]

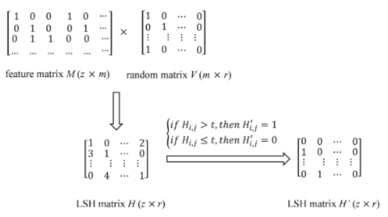
user-created and contract-created smart contracts, respectively. The matrix M is used to represent the token units contained in each contract, where a contract is labeled as 1 if it contains a certain token unit (basically a contract is represented as the one-hot encoding of the hand-selected tokens found in it). The resulting feature matrix is zm, where z is the number of user-created smart contracts and m is the number of distinct token units.

2. LSH - LSH method has been used to cluster the similar contract based on the feature matrix M as follows:

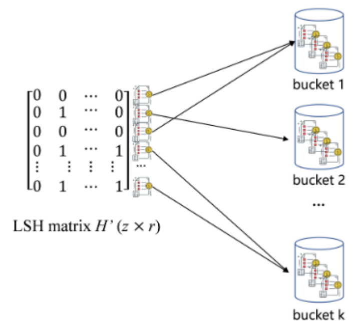
   (a) Randomly generate a zero-one matrix V with mr dimensions. Each row of this matrix can be seen as some sort of feature selector/filter . If the $i-th$ element of a row is 1 then it means that this selector gives importance to the $i-th$ feature/token in the contract, otherwise not.

   (b) Multiply matrices M and V, and obtain third matrix H. Each element $H(i.j)$ in H represent the product between the feature vector of a smart contract ci and a random zero-one vector (feature selector). If $H(i.j)$ is greater than a threshold t, the locality-sensitive hashing value $h(c_i)$ of the smart contract is 1. Otherwise, $h(c_i)$ is 0. Repeating the previous steps r times, we can get r locality-sensitive hashing values.

   (c) Splice these values together and obtain a hashing sequence consisting of 0 and 1 with r length for smart contract c_i, i.e., $H(c_i) = (h1(c_i), ..., hr(c_i))$. Finally, two smart contracts will have the same hash value if they both have a sufficient number of features wrt to r feature selectors as defined in V.


feature matrix M ($z \times m$)
random matrix V ($m \times r$)
if $H_{i,j} > t$, then $H'_{i,j} = 1$
if $H_{i,j} \le t$, then $H'_{i,j} = 0$
LSH matrix H ($z \times r$)
LSH matrix H' ($z \times r$)

(a) LSH part 1[2]



(b) LSH part 2[3]

2.2 SmartCheck

Also, I have used SmartCheck as the tool to establish the ground truth of smart-contract vulnerabilities in the dataset used in this study. So, it is worthwhile to look as how it works.

Smartcheck is a static tool that detects bugs based on pattern matching. Smartcheck has a database of vulnerable patterns. It usually includes three stages:

1. building an intermediate representation (IR), such as abstract syntax tree or three-address code, for a deeper analysis com- pared to analyzing text. SmartCheck XML parse tree as IR.

2. enriching the IR with additional information ing algorithms such as control- and dataflow analysis (syn- onym, constant, and type propagation, taint anal- ysis, symbolic execution, abstract interpretation,etc. SmartCheck does not do any of this though the authors suggest that their tool can be extended to have this feature.

3. vulnerability detection w.r.t. a database of patterns, which define vulnera- bility criteria in IR terms.

To have a better understanding of how it performs its job. Let us consider the problem of detecting Balance equality. We should avoid checking for strict balance equal- ity because an adversary can forcibly send ether to any account by mining or via selfdestruct

```
if ( this . balance == 42 ether ) { /* ... */ }
// bad
if ( this . balance >= 42 ether ) { /* ... */ }
// good
```

```
<Rule>
    <RuleId>SOLIDITY_BALANCE_EQUALITY</RuleId>
    <Patterns>
      <!-- Looks for strict balance comparisons: ".balance_==", ".
         balance_!=".   -->
      <Pattern patternId="5094ad">
        <Categories>
          <Category>Solidity</Category>
        </Categories>
        <Severity>1</Severity>
        <XPath>
          //expression
            [comparison]
            [
              expression[matches(text()[1], "\.balance$")]
              or expression/tupleExpression/expression[matches(text()
                 [1], "\.balance$")]
              or expression/expression[matches(text()[1], "\.balance$
                 ")]
            ]

        </XPath>
      </Pattern>
    </Patterns>
</Rule>
```

Similarly we can define patterns for other vulnerabilities as well :

1. Unchecked external calls - The pattern detects an external function call (call, delegatecall, or send) which is not inside an if-statement.

```
<Rule>
    <RuleId>SOLIDITY_UNCHECKED_CALL</RuleId>
    <Patterns>
        <Pattern patternId="f39eed">
            <Categories>
                <Category>Solidity</Category>
            </Categories>
            <Severity>3</Severity>
            <XPath>
                //functionCall
                    [
                        functionName//identifier
                        [matches(text()[1], "^call$|^
                           delegatecall$|^send$|^callcode$")]
                        and not(ancestor::ifStatement)
                        and not(ancestor::returnStatement)
                        and not(ancestor::functionCall)
```

```
                              and not(ancestor::variable-
                              -DeclarationStatement)
                              and not(ancestor::expression and
                                  ancestor::expression[text()[1] = "="
                                  or lvalueOperator])

                         ]
               </XPath>
          </Pattern>
     </Patterns>
</Rule>
```

2. Re-entrancy - Such bugs don't have an exact pattern. To address this, CEI (Checks Effects Interactions) pattern has been used but this is subject to significant false positives.

Chapter 3

Popular Vulnerabilities

Previous works (Rexford et.al,Yingjie et.al) have identified the following as the most prominent vulnerabilities in ethereum smart contracts :

1. **Reentrancy** The reentry is well-known smart contracts vulnerability which allows the execution and utilisation of code from external contracts. External calls are used to perform tasks such as activating an external contract or delivering cryptocurrencies to an account. An intruder might hijack the external call and compel the contracts to run reentrant code, even calling back to themselves. As a result, sim- ilar to indirect recursive method calls in programming languages, the same codes are executed repeatedly. The DAO contract's vulner- ability was discovered in 2016. Example when an account is sent a call method in a Solidity contract, a procedure called the fallback function is invoked. This process can be replicated until the gas is depleted if the callback feature invokes a call form. For eg.

   Listing 3.1: Re entrancy example

   ```
   pragma solidity 0.4.19;
   contract Fund {
   mapping ( address => uint ) balances ;
   function withdraw () public {
   if ( msg . sender . call . value ( balances [ msg . sender ]) () )
   balances [ msg . sender ] = 0;
   }
   }
   ```

2. **Transaction Ordering Dependency (TOD):** Smart contracts accuracy differs depending on transaction sequences on the blockchain. From start to finish, the transaction requires a certain amount of time. If a transaction

to change the contract is conducted within this time frame and the changed transaction is con- firmed first, the transaction that was initiated first would be impacted. In fact, this dependence on transaction order is a big issue; for example, a vendor can adjust the price after a customer has bought something, causing the buyer to pay more without their permission.

Listing 3.2: TOD example

```
pragma solidity ^0.4.18;

contract TransactionOrdering {
    uint256 price;
    address owner;
    event Purchase(address _buyer, uint256 _price);
    event PriceChange(address _owner, uint256 _price);
    modifier ownerOnly() {
        require(msg.sender == owner);
        _; }
    function TransactionOrdering() {
        // constructor
        owner = msg.sender;
        price = 100; }
    function buy() returns (uint256) {
        Purchase(msg.sender, price);
        return price;}
    function setPrice(uint256 _price) ownerOnly() {
        price = _price;
        PriceChange(owner, price);}
}
```

3. **Predictable Random Number (PRN):** To produce pseudo-random numbers, the contract can use predictable seeds such as block timestamps and block numbers. Hackers can take advantage of this in lottery contracts to improve their win- ning rate. Most tools only detect timestamp dependencies in these forms of vulnerabilities.

4. **Timestamp Dependency:** When a contract utilises block variables as a call condition to exe- cute crucial operations (e.g., transmitting tokens) or as a seed to produce random numbers, it exposes itself to this vulnerability. Amount, BLOCKHASH, COINBASE, DIFFICULTY, GASLIMIT, and TIMESTAMP are all variables that originate in the block header and may thus be influenced by miners. Miners, for example, have the option to set the block TIMESTAMP within a 900-second off- set. Miners will manipulate

the flaw through tampering with block variables if cryptocurrency is passed dependent on them

5. **Integer Overflow:** In most smart contract languages, the integer form, such as i64 in web-assembly and in Solidity's uint256, has a limited range. If the value of an integer attribute is higher than the range, it would be adjusted to fit within the range, resulting in incorrect performance

Chapter 4

Methods

4.1 Dataset

Our dataset consists of roughly 61000 smart contracts all labeled with a 25-bit one hot encoding generated by using SmartCheck indicating which vulnerabilities were detected in the smart contract.

4.2 Vulnerability unaware distance Measures

Such methods do not require us to know anything about possible vulnerabilities while measuring distances. Such methods give a general idea about the similarity between 2 smart contracts. Such measures can be used to quantify other things like plagiarism/code clones amongst smart contracts. Here, we define the distance between two contracts D and hope that vulnerable contracts appear closer to each other. Designing such a good D is always very challenging. Below, I present a few metrics that I considered :

4.2.1 Jaccard Distance

Jaccard distance is a measure of similarity between two sets, often used in text mining and information retrieval. It measures the size of the intersection of two sets divided by the size of the union of the sets. The Jaccard distance between two sets A and B can be defined as $1 - J(A, B), where$J(A,B) is the Jaccard similarity coefficient between the two sets, which is defined as $\frac{|A \cap B|}{|A \cup B|}$. The Jaccard distance takes values between 0 and 1, where 0 indicates that the sets are identical and 1 indicates that the sets have no elements in common. The Jaccard distance is popularly used to compare the similarity between documents or sets of keywords, among other applications.

4.2.2 Cosine Distance

Cosine distance is a measure of similarity between two vectors in a high-dimensional space. It measures the cosine of the angle between the two vectors, which is equivalent to $\frac{v_A.v_B}{|v_A||v_B|}$ where v_A, v_B denote the embeddings of A and B. In other words, cosine distance measures how similar the direction of two vectors is regardless of their magnitude. It ranges from -1 (opposite directions) to 1 (identical directions), with 0 indicating no correlation. Cosine distance is commonly used in natural language processing tasks, such as text classification, document clustering, and information retrieval, where the vectors represent word frequencies or TF-IDF weights of documents. Compared to other distance metrics, cosine distance is computationally efficient and insensitive to the scale of the data. However, it may not be suitable for cases where the magnitude of the vectors is important.

4.2.3 Euclidean Distance

Euclidean distance is a measure of the distance between two points in a Euclidean space. It is the straight-line distance between two points in a space with a Cartesian coordinate system. It is defined as $d_E(u, v) = ||u - v|| = \sqrt{\sum_{i=1}^{k}(v_i - u_i)^2}$. The Euclidean distance is widely used in data analysis, machine learning, and other fields as a way to measure the similarity or dissimilarity between two vectors or data points.

4.2.4 Edit Distance

Edit distance, also known as Levenshtein distance, is a measure of the similarity between two strings based on the number of operations required to transform one string into the other. These operations can include insertions, deletions, and substitutions of individual characters. The edit distance between two strings is defined as the minimum number of operations required to transform one string into the other. Edit distance has various applications, including spell checking, DNA sequence alignment, and speech recognition. However, it can be computationally expensive to calculate for long strings or large datasets.

4.3 Locality Sensitive Hashing

Definition : Let d be a distance measure, and let $d_1 < d_2$ be two distances in this measure. A family of functions F is said to be (d_1, d_2, p_1, p_2)-sensitive if for every $f \in F$ the following two conditions hold :

1. If $d(x, y) \leq d_1$ then $Probability[f(x) = f(y)] \geq p_1$
2. If $d(x, y) \geq d_2$ then $Probability[f(x) = f(y)] \leq p_2$

For the purpose of this study, we will denote F as LSH preserving the distance measure d. And we will use a looser definition of such functions as $1 - d(x, y) = Probability[f(x) = f(y)]$ (d measures the distance, thus $1 - d$ measures the similarity). To sum up, under such schemes 'closer' objects are likelier to have same hash value.

In this study, we will mainly focus on 3 well-known distance measures (Jaccard, Euclidean and Cosine) and thus we list the LSH construction for each of these types below :

4.3.1 LSH preserving Jaccard Distance

A very popular method called MinHashing addresses this problem. MinHashing is a technique used for estimating the similarity between two sets by comparing the hashes of their elements. The main idea is to represent each set as a signature (a small set of hash values) that preserves the original set's similarity with other sets. The signature is constructed by randomly selecting a set of hash functions, and for each function, computing the hash value of the minimum element in the set that maps to that function.

MinHashing can be used to preserve Jaccard similarity by selecting hash functions that are independent and uniformly distributed. In this case, the probability that two sets have the same minimum element for a given hash function is equal to their Jaccard similarity. By selecting a large number of hash functions and computing the minimum element for each function, the signature of a set can be created, and the Jaccard similarity between two sets can be estimated by comparing their signatures. The advantage of MinHashing is that it is fast and memory-efficient, making it suitable for large-scale similarity analysis of big data sets.

4.3.2 LSH preserving Euclidean Distance

The hash h is defined as follows :

1. First take a random unit vector $u \in \mathbb{R}^d$. A unit vector u satisfies that $||u|| = 1$, that is $d_E(u, \overrightarrow{0}) = 1$.
2. Project a, b onto u. The projection of u on a is defined as :

   $$a_u =< a, u >= \sum_{i=1}^{d} a_i.b_i$$

   This is contractive so $||a_u - b_u|| \leq ||a - b||$
3. Create bins of size γ on u (or appropriately, on a_u which is now in $\mathbb{R}^1$). The index of the bin falls into is $h(a)$ (here h can be any hash function, say SHA-256). Intuitively, the length of the projection of two vectors 'near' each other will be close. If $||a - b|| < \gamma/2$ then $Pr[h(a) = h(b)] \geq 1/2$. If $||a - b|| > 2\gamma$ then $Pr[h(a) = h(b)] < 2/3$. So this is $(\gamma/2, 2\gamma, 1/2, 1/3)$-sensitive.

   To see the second claim (with 2γ) we see that for a collision we need $cos(a - b, u) < \pi/3$ (out of $[0, \pi]$). Otherwise $||a - b|| > 2||a_u - b_u||$ and thus they must be in different bins. We can also take $h(a) = <a, u> mod(\gamma)$. For large enough t, the probability of collision in different bins is low enough that this can be effective.

4.3.3 LSH preserving Cosine Distance

Consider two vectors in this space, x and y, along with a hyperplane through the origin with a normal vector v. These two vectors x and y have a cosine distance that is equal to the angle $thea$ between them, and taken alone, this angle can be measured in the plane defined by the vectors. The hyperplane is randomly chosen via its normal vector v, and will then determine the sign of the dot products $v.x$ and $v.y$. If v passes between x and y then the dot products v.x and v.y will have different signs, otherwise same. Intuitively, if the angle between two vectors is small, then it is likely that our random vector will not pass through those vectors, thus these 2 vectors lie on the same side of the random vector.

The probability that v will pass between x and y is $\frac{\theta}{2\pi}$. We create each hash function $f \in F$ using a randomly chosen vector v, and can then say that $f(x) = f(y)$ if the dot products $v.x$ and $v.y$ have the same sign (say $f(x) = sign(x.v)$).

4.4 Bytecode

Smart contract bytecode is the compiled form of Solidity or other Ethereum Virtual Machine (EVM) compatible programming languages that is executed on the

Ethereum blockchain. When a Solidity source code is compiled using a compiler such as solc, it is converted into a sequence of machine instructions that can be executed by the Ethereum Virtual Machine. This sequence of instructions is known as the bytecode.

Smart contract bytecode is a low-level representation of the smart contract code that is closer to the actual instructions executed by the Ethereum Virtual Machine. It consists of a series of hexadecimal values representing the instructions that the EVM executes. Each bytecode instruction corresponds to an operation that can be executed by the EVM, such as arithmetic, logic, storage, and flow control.

The bytecode is deployed on the Ethereum blockchain as part of a smart contract transaction, and once it is deployed, it cannot be changed. This means that the bytecode is immutable and can be used to verify the integrity of the smart contract code. Furthermore, since bytecode is the compiled form of the smart contract code, it is less susceptible to obfuscation techniques and can be a more effective way of detecting code clones and vulnerabilities. This convinces us to try to use this as well to measure similarity between contracts.

4.5 Measuring correlation between a distance metric and Vulnerability

We calculate the vulnerability distance between two smart contracts as the hamming distance between the labels of those two contracts (note that the label is a one-hot encoding of size 25). This distance measure how similar the vulnerabilities are between these two contracts. Then we calculate its correlation to our proposed custom distance metric to find the correlation between the proposed metric and vulnerability.

4.6 List of Distance measures

This is a summary of the distance measures used in this study :

Table 4.1: List of distance measures

Name	Basis	Distance metric
d_1	source code, ngrams	Jaccard
d_2	source code	Edit Distance
d_3	byte code, ngrams	Jaccard
d_4	byte code	Edit Distance

Chapter 5

Results

5.1 Regarding goodness of metrics

All the * values indicate Spearman correlation, otherwise Pearson. Ideally, we have no reason to believe that the relation should be linear. We have used spearman/pearson correlation based on whichever had better p-value.

Table 5.1: Correlation to Vulnerability of Various Distance Measures

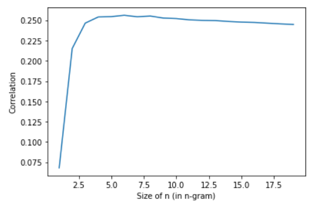
Distance Measure	Correlation
d_1	0.256
d_2	0.573
d_3	0.399
d_4	0.136

5.2 Regarding clustering

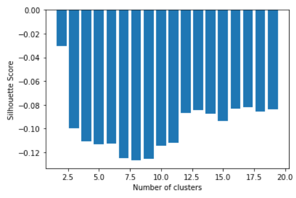
Note that I have used Hierarchical Agglomerative clustering for all the results below :

Table 5.2: Clustering performance of Various Distance Measures

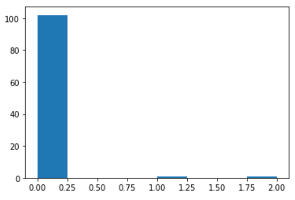
Distance Measure	Silhouette Score
d_1	-0.0301
d_2	0.626
d_3	0.47089
d_4	0.839



(a) Plot to determine optimal n(=6)[1]

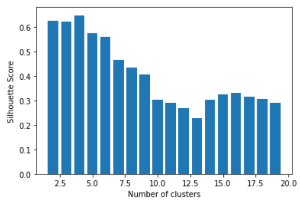


(b) Plot to determine optimal no. of clusters (=3)[2]

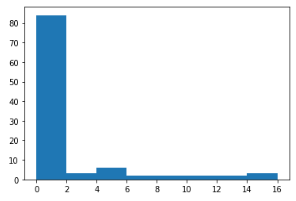


(c) Histogram of clusters showing most contracts put in a basket[3]

Figure 5.1: The following results are for d_1

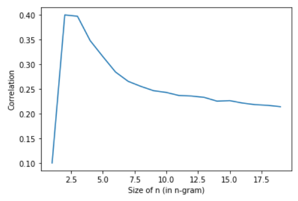


(a) Plot to determine optimal no. of clusters [4]

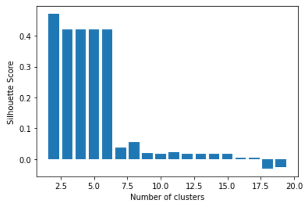


(b) Histogram of clusters [5]

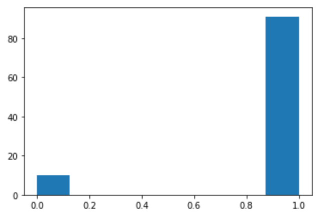
Figure 5.2: The above results are for d_2



(a) Plot to determine optimal n (n-gram) [6]

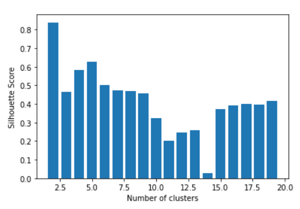


(b) Plot to determine optimal no. of clusters (=3)[7]

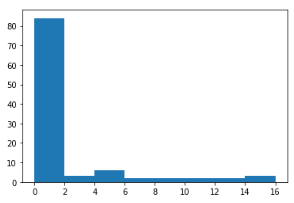


(c) Histogram of clusters showing most contracts put in a basket[8]

Figure 5.3: The following results are for d_3



(a) Plot to determine optimal no. of clusters [9]



(b) Histogram of clusters [10]

Figure 5.4: The above results are for d_4

Chapter 6

Discussion

6.1 Key Takeaways

- Clustering on the basis of bytecode is more effective than doing so on the basis of plain syntax. Currently, it seems that using the edit distance metric on bytecode of the smart-contract is the most efficient distance metric.
- Better correlation with vulnerability always does not lead to better clustering (as can be seen from the Silhouette scores).

6.2 On the accuracy of the methods

I cannot help but notice that the accuracy of the methods proposed was underwhelming. I feel this is the case because the basic problem of determining whether smart contracts have bugs based on syntax is inherently difficult. This, I feel, is because bugs are inherently highly localized incidents. A smart contract of hundreds of lines may have bugs localized to a few lines (or maybe even a single one!). This means a vulnerable contract and a fixed contract will always have most of the contents in common. This, I feel is the main reason why syntax-based analysis does not do so well in this task. Also, it should be noted that tools in this area are plagued with false positives, which often goes unnoticed as the focus is mainly on accuracy. Durieux et.al in their study have found that 97% of the contracts were tagged as vulnerable (though they used a host of tools - HoneyBadger Maian Manticore Mythril Osiris Oyente Securify Slither Smartcheck, and each of their false positives individually were not reported). This is a big problem - having such high false positives (in the 90s) and such low true positives (11-30%) really hurts the validity of studies like this since there are no reliable sources for ground truth.

Also, the validity of SmartCheck(the tool used to generate the labelings in the dataset used in this study) itself should be scrutinzed. Recently, Ferreira et.al have shown that SmartCheck could only detect 13/115 (ie a low 11% !) errors in smart contracts ! This could be due to the fact the SmartCheck depends on pattern checking and their database of patterns were built based on the type of buggy smart contracts found by the authors during that time. As time evolves, the vulnerable patterns evolve rendering such a static tool useless. The paper also pointed out that

Chapter 7

Conclusion

7.1 Summary of Main Findings

- Edit Distance on the smart contract source code has the best correlation with vulnerability.
- Edit Distance on smart contract bytecode results in the best clustering.

7.2 Directions for Future Work

Given the unreliability of the tool SmartCheck, I feel it is worthwhile to re-evaluate the methods presented in this study using some other vulnerability method detection tools. I have checked other tools like Slither and SmartEmbed on a small batch of labelled dataset and obtained the following results (a more extensive result over a larger dataset and using a larger number of tools can be found in the study published by Durieux et.al) :

I think the following tools would be suitable for a follow-up :

- **Slither** - Out of the three tools I checked ,(results above), this had the best performance. It maybe worthwhile to reproduce this study using this tool. This tool was developed by TrailOfBits.It is a static analysis tool that converts Solidity smart contracts into an intermediate representation called SlithIR and applies known program analysis techniques such as dataflow analysis and taint tracking to extract and refine information.
- **Mythril** - Though, I have not personally tested this tool, many studies have used this tool for benchmarking their contracts, giving this tool some credibility. Also, it was shown to have the highest accuracy on the SmartBugs

Table 7.1: Accuracy of tools on SmartBugs Curated Dataset

Category	**SmartCheck**	**Slither**	**SmartEmbed**
Re-entrancy	60%[9/15]	100%[15/15]	0%[0/15]
Integer Overflow	6%[1/15]	6%[1/15]	6%[1/15]
Bad PRNG	0%[0/8]	25%[2/8]	25%[2/8]
All	26%[10/38]	47%[18/38]	8%[3/38]
Popular Attacks	**SmartCheck**	**Slither**	**SmartEmbed**
DAO Example	Yes	Yes	No
Spankchain Example	No	Yes	No

dataset (which is 27% only, but at least better than the 11% reported on SmartCheck). However, like every other tool in this area, this tool is also plagued with false positives. This tool was developed by ConsenSys. It uses concolic analysis, taint analysis and control flow checking of the EVM bytecode to prune the search space and to look for values that allow exploiting vulnerabilities in the smart contract.

Bibliography

[1] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko and Y. Alexandrov, ”SmartCheck: Static Analysis of Ethereum Smart Contracts,” 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Gothenburg, Sweden, 2018, pp. 9-16.

[2] J. Feist, G. Grieco and A. Groce, ”Slither: A Static Analysis Framework for Smart Contracts,” 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Montreal, QC, Canada, 2019, pp. 8-15, doi: 10.1109/WETSEB.2019.00008.

[3] Z. Gao, L. Jiang, X. Xia, D. Lo and J. Grundy (2021), ”Checking Smart Contracts With Structural Code Embedding,” in IEEE Transactions on Software Engineering, vol. 47, no. 12, pp. 2874-2891, 1 Dec. 2021, https://ieeexplore.ieee.org/document/8979435

[4] Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: POPL (1988)

[5] https://github.com/smartbugs/smartbugs-curated

[6] Rexford Nii Ayitey Sosu, Jinfu Chen, William Brown-Acquaye et al. A Vulnerability Detection Approach for Automated Smart Contract Using Enhanced Machine Learning Techniques, 31 August 2022, PREPRINT (Version 1) available at Research Square [https://doi.org/10.21203/rs.3.rs-1961251/v1]

[7] Yingjie Xu, Gengran Hu, Lin You, Chengtang Cao, ”A Novel Machine Learning-Based Analysis Model for Smart Contract Vulnerability”, Security and Communication Networks, vol. 2021, Article ID 5798033, 12 pages, 2021. https://doi.org/10.1155/2021/5798033

[8] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2021. SmartBugs: a framework to analyze solidity smart contracts. In Proceedings of the

35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20). Association for Computing Machinery, New York, NY, USA, 1349–1352. https://doi.org/10.1145/3324884.3415298

[9] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20). Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/3377811.3380364

[10] Andrew Wylie. 2013. Locality-Sensitive Hashing. https://cse.iitkgp.ac.in/ animeshm/algoml/lsh.pdf

[11] Jeff M Phillips.Locality Sensitive Hashing. https://users.cs.utah.edu/ jeffp/teaching/cs5955/L6-LSH.pdf