Fuzzy Hashing Ethereum Smart Contracts

Bachelor's Thesis in Software and Information Engineering

April 10, 2022

Author: Raphael Nußbaumer - 01526647 - nussi.rn@gmx.at

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gernot Salzer

1 Abstract

Because of the high number of smart contracts deployed every hour, fast classification is needed for analysis of blockchain activity. Fuzzy hashing (similarity preserving hashing) is a popular tool when dealing with large amounts of data. In this work we explore the landscape of binary similarity hashing and evaluate such methods for use on ethereum smart contracts. To aid the process a evaluation framework in python was implemented and sets of pre-classified contracts where defined.

2 Terms

Fuzzy hashing function Function producing similar hashes for similar inputs.

Ethereum The computer network running the EVM.

EVM Ethereum Virtual Machine. There is one EVM with one state. Transactions change the state.

Blockchain Complete record of all transaction.

Transaction Can change account balances, call smart contract functions or deploy smart contracts.

Ethereum account Accounts are uniquely identified by there 20 byte address, have an Ether balance and can execute transactions.

Ether Currency used to pay for the execution of transactions.

Smart Contract An Ethereum account with associated runnable code and data stored on the EVM. Can refer to just the runtime code or the source code or the semantics of the contract interface.

Contract interface Smart Contract have a Set of contract functions callable by other accounts via message or transaction.

Contract function Can change account data and Ether balances and call functions of the same of other contracts via messages.

Meassage Smart contracts can call other contracts functions via messages. Unlike transactions and logs, these messages are not stored on the blockchain.

Logs Smart contracts can create logs, which are stored on the blockchain and can therefore easily be observed and referenced. Decentralized applications are implemented using logs.

Runtime code The runnable code stored on the EVM as a sequence of opcodes. Used synonymously with code or bytecode.

Deployment code Runs once and stores the runtime code.

Opcode An EVM instruction encoded as one byte.

Levenshtein distance Also called edit distance. The minimum number of inserts, deletions and substitutions necessary to change one string into the other.

Levenshtein similarity $similarity(a,b) = 1 - distance(a,b)/max\{|a|,|b|\} \in [0,1]$.

Jaccard-Index $J(A,B) = |A \cap B|/|A \cup B| \in [0,1]$

ABI Standard encoding of the Contract interface.

Function signature String containing function name and parameters types.
E.g.: `deposit(uint256,address)`

Fourbyte signature Function signature hashed to a 4-byte value.

solc Standard Solidity compiler.

Solidity Most common and original smart contract language.

3 Introduction

There are various ways of classifying smart contracts, they fall into the two main categories of static analysis and dynamic analysis. Dynamic analysis is concerned with transactions, messages, contract creation and in general temporal associations between accounts. Static analysis is concerned with static data stored on the EVM.

This work is focused on one type of static data, the runtime code (a.k.a. deployed code) of smart contracts stored on the EVM, which needs to be distinguished from the deployment code used to generate to runtime code. To find associations between runtime codes one can exacts match code skeletons and extract the fourbyte signatures of there interface functions, explained in detail in the Section "5.2 Static Analysis" of the paper (Di Angelo et al. 2020b [3]) and quickly contextualized in later sections.

Exact matching skeletons is specific but not sensitive and interface similarity is sensitive but not specific, other methods are desired to fill the gap. For this purpose I looked at the landscape of fuzzy hashing functions.

Related fields are identification of malicious executables, email/comment spam detection, typing auto correction, fuzzy text search, DNA distance metrics and finding video/audio copies and edits.

4 Methods Overview

To obtain similarity scores I preprocessed the codes, calculated digests(hashes) and compared those digest via similarity measures. Following is a quick summary of the methods used in this work.

4.1 Digest Methods

The following methods where used to obtain digests from codes for quick comparison via the similarity measures.

Fourbytes Extraction of the set of fourbyte signatures used to identify the interface functions. [10]

Bytebag Counting all opcodes in the code to form a multiset or bag of byte-values, a bytebag. [5]

Size Just the length of the code in bytes.

ssdeep A Context Triggered Piecewise Hashing (CTPH) Function. [4]

ppdeep Slightly different implementation of ssdeep in pure python. [12]

ppdeep_mod Modified version of ppdeep. [5]

LZJD - Lempel-Ziv Jaccard Distance Generates a compression dictionary. [9]

jumpHash A piecewise hash splitting by the `JUMPI` instruction opcode `0x57`. [5]

bzHash Also splits by `JUMPI` and calculate a compression-ratio for each piece. [5]

4.2 Similarity Measures

To compare the digests the following similarity measures where applied.

Jaccard Index Defined on two sets A, B, the Jaccard Index J is the ratio of common entries to all entries.

$$J(A, B) = |A \cap B|/|A \cup B| \in [0, 1] \tag{1}$$

Levenshtein similarity Based on the Levenshtein Distance.

Normalized Compression Distance (NCD) NCD is a measure for how well two files co-compress. The more features two files have in common the shorter the result when compressing the concatenation of the two files.

4.3 Pre-Processing Methods

Because small changes in the codes, can cause big changes in the digests pre-processing aims the remove non-essential parts.

Segment selection Selecting e.g. only the first code segment, the actual contract.

Skeletonization Setting argument, data and meta section bytes to zero.

Opcode filteringn Filtering for more significant opcodes by setting the others to zero or cutting them out.

5 Pre-Processing

5.1 Segmentation

Segmentation splits the codes into `code`, `data` and `meta` sections *ethutils*[10].

`meta` sections have no effect on execution and change between compilations.

`data` sections are e.g. constructor arguments. Constructor-arguments are deployed by the actual contract (first code-section). Generally they're only used for parameter initialization and have no essential effect on the execution. Limiting the possible effectiveness is the fact that detection is heuristic.

The first `code` section is the the actual contract, the other code-sections are in essence just data for the first-section, if it itself deploys further contracts.

5.2 Skeletonization

Skeletons are runtime codes where constructor arguments, data sections, meta sections and push arguments are set to zero *ethutils*[10]. Contracts can be associated via skeletons, because many deployed codes have identical skeletons. Push-operations are the only EVM-instructions followed by data. The reasoning behind this removal is that, these data bytes have no essential effect on the execution, e.g. jump-addresses and ethereum-addresses. Setting the push arguments to zero has the benefit of preserving the ability to disassemble the code.

5.3 Opcode filtering

The same externally observable behavior can be achieved via different opcode sequences. Some opcodes are more likely to change with solc-versions or compile-options than others. Removing less significant opcodes before hashing should yield more meaningful similarity scores.

5.3.1 `fStat` Filter [5]

Determining exactly which opcode sequences change, would most likely require a lot of manual work. To quickly and simply distinguish between opcode which are likely to change and once that don't change I used the *solc-versions-testset*[6] and reduced the codes to *bytebags*. Then I calculated an the *One-way ANOVA F-test statistic* (2) for each opcode, using common source-code as grouping criterion and the count of the opcode as value.

$$F = \frac{\textit{between group variability}}{\textit{within group variability}} \tag{2}$$

To define the *fStat* filter I selected to top 30 opcodes by f-statistic value, plus a few that never occurred or only in one group.

```
OPCODES = (
  # top 30 fStat values
  ADDRESS, LOG3, TIMESTAMP, ORIGIN, LOG4, SHA3, SWAP14, CALLDATASIZE,
  CALLDATACOPY, SIGNEXTEND, CALL, LOG2, RETURNDATASIZE, CALLER, EXTCODESIZE,
  JUMPI, STATICCALL, RETURNDATACOPY, GAS, DUP13, DUP5, DUP8, GASPRICE,
  SHR, PUSH4, ISZERO, DUP7, ADD, DUP9, MUL,
  # occurred in only one group
  XOR, CALLVALUE, DELEGATECALL, SELFDESTRUCT,
  # never occurred
  SAR, LOG0, CREATE,
)
```

6 Similarity Measures

Jaccard Index, Levenshtein similarity, NCD

7 Data sets

7.1 solc-versions-testset

To evaluate the similarity measures I selected a set of 13 solidity smart contracts and compiled them with different solc versions and compiler options *solc-versions-testset*[6]. Necessary changes where made to the source code to ensure compatibility with the various solc versions.

To test the robustness against compiler version the contracts where compiled with the four solc versions `0.5.16`, `0.6.12`, `0.7.6` and `0.8.4`. To evaluate the effect of code optimization, the four optimization-options `{enabled:false,runs:200}`, `{enabled:true,runs:0}`, `{enabled:true,runs:200}` and `{enabled:true,runs:999999}` where applied. Finally ABI encoders `v1` and `v2` where used.

13 source-codes x 4 solc-versions x 4 optimization-options x 2 abi-encodings ≈ 264 codes.

7.1.1 Optimization

The runs setting determines whether the compiler optimizes the code for cheap deployment or cheap execution, i.e. cheap deployment code execution or cheap runtime code execution.

To determine the relevant optimization options I calculated a statistic [Tbl. 1] on contracts with verified source available on etherscan.io.

optimization-runs	count	proportion	optimization-enabled
0	1379	0.6 %	31.6 %
1	671	0.3 %	99.0 %
< x <	1804	0.8 %	99.4 %
200	202289	90.3 %	50.0 %
< x	17973	8.0 %	99.2 %
total	224116	100 %	54.4 %

Tbl. 1: optimization setting statistic

7.2 Wallets

Extensive set of wallet contract codes, classified into 40 blueprints via various automated and manual means described in (Angelo et al. 2020 [1]).

This dataset is interesting because it's large, the types are human verified and quite different from each other.

7.3 Proxies

Individual wallets are often implemented via proxy where base functionality is implemented in a blueprint contract, that is called by the proxy. This dataset consists of proxies for the wallets in the wallet dataset.

This dataset is interesting because the codes are extremely short, meaning half the code is data and arguments.

7.4 Small Groups with the same Name and ABI

A sample of contracts with verified source available on etherscan.io, grouped by same name and ABI signatures. The dataset is comprised of 89 groups. The groups contain 5 to 10 contracts with the same ABI interface and the same name. The groups have distinct ABI interfaces and distinct names. In total there are 541 codes with distinct skeletons and between 15 to 25 interface functions.

8 Evaluation Framework

To compare the similarity measures and evaluate there efficacy a package including test-sets, similarity-measures, python utils for exploration and evaluate was created *ethereum-contract-similarity*[5].

9 Digest Methods

9.1 Fourbytes - macro-similarity

The interface of a contract is the set of functions callable by other accounts/contracts. Almost all deployed contracts follow the ABI standard for encoding there interface, which lets the caller select the desired function via a four-byte hash of the function signature. Contracts can be associated via interface by calculating the Jaccard-Index on there sets of fourbyte signatures.

The macro-similarity based on the contract-interface is described in 'II.C.2) Interface Restoration' (Di Angelo et al. 2020a [2])

9.2 ssdeep

ssdeep is a Context Triggered Piecewise Hash (CTPH) based on *spamsum*[11] which was written for email spam detection. It is described in detail in the accompanying paper (Kornblum 2006 [4])

Context Triggered Piecewise Hashes follow the steps:

1. Compute a rolling hash of the last n bytes for every position.
2. The rolling hash is used to determine the cutoff points.
3. The resulting chunks are hashed using a traditional cryptographic hash.
4. The final hash results from concatenating part of the chunk-hashes (e.g. the last byte).

9.3 ppdeep

To make modifications easy I used a version of ssdeep implemented in pure python *ppdeep*[12]. The score calculation differs from the original implementation, but scatter plots showed insignificant impact for the purposes of this work.

The following modifications where made:

- Remove sequence-stripping. It made many hashes incomparable because of long strips of 'K' chunk-hashes.
- Remove rounding in the score calculation to differentiate between exact match and close match as well as incomparable and minor similarity.
- Remove common substring detection to make more hashes comparable.
- Handle case where first chunk is never triggered.
- Add option to use Jaccard-Index for comparison, the default is Levenshtein-similarity.

9.4 jumpHash

Inspired by ssdeep and the learnings from the *solc-versions-testset*[6] I implemented jumpHash [5], it follows the steps:

1. Split the code by the opcode `JUMPI=0x57` into chunks.
2. Hash each chunk with `sha1`.
3. Map the first byte of the `sha1` hash to a Unicode character.
4. Concatenate the Unicode characters to a hash-string.
5. Compare the hash-strings via Levenshtein-similarity.

```
from hashlib import sha1

def h(b: bytes) -> str:
  return chr(sha1(b).digest()[0] + 0xb0)

def hash(code: bytes) -> str:
  jumpi = b'\x57'
  chunks = code.split(jumpi)
  return ''.join(h(chunk) for chunk in chunks)
```

The Unicode character `°=chr(0xb0)` was chosen as `0` chunk hash because it is followed by 255 valid characters.

9.5 Bytebag - Opcode Frequency

As lower reference bound for more complex similarity detection I implemented the following measure:

1. Count every byte-value in the code forming a multiset or bag of byte-values, a bytebag.
2. Compare via Jaccard-Index for bags.

```
def byteBag(code: bytes) -> Dict[int,int]:
  def reducer(counts: Dict[int,int], b: int):
    counts[b] = counts.get(b, 0) + 1
    return counts
  return functools.reduce(reducer, code, {})

def jaccard(a: Dict[int,int], b: Dict[int,int]) -> float:
  return sum(min(a[i], b[i]) for i in range(256)) /
    sum(max(a[i], b[i]) for i in range(256))
```

9.6 LZJD - Lempel-Ziv Jaccard Distance

Designed as a fast approximation of the Normalized Compression Distance (*NCD*) [8], Lempel-Ziv Jaccard Distance (*LZJD*) can be used as alternative to *sdhash* and *ssdeep* [9].

The digest generated by *LZJD* (*LZSet*) is a LZ dictionary, which is a set of sub-sequences, defined by Algorithm 1 in [8]. Distance is calculated via the Jaccard-Index (1) on the *LZSets* (3).

$$LZJD(x, y) = 1 - J(LZSet(x), LZSet(y)) \tag{3}$$

The pyLZJD implementation was used [7].

9.7 bzHash

bzHash is based on peHash[13] which calculates the compression ratio for each section of potentially malicious executables to generate a fingerprint.

1. Split code by `JUMPI` into chunks.
2. For each chunk: calculate the bz-compression-ratio.
3. Calculate mean and standard-deviation of the ratios.
4. Calculate distance from mean as multiple of the standard-deviation.
5. Clamp values to [-2, 2] standard-deviations from mean.
6. Discretize to positive integer values.
7. Map the integers to characters.
8. Concatenate the characters to the final hash-string.
9. Compare the hash-strings with Levenshtein.

```
import numpy as np
import bz2

def bzCompRatio(bs: bytes):
  return len(bz2.compress(bs)) / len(bs) if len(bs) > 0 else 1

_map = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz-_'

def bzHash(bs:bytes, chunkRes=4) -> str:
  jumpi = b'\x57'
  res = [bzCompRatio(chunk) for chunk in bs.split(jumpi)]
  res = sdFromMean(res)
  return ''.join(_map[discretize(-2, 2, chunkRes, val)] for val in res)

def discretize(mi, ma, resolution, val):
  span = ma - mi
  shifted = val - mi
  scaled = shifted * (resolution / span)
  return max(0, min(resolution - 1, int(scaled)))

def sdFromMean(x: Iterable) -> Iterable:
  x = np.fromiter(x, float)
  if len(x) == 0:
    return x
  elif x.std() == 0:
    return x - x
  return (x - x.mean()) / x.std()
```

9.8 Normalized compression distance NCD

NCD is a measure for how well two files co-compress.

The more features two files have in common the shorter the length of the compressed concatenation.

$Z(x)$ is the length of the compressed file x ; xy is the concatenation of x and y.

$$NCD(x,y) = \frac{Z(xy) - \min\{Z(x), Z(y)\}}{\max\{Z(x), Z(y)\}} \tag{4}$$

I calculation the similarity instead of the distance to be consistent with the other measures.

```
import lzma

def Z(contents: bytes) -> int:
  return len(lzma.compress(contents, format=lzma.FORMAT_RAW))

def NCD(a: bytes, b: bytes):
  return (Z(a + b) - min(Z(a), Z(b))) / max(Z(a), Z(b))

def similarity(a: bytes, b: bytes):
  return (Z(a) + Z(b) - Z(a + b)) / max(Z(a), Z(b))
```

10 Results

10.1 Test Specification

Violin Plots

Unless otherwise specified the violin plots [Fig. 1] in this work all adhere to the following specification.

- They show the similarity-sores of all possible code-pairs by on measure.

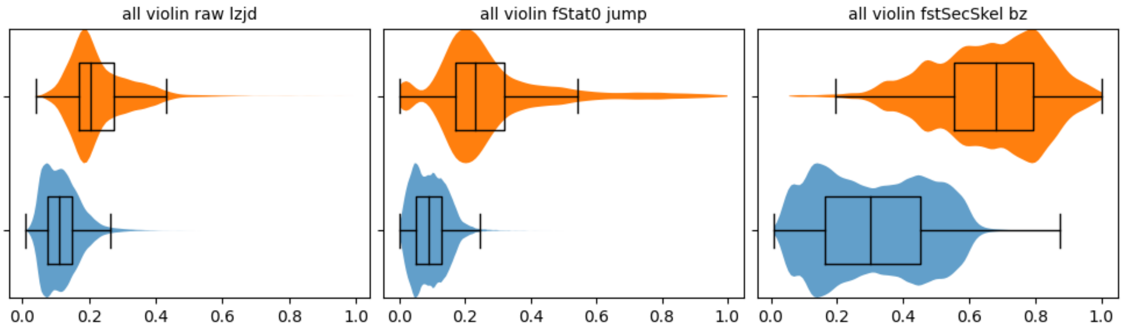


Fig. 1: violin plots

- The measure (e.g. lzjd) and the pre-processing (e.g. fStat0) is specified in the title.
- The pairs are partitioned into "same" (top violin) and "cross" (bottom violin).
- "same" contains all pairs where the two codes are from the same group.
- "cross" contains all pairs where the two codes are from different groups.
- The x-axis is the similarity-score ∈ [0,1].
- The box plots show the first, second and third quartile; the whiskers have a length of 1.5 times the inter-quartile-range.

Histograms
Unless otherwise specified the histograms [Fig. 2] in this work all adhere to the following specification.

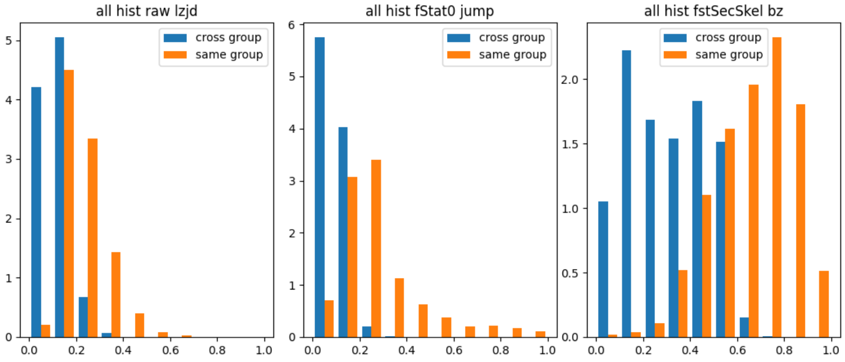


Fig. 2: histograms

- They show the similarity-sores of all possible code-pairs by on measure.
- The measure (e.g. lzjd) and the pre-processing (e.g. fStat0) is specified in the title.
- The pairs are partitioned into "cross group" and "same group".
- "cross group" contains all pairs where the two codes are from different groups.
- "same group" contains all pairs where the two codes are from the same group.
- The x-axis is the similarity-score ∈ [0,1].
- The y-axis is the density, i.e. the sum of the ten buckets is always ten.
- The "same group" and "cross group" series are plotted separately, i.e. all ten buckets add up to ten for both series separately.

Pre-Processing
I tested with the following pre-processing settings.

raw Tho whole code unprocessed.
fstSec or firstSection The first code section.
skel or skeleton The skeleton of the whole code.
fstSecSkel The skeleton of the fist code section.

fStat fstSecSkel filtered for the opcodes in the `fStat` filter by cutting the others out.
fStat0 fStat but instead of cutting out the other opcodes are set to zero.
fStatV2 fStat with a few opcodes removed.
fStat0V2 fStat0 with a few opcodes removed.

Separation

To quickly evaluate how well the methods distinguish between code-pairs with both codes from the same group and pairs of codes from different groups the `separation` was defined. It specifies the following ratio. When the pairs are ordered by similarity-score, how many same-group pairs are in the upper window of size total number of same-group pairs.

qDist

As a second measure for the clustering-performance I calculated $qDist(s, c)$ defined as follows.

s ... Ordered similarity-scores of all code pairs where the codes are from the same group.
c ... Ordered similarity-scores of all code pairs where the codes are from different groups.
Q_n^x ... Nth quartile of x
Q_2^x ... Median of x.

$$qDist(x, y) = \frac{Q_2^x - Q_2^y}{Q_2^x - Q_1^x + Q_3^y - Q_2^y} \tag{5}$$

10.2 Run: solc versions clustered with bytebag

Data

An older version of the *solc-versions-testset*[6], commit 923fa5bb5abb2939e54fb5404e9e287b1f0f0cec. Nine Solidity source files compiled with four solc versions (5, 6, 7, 8) times three optimization settings (no optimization, runs = 1, runs = 999999). Necessary changes where made to the source code to ensure compatibility with the different solc version without altering the function of the contracts.

Pre-Processing

The runtime codes where segmented and the first segment skeletonized and then filtered with the following opcode filter. I defined the filter based on my intuition, for which opcodes are hard to be replaced by others.

```
(OP.is_log() or OP.is_storage() or OP.is_sys_op() or OP.is_env_info()
  or OP.is_block_info() or OP == opcodes.SHA3 or OP == opcodes.GAS)
```

```
# GASPRICE CALLER SSTORE TIMESTAMP LOG3 ORIGIN INVALID LOG2 ADDRESS BLOCKHASH STATICCALL
    CALLDATALOAD CALL CALLDATASIZE RETURNDATASIZE EXTCODEHASH GAS LOG0 DIFFICULTY CODESIZE
    DELEGATECALL CALLVALUE RETURNDATACOPY RETURN NUMBER SELFDESTRUCT CALLCODE REVERT
    CALLDATACOPY COINBASE EXTCODESIZE CODECOPY CREATE SHA3 LOG4 SLOAD EXTCODECOPY GASLIMIT
    CREATE2 LOG1 BALANCE
```

Similarity

Similarity for all code pairs was calculated via Jaccard Index on the bytebags of the filtered codes.

Clustering

The clustering [Fig. 3a] was done in Gephi (0.9.2) with the settings [Fig. 3b]. Nodes with the same color have the same source code. The graph is fully connected and the edge weights are determined by the similarity scores of the pairs.

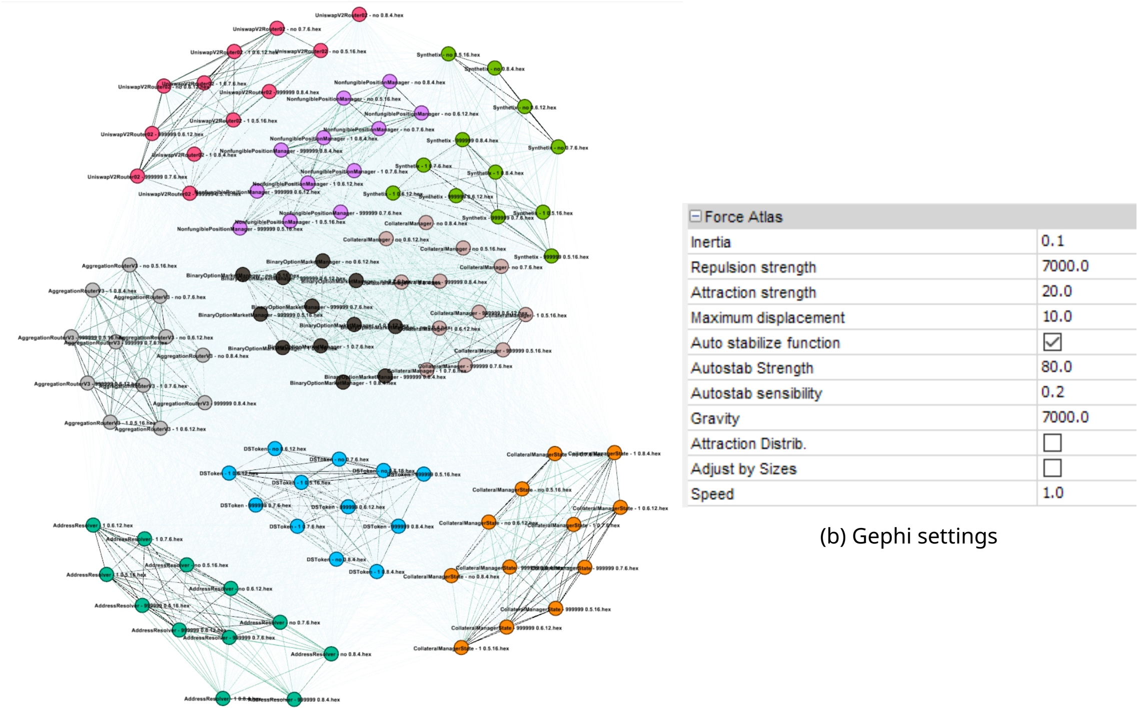
Observation

The clustering algo formed cohesive clusters, mostly due to the fact that the dataset is small and the contracts differ significantly in size alone. Nonetheless, noticeable is that the Synthetix codes without optimization from all 4 solc versions form a cluster separate from the other Synthetix codes [Fig. 4a]. The no optimization variants in the CollateralManagerState cluster also group tightly [Fig. 4a].

Analysis

When comparing the frequency of the opcodes in `Synthetix - no 0.8.4.hex` and `Synthetix - 999999 0.8.4.hex` these 7 op-codes show the highest change [Tbl. 2].

`54 0x36 CALLDATASIZE` is the most consistent across solc versions and optimization settings.

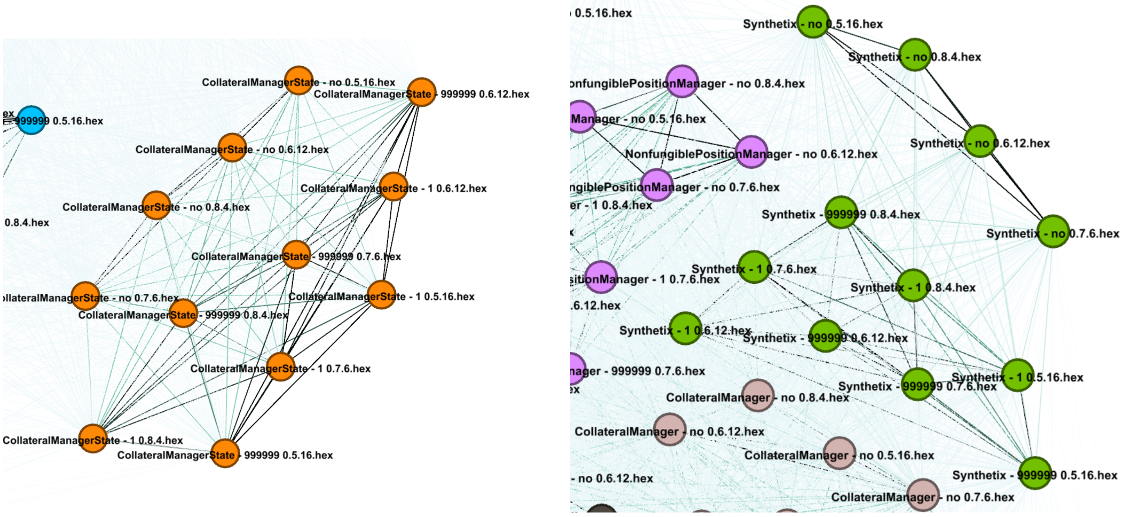


(a) clustering

⊟ Force Atlas	
Inertia	0.1
Repulsion strength	7000.0
Attraction strength	20.0
Maximum displacement	10.0
Auto stabilize function	☑
Autostab Strength	80.0
Autostab sensibility	0.2
Gravity	7000.0
Attraction Distrib.	□
Adjust by Sizes	□
Speed	1.0

(b) Gephi settings

Fig. 3: bytebag solc-versions-testset


CollateralManagerState - no 0.5.16.hex
CollateralManagerState - 999999 0.6.12.hex
CollateralManagerState - no 0.6.12.hex
CollateralManagerState - 1 0.6.12.hex
CollateralManagerState - no 0.8.4.hex
CollateralManagerState - 999999 0.7.6.hex
CollateralManagerState - no 0.7.6.hex
CollateralManagerState - 1 0.5.16.hex
CollateralManagerState - 999999 0.8.4.hex
CollateralManagerState - 1 0.7.6.hex
CollateralManagerState - 1 0.8.4.hex
CollateralManagerState - 999999 0.5.16.hex
Synthetix - no 0.5.16.hex
Synthetix - no 0.8.4.hex
NonfungiblePositionManager - no 0.8.4.hex
NonfungiblePositionManager - no 0.6.12.hex
Synthetix - no 0.6.12.hex
Synthetix - 999999 0.8.4.hex
Synthetix - no 0.7.6.hex
Synthetix - 1 0.7.6.hex
Synthetix - 1 0.8.4.hex
Synthetix - 1 0.6.12.hex
Synthetix - 999999 0.6.12.hex
Synthetix - 999999 0.7.6.hex
Synthetix - 1 0.5.16.hex
CollateralManager - no 0.8.4.hex
CollateralManager - no 0.6.12.hex
CollateralManager - no 0.5.16.hex
Synthetix - 999999 0.5.16.hex
CollateralManager - no 0.7.6.hex

(a) CollateralManagerState

(b) Synthetix

Fig. 4: no optimization

dec	hex	op-code	no 0.8.4	999999 0.8.4	diff
53	`0x35`	CALLDATALOAD	6	41	35
59	`0x3B`	CODESIZE	65	46	-19
61	`0x3D`	RETURNDATASIZE	171	121	-50
62	`0x3E`	RETURNDATACOPY	60	46	-14
90	`0x5A`	GAS	65	45	-20
243	`0xF3`	RETURN	45	1	-44
253	`0xFD`	REVERT	204	156	-48

Tbl. 2: optimization differences

`59 0x3B EXTCODESIZE` is also very consistent.

90 0x5A GAS has a high absolute difference between the Synthetix contracts but it has higher differences across contracts.

Interpretation
Many of the opcodes, I thought significant show hight changes in frequency when optimization is applied, especially the RETURN opcode should not be included in the filter, since optimization drastically reduces its prevalence from 45 to 1. Manually analysis of the assemblies showed that optimization options change the codes more than solc version changes.

10.3 Run: Determine significant opcodes

Because my intuition was wrong a quantitative method for determining significant opcodes was needed.

Idea: Find bytes that change more between groups than in groups between codes i.e. bytes with higher between group variance than in group variance. → f-statistic ANOVA

Data
An older version of the *solc-versions-testset*[6] was used.
(commit: ec2b5c957c96a107926d64f469264be043c97d3b)

Method
Calculate and f-statistic value for every byte value from 0 to 255, where codes complied from the same source with different solc-versions and -options form the groups.

Results
ethereum-contract-similarity/runs/byteDistribution/out/f-stat-by-byte.csv
commit 51864956b37231e27effcdc1dc17e842eee2078d

Observations
RETURN has the lowest score of 1.3 confirming the interpretation of the clustering test-run.

JUMPI has a very high score of 489.5 despite its high prevalence of on average 207.5 per contract. It has the 20th highest f-statistic and ist the 13th most prevalent. It also has the sixth highest minimum of 32 occurrences in one contract. Opcodes scoring higher are much less frequent at a max mean of 24.1. In oder of f-statistic ADD is the next op-code with higher prevalence at rank 38 (f-stat 171.9, mean count 408.9).

ISZERO looks surprisingly significant at a f-statistic of 489.5 and a mean count of 207.5

Interpretation
In this dataset the ABI encodings where mostly consistent within groups and they contain a high number of JUMPI ops, but the ABI section also contains an equal number of PUSH4, DUP1 and EQ, which have lower f-stat values of 224.6, 57.6 and 73.5. And the rest of the code does also contain a high number of JUMPI.

The ISZERO opcodes are related to JUMPI since conditional jumps of often implemented by combining these two operations. ISZERO might be more relevant since it is not used in the ABI section.

A dataset where the ABI tables are cut off could be used, since ABI-similarity can be better obtained with other methods (fourbytes).

Conclusion
I extended the dataset with fixed ABI encodings v1 and v2 for all contracts, because the default changed with solc version 0.8.0. Based on the recalculated scores the fStat filter was defined.

10.4 Misc.

$\mathcal{H}$ Splitting by JUMPI achieves better results than the context trigger used by *ssdeep*.

$\mathcal{H}$ Filtering opcodes with *fStat* improves the results for chunk-hashes and *bytebag*.

$\mathcal{H}$ Chunk-hashes perform better on the skeleton of the first code section than on the whole code.

$\mathcal{H}$ Enabling optimization when compiling with *solc* causes bigger changes to the codes than switching *solc* version.

$\mathcal{H}$ *Bytebag* is simple and fast, jet it seams to be a useful way to determine similarity.

$\mathcal{H}$ Removing sequence striping improves results with *ppdeep*.

$\mathcal{H}$ Levenshtein works better than Jaccard with chunk hashes, because *solc* orders functions by signature in the runtime code.

$\mathcal{H}$ *ncd* is very flexible, it deals with the very short proxy codes and handles the biggest compile changes better than the other methods.

$\mathcal{O}$ *ncd* comparison is very slow, because all code-pairs need to be compressed together, i.e. *ncd* comparison is performed on the full codes and the other methods compare the much smaller digests, which are calculated only once for each code.

	Measure	Separation
1	raw fourbytes	99 %
2	fStat jump	80 %
3	fStat0 jump	75 %
4	fStat0 bz	74 %
5	fstSecSkel bz	74 %
6	fstSecSkel jump	74 %
7	skeletons jump	72 %
8	skeletons bz	69 %
9	fStat bz	68 %
10	fStat byteBag	68 %
11	fStat0 byteBag	68 %
12	raw bz	67 %
13	fstSecSkel byteBag	64 %
14	skeletons byteBag	62 %
15	raw lzjd	61 %
16	fstSecSkel ppdeep_mod	54 %
17	skeletons ppdeep_mod	54 %
18	fstSecSkel lzjd	53 %
19	skeletons lzjd	52 %
20	raw byteBag	51 %
21	fStat ppdeep_mod	44 %
22	raw ppdeep_mod	42 %
23	fStat0 lzjd	40 %
24	fStat ssdeep	36 %
25	fStat ppdeep	36 %
26	fStat0 ssdeep	34 %
27	fStat0 ppdeep	34 %
28	fstSecSkel ssdeep	33 %
29	fstSecSkel ppdeep	33 %
30	skeletons ssdeep	32 %
31	skeletons ppdeep	32 %
32	fStat0 ppdeep_mod	31 %
33	raw ssdeep	31 %
34	raw ppdeep	31 %
35	skeletons fourbytes	30 %
36	fstSecSkel fourbytes	30 %
37	fStat fourbytes	30 %
38	fStat0 fourbytes	30 %
39	raw jump	25 %
40	fStat lzjd	13 %

	Measure	qDist
1	raw fourbytes	400 %
2	fStat jump	161 %
3	fStat0 jump	143 %
4	fstSecSkel jump	141 %
5	fStat0 bz	138 %
6	skeletons jump	137 %
7	fStat byteBag	137 %
8	fStat0 byteBag	137 %
9	fstSecSkel bz	136 %
10	fstSecSkel byteBag	129 %
11	fStat bz	127 %
12	raw lzjd	124 %
13	raw byteBag	120 %
14	skeletons byteBag	120 %
15	skeletons bz	117 %
16	raw bz	116 %
17	skeletons ppdeep_mod	106 %
18	fstSecSkel ppdeep_mod	105 %
19	fstSecSkel lzjd	103 %
20	skeletons lzjd	100 %
21	raw ppdeep_mod	87 %
22	fStat ppdeep_mod	87 %
23	fStat0 lzjd	76 %
24	fStat0 ppdeep_mod	49 %
25	raw jump	0 %
26	raw ssdeep	0 %
27	raw ppdeep	0 %
28	skeletons ssdeep	0 %
29	skeletons ppdeep	0 %
30	skeletons fourbytes	0 %
31	fstSecSkel ssdeep	0 %
32	fstSecSkel ppdeep	0 %
33	fstSecSkel fourbytes	0 %
34	fStat ssdeep	0 %
35	fStat ppdeep	0 %
36	fStat fourbytes	0 %
37	fStat0 ssdeep	0 %
38	fStat0 ppdeep	0 %
39	fStat0 fourbytes	0 %
40	fStat lzjd	−1 %

Tbl. 3: *wallet* dataset *separations* and *qDists*

10.5 LZJD

$\mathcal{H}$ *lzjd* struggles on the full *solc-versions-testset*[6] with all hash-size settings [Fig. 5] [Fig. 6]. As reference on the right side *ncd* and *bz* seam to perform better on the *raw* codes.

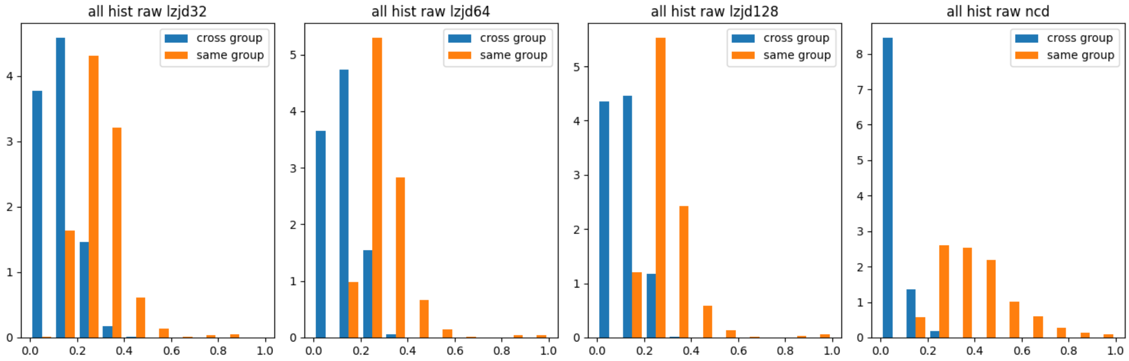


Fig. 5: *lzjd[32,64,128]* on all *raw solc-versions-testset* codes

	Measure	Separation
1	raw ncd	89 %
2	raw bz	75 %
3	fstSecSkel ncd	75 %
4	skeletons ncd	75 %
5	skeletons jump	75 %
6	fstSecSkel jump	75 %
7	raw lzjd	72 %
8	skeletons bz	72 %
9	fstSecSkel bz	70 %
10	fstSecSkel lzjd	58 %
11	skeletons lzjd	58 %
12	skeletons byteBag	53 %
13	fstSecSkel byteBag	53 %
14	fstSecSkel ppdeep_mod	51 %
15	skeletons ppdeep_mod	50 %
16	raw byteBag	46 %
17	raw ppdeep_mod	38 %
18	fstSecSkel ppdeep	32 %
19	skeletons ppdeep	32 %
20	skeletons ssdeep	32 %
21	fstSecSkel ssdeep	32 %
22	raw jump	30 %
23	raw ssdeep	12 %
24	raw ppdeep	12 %

	Measure	qDist
1	raw ncd	279 %
2	fstSecSkel ncd	182 %
3	skeletons ncd	182 %
4	fstSecSkel jump	164 %
5	skeletons jump	163 %
6	raw lzjd	158 %
7	raw bz	148 %
8	skeletons bz	142 %
9	fstSecSkel bz	138 %
10	skeletons lzjd	137 %
11	fstSecSkel lzjd	134 %
12	skeletons byteBag	126 %
13	fstSecSkel byteBag	126 %
14	raw byteBag	116 %
15	raw ppdeep_mod	106 %
16	fstSecSkel ppdeep_mod	102 %
17	skeletons ppdeep_mod	101 %
18	raw ssdeep	0 %
19	raw ppdeep	0 %
20	skeletons ssdeep	0 %
21	skeletons ppdeep	0 %
22	fstSecSkel ssdeep	0 %
23	fstSecSkel ppdeep	0 %
24	raw jump	−12 %

Tbl. 4: *solc-versions-testset separations* and *qDists*

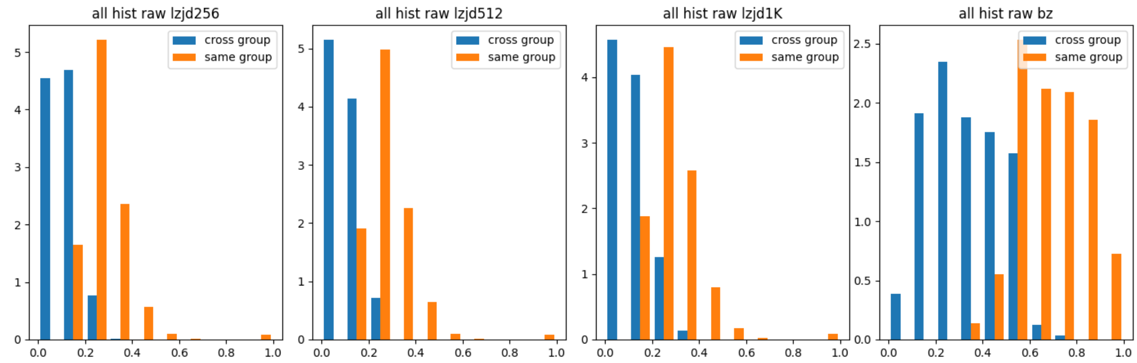


Fig. 6: *lzjd[256,512,1K]* on all *raw solc-versions-testset* codes

$\mathcal{H}$ *lzjd* works better with *raw* codes and 256 is a good default *hash_size* setting [Tbl. 5] [Tbl. 6] [Fig. 7].

	Measure	Separation
1	raw lzjd256	72 %
2	raw lzjd512	72 %
3	raw lzjd128	70 %
4	raw lzjd64	64 %
5	skel lzjd32	59 %
6	skel lzjd256	58 %
7	raw lzjd32	57 %
8	skel lzjd64	57 %
9	skel lzjd512	56 %
10	skel lzjd128	56 %
11	raw lzjd1K	55 %
12	skel lzjd1K	50 %

	Measure	qDist
1	raw lzjd256	158 %
2	raw lzjd512	157 %
3	raw lzjd128	157 %
4	raw lzjd64	156 %
5	raw lzjd1K	148 %
6	skel lzjd1K	143 %
7	skel lzjd64	141 %
8	raw lzjd32	139 %
9	skel lzjd512	138 %
10	skel lzjd256	134 %
11	skel lzjd32	133 %
12	skel lzjd128	131 %

Tbl. 5: *lzjd separations* and *qDists* on all codes from *solc-versions-testset*

	Measure	Separation
1	raw lzjd	61 %
2	fstSecSkel lzjd	53 %
3	skeletons lzjd	52 %
4	fStat0 lzjd	40 %
5	fStat lzjd	13 %

	Measure	qDist
1	raw lzjd	124 %
2	fstSecSkel lzjd	103 %
3	skeletons lzjd	100 %
4	fStat0 lzjd	76 %
5	fStat lzjd	−1 %

Tbl. 6: *lzjd separations* and *qDists* on all *wallet* codes

11 Remarks

solc

ABI v2 moves the majority of the interface code from the start to the end of the code compared to v1.

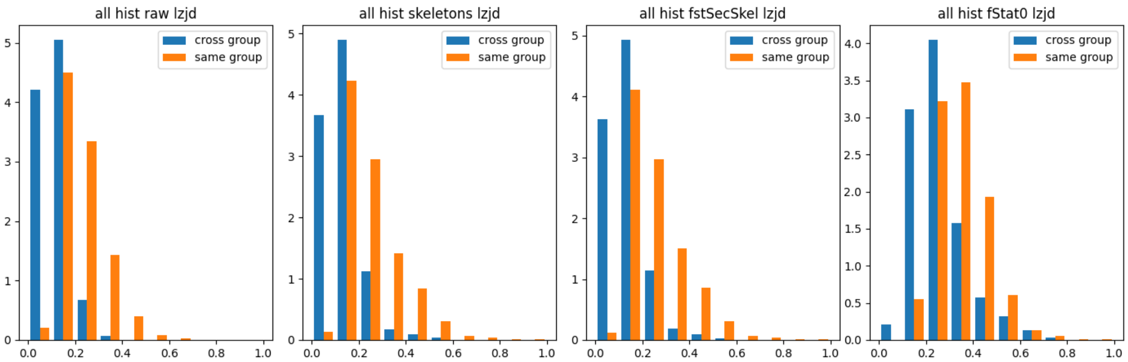


Fig. 7: *lzjd[256]* on *wallets* dataset with different pre-processing methods

Optimization changes the how the ABI jump table is realized, solely causing significant changes for domain independent similarity measures.

Optimizations with high runs settings lead to a heavy reliance on storage operations, and causes a dramatically increase in overall code length.

Version changes are comparably smaller, but the default ABI encoding changed from v1 to v2 with solc version 0.8.0.

jumpHash
Considering its simplicity it performs surprisingly well in separating contracts from different groups, partially due to the fact that the number of JUMPI opcodes has a very high f-statistic value.

It correlates strongly with NCD, which seams to be more robust to optimization changes, but comparisons take 30 times longer and jumpHash separates groups more sharply.

The nativ Levenshtein implementation used for comparison is the fastest out of all hash similarities used in this work. Only Jaccard applied to the much shorter Fourbyte signature sets is faster.

12 Conclusion

next steps

goals for followup papers

References

[1] Monika Di Angelo and Gernot Salzer. “Wallet Contracts on Ethereum”. In: *CoRR* abs/2001.06909 (2020). arXiv: `2001.06909`. URL: `https://arxiv.org/abs/2001.06909`.

[2] Monika Di Angelo and Gernot Salzer. “Characteristics of Wallet Contracts on Ethereum”. In: *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE. 2020, pp. 232–239.

[3] Monika Di Angelo and Gernot Salzer. “Characterizing Types of Smart Contracts in the Ethereum Landscape”. In: Aug. 2020, pp. 389–404. ISBN: 978-3-030-54454-6. DOI: `10.1007/978-3-030-54455-3_28`.

[4] Jesse Kornblum. “Identifying almost identical files using context triggered piecewise hashing”. In: *Digital investigation* 3 (2006), pp. 91–97.

[5] Raphael Nußbaumer. *ethereum-contract-similarity*. 2022. URL: `https://github.com/mrNuTz/ethereum-contract-similarity` (visited on 03/07/2022).

[6] Raphael Nußbaumer. *solc-versions-testset*. 2022. URL: `https://github.com/mrNuTz/solc-versions-testset` (visited on 03/10/2022).

[7] Edward Raff, Joe Aurelio, and Charles Nicholas. “PyLZJD: an easy to use tool for machine learning”. In: *UMBC Faculty Collection* (2019). URL: `http://hdl.handle.net/11603/14971`.

[8] Edward Raff and Charles Nicholas. "An Alternative to NCD for Large Sequences, Lempel-Ziv Jaccard Distance". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '17. Halifax, NS, Canada: ACM, 2017, pp. 1007–1015. ISBN: 978-1-4503-4887-4. DOI: `10.1145/3097983.3098111`. URL: `http://doi.acm.org/10.1145/3097983.3098111`.

[9] Edward Raff and Charles K. Nicholas. "Lempel-Ziv Jaccard Distance, an Effective Alternative to Ssdeep and Sdhash". In: *CoRR* abs/1708.03346 (2017). arXiv: `1708.03346`. URL: `http://arxiv.org/abs/1708.03346`.

[10] Gernot Salzer. *ethutils: Utilities for the Analysis of Ethereum Smart Contracts*. 2017. URL: `https://github.com/gsalzer/ethutils` (visited on 03/10/2022).

[11] Andrew Tridgell. *Spamsum*. 2002. URL: `http://samba.org/ftp/unpacked/junkcode/spamsum/README` (visited on 03/09/2022).

[12] Marcin Ulikowski. *Pure-Python library for computing fuzzy hashes (ssdeep)*. 2020. URL: `https://github.com/elceef/ppdeep` (visited on 03/10/2022).

[13] Georg Wicherski. "peHash: A Novel Approach to Fast Malware Clustering." In: *LEET* 9 (2009), p. 8.