Fuzzy Hashing of Ethereum Smart Contracts

Bachelor's Thesis in Software and Information Engineering

April 26, 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 the 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.

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 and have an Ether balance.

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 Contracts 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.

Message 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.

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 commonly used programming language for writing Ethereum smart contracts.

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 observed runtime behavior, e.g. transactions, messages, contract creation and other temporal associations between accounts. Static analysis is concerned with static properties of the data stored on the EVM.

This work focuses 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 the runtime code. To find associations between runtime codes one can exactly match code skeletons [Sec. 4.2] and extract the fourbyte signatures [Sec. 5.1] 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 of 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, we look at the landscape of fuzzy hashing functions.

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

To obtain similarity scores, we preprocessed codes, calculated digests (hashes) and compared those digest via similarity measures.

4. Pre-Processing

What runtime codes do, is firstly they get external data through function arguments, blockchain data or calling of other contracts, secondly they prepare data on the stack, thirdly they use that data to cause an externally observable effect like storing, logging, transacting, self destruction, calling contracts and deploying contracts, and fourthly they repeat one, two and three.

Small changes in the codes can cause big changes in the digests—pre-processing aims to remove non-essential parts.

4.1 Segmentation

Segmentation splits the codes into `code`, `data` and `meta` sections [11].

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

`data` sections are e.g. constructor arguments. Constructor-arguments are processed by the deployment code. Generally they are 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.

4.2 Skeletonization

Skeletons are runtime codes where constructor arguments, data sections, meta sections and push arguments are set to zero [11]. 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.

4.3 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.

4.3.a *fStat* Filter

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 ones that do not change, we used the *solc-versions-testset* [Sec. 7.1] and reduced the codes to *bytebags* [Sec. 5.6]. Then, we calculated an the *One-way ANOVA F-test statistic* (1) for each opcode, using common source-code as grouping criterion and the count of the opcode as value.

To define the *fStat* filter, we selected the top 30 opcodes by f-statistic value, plus a few that never occurred or only in one group. See [Sec. 9.2] for the f-statistic values and more details.

$$F = \frac{\textit{between group variability}}{\textit{within group variability}} \tag{1}$$

```
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,
)
```

5. Digest Methods

5.1 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 their interface, which lets the caller select the desired function via a four-byte hash of the function signature. The macro-similarity based on the contract-interface is described in *II.C.2) Interface Restoration* (Di Angelo et al. 2020a [2]).

The digest produced by this method is the set of extracted fourbyte signatures. Similarity scores are obtained via the Jaccard-Index [Sec. 6.1] of these sets.

5.2 ssdeep

ssdeep is a Context Triggered Piecewise Hash (CTPH) based on *spamsum*[12] which was written for email spam detection. It is described in detail in the accompanying paper (Kornblum 2006 [5])

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).

ssdeep compares the hashes via a custom edit-distance and sores similarity from 0 to 100, we divide by 100 to get scores in [0,1].

5.3 ppdeep

To make modifications easy, we used a version of ssdeep implemented in pure Python *ppdeep*[13]. The score calculation differs from the original implementation, but scatter plots [Fig. 9] showed insignificant impact for the purposes of this work. *ppdeep* hashes are compared via Levenshtein distance [Sec. 6.2] and custom weighting to produce a similarity scores in [0,100], which we divide by 100 to get scores in [0,1].

5.4 ppdeep_mod

The following modifications where made to *ppdeep*:

- 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 the case where the first chunk is never triggered.
- Add an option to use the Jaccard-Index for comparison, the default is Levenshtein-similarity.

5.5 jumpHash

Inspired by ssdeep and the lessons from the *solc-versions-testset*[7], we implemented jumpHash [6], which works as follows:

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 [Sec. 6.2].

```
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.

5.6 Bytebag - Opcode Frequency

As lower reference bound for more complex similarity detection, we 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 (code below).

```
def byteBag(code: bytes) -> Dict[int,int]:
  def reducer(counts: Dict[int,int], b: int):
    counts[b] = counts[b] + 1 if b in counts else 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))
```

5.7 LZJD - Lempel-Ziv Jaccard Distance

Designed as a fast approximation of the Normalized Compression Distance (*NCD*) [9], Lempel-Ziv Jaccard Distance (*LZJD*) can be used as alternative to *sdhash* and *ssdeep* [10].

The digest generated by *LZJD* (*LZSet*) is an LZ dictionary, which is a set of sub-sequences, defined by *Algorithm 1* in [9]. Distance is calculated via the Jaccard-Index [Sec. 6.1] on the *LZSets* (2).

$$LZJD(x, y) = 1 - J(LZSet(x), LZSet(y)) \tag{2}$$

We used the *pyLZJD*[8] implementation and calculated similarity instead of distance.

5.8 bzHash

bzHash is based on peHash[14] which calculates the compression ratio for each section of potentially malicious executables to generate a fingerprint. The digest is calculated via the following steps.

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 [Sec. 6.2].

```
import numpy as np
import bz2

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

_map = ’0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz-_’

def bzHash(code:bytes, chunkRes=4) -> str:
  jumpi = b’\x57’
  res = [bzCompRatio(chunk) for chunk in code.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()
```

6. Similarity Measures

The following methods where used to obtain similarity scores, applied to digests and undigested codes.

6.1 Jaccard-Index

Defined on two sets *A* and *B*, the Jaccard-Index *J* is the ratio of the cardinalities intersection over union. Can be applied to bags (multisets) via the same definition using intersection and union operations for bags.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \in [0,1] \tag{3}$$

Jaccard was used to compare the digest of LZJD [Sec. 5.7], Fourbyte [Sec. 5.1] and Bytebag [Sec. 5.6].

6.2 Levenshtein Distance

It is also called edit distance, and is defined to be the minimum number of character-inserts, -deletions and -substitutions necessary to change one string into the other.

To be consistent with the other measures, we calculated the Levenshtein similarity as follows:

$$similarity(a,b) = 1 - \frac{distance(a,b)}{\max\{|a|,|b|\}} \in [0,1] \tag{4}$$

We used the *pyPI* package *python-Levenshtein*[4].

The digests of *ppdeep* [Sec. 5.3], *bzHash* [Sec. 5.8] and *jumpHash* [Sec. 5.5] where compared via Levenshtein similarity. We also applied it on whole codes from the *proxies* dataset [Sec. 7.3]; the codes from the other datasets are too long.

6.3 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.

$$NCD(x,y) = \frac{Z(xy) - \min\{Z(x), Z(y)\}}{\max\{Z(x), Z(y)\}} \tag{5}$$

We calculated 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))

# Simplified Python code: Z(a) is calculated of all codes a before comparing
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))
```

We used *NCD* to calculate the similarity between whole codes.

6.4 Code Length

To test if the other methods are just an expensive proxy for length comparison, we obtained a similarity score purely based on the size of the codes.

$$sim(a,b) = \frac{\min\{|a|,|b|\}}{\max\{|a|,|b|\}} \in [0,1] \tag{6}$$

7. Data sets

7.1 *solc-versions-testset*

To evaluate the similarity measures, we selected a set of 13 solidity smart contracts and compiled them with different solc versions and compiler options. Necessary changes where made to the source code to ensure compatibility with the various solc versions. The dateset and code is available online [7].

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-settings `{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-settings x 2 abi-encodings ≈ 264 codes.

7.1.a 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, we 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

An 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 is 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 very short, with half of the code being data and arguments.

7.4 Small Groups with the same Name and Fourbyte Signatures

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. Results Specification

8.1 Evaluation Framework

The majority of the effort went into creating an Evaluation Framework (*ethereum-contract-similarity*[6]) in Python. The idea was to allow for quick comparison of different datasets, pre-processors, pre-processing parameters, digests, digest-parameters, similarity-measures and similarity-parameters as well as plotting and statistical exploration of results.

8.2 Pre-Processing

We tested with the following pre-processing settings.

raw Tho whole code unprocessed.
fstSec or firstSection The first code section [Sec. 4.1].
skel or skeleton The skeleton of the whole code [Sec. 4.2].

fstSecSkel The skeleton of the fist code section.
fStat *fstSecSkel* filtered for the opcodes in the *fStat* filter by cutting out the other opcodes [Sec. 4.3.a].
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.

8.3 Digest and Similarity Methods

The keywords used in the following sections to identify the pairs of digest and similarity method.

fourbytes Set of fourbyte signatures [Sec. 5.1] scored via Jaccard [Sec. 6.1].
ssdeep *ssdeep* hashes compared via ssdeep's custom edit-distance [Sec. 5.2].
ppdeep *ppdeep* [Sec. 5.3] hashes scored via standard Levenshtein [Sec. 6.2] and ssdeep's scoring logic.
ppdeep_mod *ppdeep* with modified scoring logic [Sec. 5.4].
jump *jumpHash* [Sec. 5.5] scored via Levenshtein similarity [Sec. 6.2].
bytebag Bags of byte-values [Sec. 5.6] scored via Jaccard [Sec. 6.1].
lzjd LZ dictionaries compared via Jaccard [Sec. 5.7].
bz *bzHash* [Sec. 5.8] with Levenshtein similarity [Sec. 6.2].
lev Levenshtein similarity [Sec. 6.2] on whole codes; only used with the small *proxies* codes [Sec. 7.3].
ncd Normalized Compression Distance applied to whole codes [Sec. 6.3].
size Codes compared via there length only [Sec. 6.4].

8.4 Datasets

Specification of the datasets used in the following sections.

solc-versions-testset See [Sec. 7.1].
wallets A subset of the wallets dataset [Sec. 7.2] including one code per unique skeleton.
proxies A subset of proxies dataset [Sec. 7.3] including one code per unique skeleton.
smallGroupsByAbi See [Sec. 7.4].

8.5 Performance Measures

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 with a size equaling the total number of same-group pairs.

qDist

As a second measure for the clustering-performance, we 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.
$\mathrm{Q}_n(x)$ ... Nth quartile of the series x
$\mathrm{Q}_2(x)$ ... Median of the series x.

$$qDist(x, y) = \frac{\mathrm{Q}_2(x) - \mathrm{Q}_2(y)}{\mathrm{Q}_2(x) - \mathrm{Q}_1(x) + \mathrm{Q}_3(y) - \mathrm{Q}_2(y)} \tag{7}$$

8.6 Plots

Violin Plots

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

- They show the similarity-scores of all possible code-pairs by one measure.
- 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.

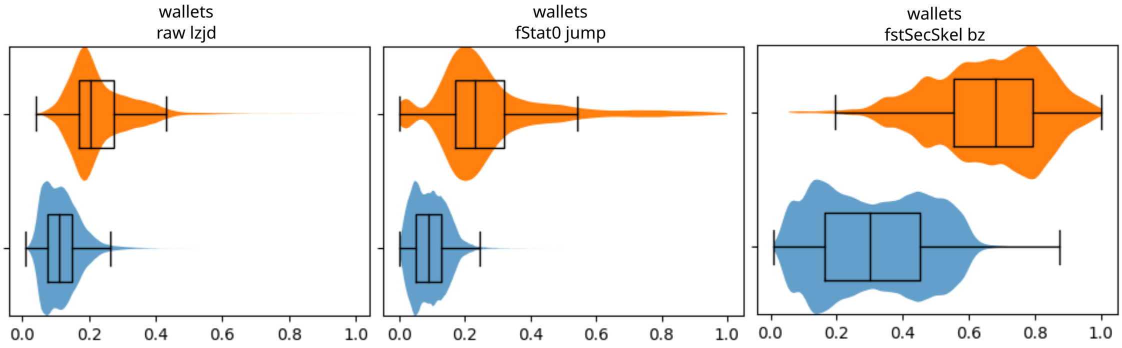


Fig. 1: violin plots

Histograms

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

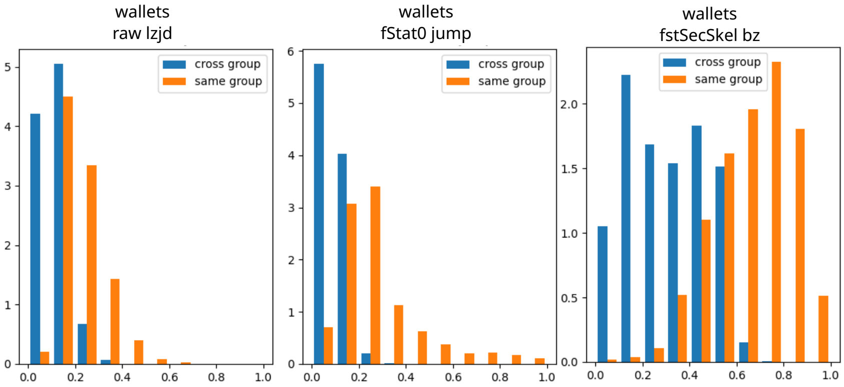


Fig. 2: histograms

- 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.

Statements

To signify the type of statement in the Results section, the following symbols are used:

$\mathcal{H}$ Hypothesis

$\mathcal{O}$ Observation

$\mathcal{R}$ Result

$\mathcal{C}$ Conclusion

9. Results

9.1 Testrun: Solc Versions Clustered with Bytebag

Data

An older version of the *solc-versions-testset*[7], 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. We defined the filter based on our intuition, 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)
```

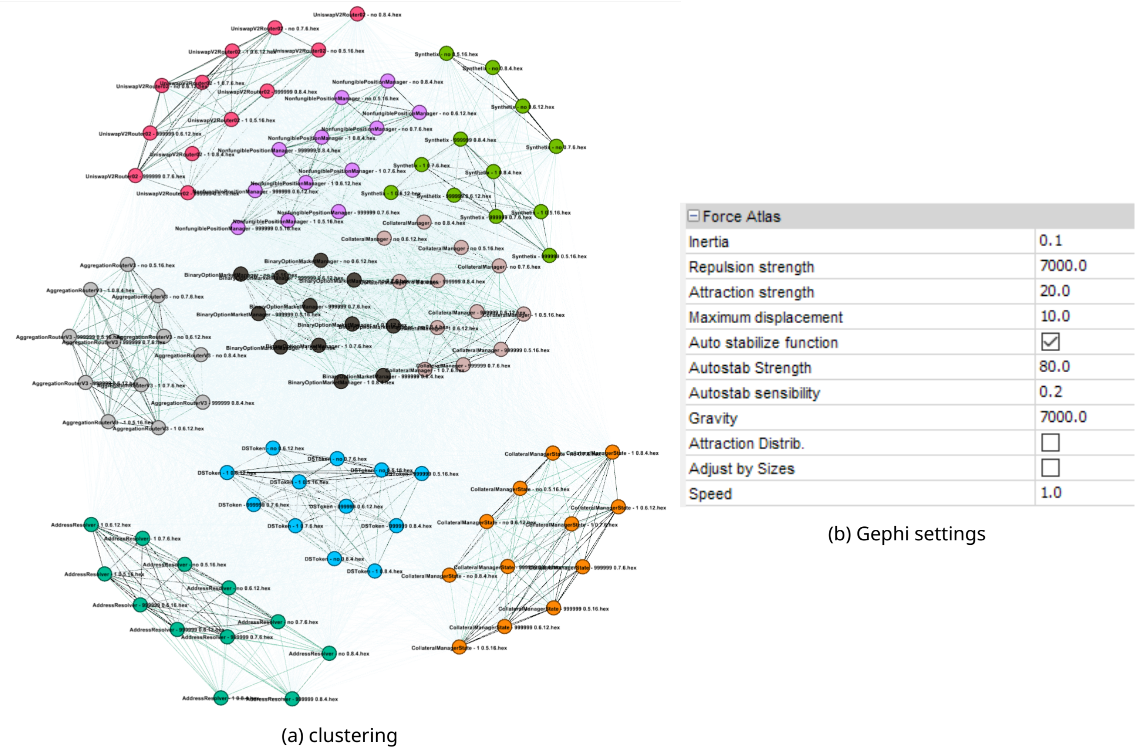
```
# GASPRICE CALLER SSTORE TIMESTAMP LOG3 ORIGIN INVALID LOG2 ADDRESS BLOCKHASH STATICCALL
    CALLDATALOAD CALL CALLDATASIZE RETURNDATASIZE EXTCODEHASH GAS LOG0 DIFFICULTY CODESIZE
    DELEGATECALL CALLVALUE RETURNDATACOPY RETURN NUMBER SELFDESTRUCT CALLCODE REVERT
    CALLDATACOPY COINBASE EXTCODESIZE 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 (*byteBag-significantOnly* column in the csv[1]).



(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

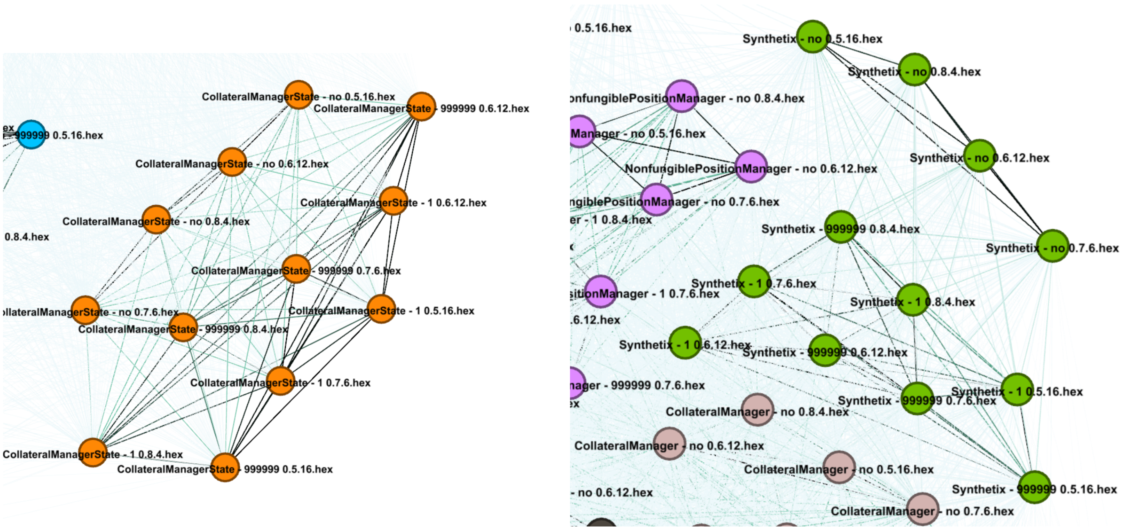
(b) Gephi settings

Fig. 3: Clustering of *solc-versions-testset* with *bytebag*

Observation

The clustering algorithm 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. 4b]. The no optimization variants in the CollateralManagerState cluster also group tightly [Fig. 4a]. UniswapV2Router02 changes significantly from solc 0.7.6 to 0.8.4, one reason ist the ABI default changed from v1 to v2, 0.8.4 also has a big table of PUSH and JUMP operations at the end not present in 0.7.6.

[1] `https://github.com/mrNuTz/ethereum-contract-similarity/blob/e18e6b01df1efbd3bc4969d9ad999ce1125f3649/runs/byteFiltersOnSolcVersions/out/comparisons_all.csv`


CollateralManagerState - no 0.5.16.hex
CollateralManagerState - 999999 0.6.12.hex
999999 0.5.16.hex
CollateralManagerState - no 0.6.12.hex
CollateralManagerState - 1 0.6.12.hex
CollateralManagerState - no 0.8.4.hex
0.8.4.hex
CollateralManagerState - 999999 0.7.6.hex
llateralManagerState - 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
0.5.16.hex
Synthetix - no 0.5.16.hex
Synthetix - no 0.8.4.hex
onfungiblePositionManager - no 0.8.4.hex
Manager - no 0.5.16.hex
NonfungiblePositionManager - no 0.6.12.hex
Synthetix - no 0.6.12.hex
ngiblePositionManager - no 0.7.6.hex
er - 1 0.8.4.hex
Synthetix - 999999 0.8.4.hex
Synthetix - no 0.7.6.hex
sitionManager - 1 0.7.6.hex
Synthetix - 1 0.7.6.hex
Synthetix - 1 0.8.4.hex
.6.12.hex
Synthetix - 1 0.6.12.hex
Synthetix - 999999 0.6.12.hex
nager - 999999 0.7.6.hex
Synthetix - 1 0.5.16.hex
Synthetix - 999999 0.7.6.hex
CollateralManager - no 0.8.4.hex
CollateralManager - no 0.6.12.hex
CollateralManager - no 0.5.16.hex
- no 0.6.12.hex
Synthetix - 999999 0.5.16.hex
CollateralManager - no 0.7.6.hex

(a) CollateralManagerState

(b) Synthetix

Fig. 4: no optimization

Analysis

To further investigate the Synthetix observation, we calculated the difference of the opcode frequencies between *Synthetix - no 0.8.4.hex* and *Synthetix - 999999 0.8.4.hex* [Tbl. 2][2]. The column *o0* lists the count of opcodes in *Synthetix - no 0.8.4.hex*, column *runs999999* the counts in *Synthetix - 999999 0.8.4.hex*, *diff* is the differences *runs999999* minus *o0* and *relDiff* is defined via (8).

$$relDiff\,(o0, runs999999) = \frac{runs999999 - o0}{\max\{o0, runs999999\}} \in [-1, 1] \tag{8}$$

Conclusion

Many of the opcodes, that we considered significant, show high 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.

It seams optimization options change the codes more than solc version changes.

9.2 Testrun: Determine Significant Opcodes

Because our 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. → ANOVA f-statistic

Data

We used the *solc-versions-testset*[7], because codes from the same group behave identically except for gas cost, runtime and storage of intermediate results.

Method

Calculate an f-statistic value for every byte value from 0 to 255, where codes compiled from the same source with different solc-versions and -options form the groups. See [3] for the source-code.

Results

See [Fig. 3] for the opcodes referenced in this work; for the full table see [4].

[2] https://github.com/mrNuTz/solc-version-testset/blob/2f431ed8b4bc3be498913bf224ac7991f0658b5d/runs/syntheticOptimization/index.mjs

[3] https://github.com/mrNuTz/ethereum-contract-similarity/blob/2633cfca61446fe5344a8ee57a2f7d92f18843e3/runs/byteDistribution/run.py

[4] https://github.com/mrNuTz/ethereum-contract-similarity/blob/2633cfca61446fe5344a8ee57a2f7d92f18843e3/runs/byteDistribution/out/solcOptions%20f-stat-by-byte.csv

dec	hex	opcode	o0	runs999999	diff	relDiff
8	0x08	ADDMOD	1	0	-1	-100 %
60	0x3C	EXTCODECOPY	1	0	-1	-100 %
83	0x53	MSTORE8	0	1	1	100 %
105	0x69	PUSH10	0	1	1	100 %
107	0x6B	PUSH12	0	1	1	100 %
117	0x75	PUSH22	0	1	1	100 %
118	0x76	PUSH23	1	0	-1	-100 %
140	0x8C	DUP13	0	2	2	100 %
153	0x99	SWAP10	0	1	1	100 %
163	0xA3	LOG3	0	1	1	100 %
10	0x0A	EXP	88	1	-87	-99 %
243	0xF3	RETURN	45	1	-44	-98 %
0	0x00	STOP	25	1	-24	-96 %
4	0x04	DIV	85	5	-80	-94 %
53	0x35	CALLDATALOAD	6	41	35	85 %
138	0x8A	DUP11	23	6	-17	-74 %
137	0x89	DUP10	31	10	-21	-68 %
103	0x67	PUSH8	6	2	-4	-67 %
136	0x88	DUP9	37	13	-24	-65 %
27	0x1B	SHL	67	24	-43	-64 %
3	0x03	SUB	240	92	-148	-62 %
80	0x50	POP	1366	527	-839	-61 %
2	0x02	MUL	46	18	-28	-61 %
86	0x56	JUMP	1041	410	-631	-61 %
145	0x91	SWAP2	498	205	-293	-59 %
91	0x5B	JUMPDEST	1347	600	-747	-55 %
97	0x61	PUSH2	1876	914	-962	-51 %
28	0x1C	SHR	1	2	1	50 %
100	0x64	PUSH5	2	1	-1	-50 %
139	0x8B	DUP12	10	5	-5	-50 %
141	0x8D	DUP14	1	2	1	50 %
162	0xA2	LOG2	2	1	-1	-50 %
144	0x90	SWAP1	970	490	-480	-49 %
150	0x96	SWAP7	21	11	-10	-48 %
131	0x83	DUP4	302	161	-141	-47 %
115	0x73	PUSH20	202	117	-85	-42 %
25	0x19	NOT	51	30	-21	-41 %
130	0x82	DUP3	427	252	-175	-41 %
127	0x7F	PUSH32	85	142	57	40 %
133	0x85	DUP6	81	51	-30	-37 %
81	0x51	MLOAD	362	239	-123	-34 %
146	0x92	SWAP3	193	128	-65	-34 %
151	0x97	SWAP8	9	6	-3	-33 %
152	0x98	SWAP9	2	3	1	33 %
96	0x60	PUSH1	1972	1319	-653	-33 %
135	0x87	DUP8	79	53	-26	-33 %
134	0x86	DUP7	99	67	-32	-32 %
241	0xF1	CALL	31	21	-10	-32 %
90	0x5A	GAS	65	45	-20	-31 %
62	0x3E	RETURNDATACOPY	66	46	-20	-30 %
250	0xFA	STATICCALL	34	24	-10	-29 %
61	0x3D	RETURNDATASIZE	171	121	-50	-29 %
59	0x3B	EXTCODESIZE	65	46	-19	-29 %
132	0x84	DUP5	110	78	-32	-29 %
82	0x52	MSTORE	282	397	115	29 %
128	0x80	DUP1	778	562	-216	-28 %
149	0x95	SWAP6	33	24	-9	-27 %
21	0x15	ISZERO	340	252	-88	-26 %
161	0xA1	LOG1	4	3	-1	-25 %
18	0x12	SLT	29	22	-7	-24 %
22	0x16	AND	317	241	-76	-24 %
253	0xFD	REVERT	204	156	-48	-24 %
99	0x63	PUSH4	216	167	-49	-23 %
1	0x01	ADD	602	480	-122	-20 %
87	0x57	JUMPI	319	264	-55	-17 %
148	0x94	SWAP5	30	25	-5	-17 %
147	0x93	SWAP4	64	54	-10	-16 %
129	0x81	DUP2	541	457	-84	-16 %
32	0x20	KECCAK256	7	6	-1	-14 %
84	0x54	SLOAD	98	86	-12	-12 %
51	0x33	CALLER	17	15	-2	-12 %
20	0x14	EQ	97	88	-9	-9 %
23	0x17	OR	11	10	-1	-9 %
17	0x11	GT	31	30	-1	-3 %
54	0x36	CALLDATASIZE	46	45	-1	-2 %
16	0x10	LT	28	28	0	0 %
48	0x30	ADDRESS	3	3	0	0 %
52	0x34	CALLVALUE	1	1	0	0 %
55	0x37	CALLDATACOPY	3	3	0	0 %
85	0x55	SSTORE	11	11	0	0 %
254	0xFE	INVALID	1	1	0	0 %

Tbl. 2: Synthetix optimization differences

Observations

`RETURN` has the lowest score of 2.5 confirming the interpretation of the clustering test-run.

op	dec	hex	min	max	mean	sd	f-stat
MOD	6	0x06	0	1	0.1	0.2	inf
MULMOD	9	0x09	0	2	0.1	0.5	inf
XOR	24	0x18	0	1	0.1	0.3	inf
BYTE	26	0x1a	0	6	0.4	1.4	inf
CALLVALUE	52	0x34	1	36	10.3	11.6	inf
MISSING	71	0x47	0	4	0.3	1.0	inf
DELEGATECALL	244	0xf4	0	4	0.7	1.3	inf
SELFDESTRUCT	255	0xff	0	1	0.1	0.2	inf
ADDRESS	48	0x30	0	18	3.1	4.8	8559.1
LOG3	163	0xa3	0	2	0.7	0.8	4129.7
TIMESTAMP	66	0x42	0	15	2.8	4.5	3774.3
ORIGIN	50	0x32	0	2	0.2	0.7	2680.2
LOG4	164	0xa4	0	4	0.5	1.1	2278.2
SHA3	32	0x20	0	68	19.6	17.6	2161.8
SWAP14	157	0x9d	0	15	1.0	3.3	2070.2
CALLDATASIZE	54	0x36	6	46	23.0	10.0	1945.2
CALLDATACOPY	55	0x37	0	74	5.0	10.0	1366.0
SIGNEXTEND	11	0x0b	0	45	2.4	9.4	1255.8
CALL	241	0xf1	0	31	8.7	9.1	1107.7
LOG2	162	0xa2	0	4	0.7	1.4	995.3
RETURNDATASIZE	61	0x3d	0	171	46.0	42.4	941.9
CALLER	51	0x33	2	24	12.1	6.0	882.9
EXTCODESIZE	59	0x3b	0	65	15.7	15.6	845.7
JUMPI	87	0x57	32	758	192.8	128.6	845.3
STATICCALL	250	0xfa	0	34	7.7	8.2	837.0
RETURNDATACOPY	62	0x3e	0	66	16.8	15.7	822.1
GAS	90	0x5a	0	65	17.1	15.6	804.4
DUP13	140	0x8c	0	89	6.2	12.8	682.1
DUP5	132	0x84	3	316	69.6	56.8	601.2
DUP8	135	0x87	0	104	24.6	27.4	600.9
GASPRICE	58	0x3a	0	3	0.1	0.5	530.5
SHR	28	0x1c	1	26	3.0	5.6	445.3
PUSH4	99	0x63	6	228	76.4	56.6	442.0
ISZERO	21	0x15	8	500	148.1	111.8	434.1
DUP7	134	0x86	0	130	38.4	33.2	366.3
ADD	1	0x01	17	1809	346.1	306.5	350.8
DUP9	136	0x88	0	74	15.9	19.7	345.0
MUL	2	0x02	0	419	37.1	51.5	331.2
DUP3	130	0x82	9	711	187.4	143.2	322.6
MLOAD	81	0x51	13	733	167.3	139.7	322.4
EQ	20	0x14	10	118	47.3	22.2	240.4
DUP1	128	0x80	42	1260	358.7	259.3	101.7
JUMPDEST	91	0x5b	61	1640	428.3	316.6	72.6
RETURN	243	0xf3	1	45	6.4	8.7	2.5
SDIV	5	0x05	0	0	0.0	0.0	nan
SAR	29	0x1d	0	0	0.0	0.0	nan
EXTCODECOPY	60	0x3c	0	0	0.0	0.0	nan
BLOCKHASH	64	0x40	0	0	0.0	0.0	nan
LOG0	160	0xa0	0	0	0.0	0.0	nan
CREATE	240	0xf0	0	0	0.0	0.0	nan

Tbl. 3: fStat values with *solc-versions-testset*

`JUMPI` has a high score of 845.3 despite its high prevalence of on average 192.8 per contract. It has the 16th highest f-statistic and ist the 11th most prevalent. It also has the 6th highest minimum of 32 occurrences in one contract. Opcodes scoring higher are much less frequent at a max mean of 46. In oder of f-statistic `ADD` is the next op-code with higher prevalence at rank 28 (f-stat 350.8, mean count 346.1).

`ISZERO` looks surprisingly significant at an f-statistic of 434.1 and a mean count of 148.1.

Interpretation

The *ABI* function signature jump tables at the beginning of every contract are identical within groups and they contain a high number of `JUMPI` ops, but they also contain an equal number of `PUSH4`, `DUP1` and `EQ` ops, which have lower f-stat values of 442.0, 101.7 and 240.4. The rest of the code does also contains a high number of `JUMPI` ops.

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

Remarks

We first ran this experiment with an older version of *solc-versions-testset*, in which the *ABI* encodings where not fixed programmatically. We noticed that the default encoding changed with solc version 0.8.0 from v1 to v2, resulting in considerable changes to the codes. Therefore we extended the dataset by the dimension ABI encoding, removed the `pragma` statements, for selecting the ABI version, from the source files and re-inject them during the generation of the codes.

Conclusion

A dataset, where the *ABI* tables are cut off, could be used, since interface-similarity can be better obtained via *fourbyte* [Sec. 5.1]. Based on the scores the *fStat* [Sec. 4.3.a] filter was defined.

9.3 Comparison Tables of all Measures

To get an overview, tables of all *qDists* and *Separations* are included (*wallets* [Tbl. 4], *solc-versions-testset* [Tbl. 5], *smallGroupsByAbi* [Tbl. 6], *proxies* [Tbl. 7]).

	Preprocess and Hash	qDist
1	raw fourbytes	400 %
2	raw ncd	208 %
3	fStat ncd	161 %
4	fStat jump	161 %
5	fstSecSkel ncd	158 %
6	skeletons ncd	154 %
7	fStat0 ncd	152 %
8	fStat0 jump	143 %
9	fstSecSkel jump	141 %
10	fStat0 bz	138 %
11	skeletons jump	137 %
12	fStat byteBag	137 %
13	fStat0 byteBag	137 %
14	fstSecSkel bz	136 %
15	fstSecSkel byteBag	129 %
16	fStat bz	127 %
17	raw lzjd	124 %
18	raw byteBag	120 %
19	skeletons byteBag	120 %
20	skeletons bz	117 %
21	raw bz	116 %
22	skeletons ppdeep_mod	106 %
23	fstSecSkel ppdeep_mod	105 %
24	fstSecSkel lzjd	103 %
25	skeletons lzjd	100 %
26	raw ppdeep_mod	87 %
27	fStat ppdeep_mod	87 %
28	fStat0 lzjd	76 %
29	fStat0 ppdeep_mod	49 %
30	raw jump	0 %
31	raw ssdeep	0 %
32	raw ppdeep	0 %
33	skeletons ssdeep	0 %
34	skeletons ppdeep	0 %
35	fstSecSkel ssdeep	0 %
36	fstSecSkel ppdeep	0 %
37	fStat ssdeep	0 %
38	fStat ppdeep	0 %
39	fStat0 ssdeep	0 %
40	fStat0 ppdeep	0 %
41	fStat lzjd	−1 %

	Preprocess and Hash	Separation
1	raw fourbytes	99 %
2	raw ncd	87 %
3	fStat jump	80 %
4	fStat0 jump	75 %
5	fStat0 bz	74 %
6	fstSecSkel bz	74 %
7	fStat ncd	74 %
8	fstSecSkel jump	74 %
9	fstSecSkel ncd	73 %
10	skeletons jump	72 %
11	skeletons ncd	72 %
12	fStat0 ncd	71 %
13	skeletons bz	69 %
14	fStat bz	68 %
15	fStat byteBag	68 %
16	fStat0 byteBag	68 %
17	raw bz	67 %
18	fstSecSkel byteBag	64 %
19	skeletons byteBag	62 %
20	raw lzjd	61 %
21	fstSecSkel ppdeep_mod	54 %
22	skeletons ppdeep_mod	54 %
23	fstSecSkel lzjd	53 %
24	skeletons lzjd	52 %
25	raw byteBag	51 %
26	fStat ppdeep_mod	44 %
27	raw ppdeep_mod	42 %
28	fStat0 lzjd	40 %
29	fStat ssdeep	36 %
30	fStat ppdeep	36 %
31	fStat0 ssdeep	34 %
32	fStat0 ppdeep	34 %
33	fstSecSkel ssdeep	33 %
34	fstSecSkel ppdeep	33 %
35	skeletons ssdeep	32 %
36	skeletons ppdeep	32 %
37	fStat0 ppdeep_mod	31 %
38	raw ssdeep	31 %
39	raw ppdeep	31 %
40	raw jump	25 %
41	fStat lzjd	13 %

Tbl. 4: *wallet* dataset *qDists* and *separations*

	Preprocess and Hash	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 %

	Preprocess and Hash	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 %

Tbl. 5: *solc-versions-testset* *qDists* and *separations*

	Preprocess and Hash	qDist
1	skeletons bz	220 %
2	fStat0 bz	216 %
3	fStat0V2 bz	215 %
4	raw bz	213 %
5	fstSecSkel bz	209 %
6	fStat byteBag	202 %
7	fStat0 byteBag	202 %
8	fStatV2 byteBag	201 %
9	fStat0V2 byteBag	201 %
10	raw ncd	194 %
11	fStat jump	181 %
12	fStatV2 jump	180 %
13	fStatV2 bz	174 %
14	fStat bz	173 %
15	fStat ncd	169 %
16	fStatV2 ncd	169 %
17	fstSecSkel byteBag	163 %
18	skeletons byteBag	163 %
19	fStat0V2 lzjd	159 %
20	fStat0 lzjd	158 %
21	raw lzjd	158 %
22	raw byteBag	151 %
23	skeletons ncd	150 %
24	fstSecSkel ncd	149 %
25	fStat lzjd	148 %
26	fStatV2 lzjd	148 %
27	fStat0 jump	142 %
28	fStat0V2 jump	142 %
29	fstSecSkel jump	140 %
30	skeletons jump	140 %
31	fStat0 ncd	138 %
32	fStat0V2 ncd	138 %
33	skeletons lzjd	134 %
34	fstSecSkel lzjd	134 %
35	fstSecSkel ppdeep_mod	126 %
36	skeletons ppdeep_mod	125 %
37	raw ppdeep_mod	117 %
38	fStat ppdeep_mod	114 %
39	fStatV2 ppdeep_mod	114 %
40	fStat0 ppdeep_mod	109 %
41	fStat0V2 ppdeep_mod	108 %
42	skeletons ssdeep	100 %
43	skeletons ppdeep	100 %
44	fstSecSkel ssdeep	100 %
45	fstSecSkel ppdeep	100 %
46	fStat ssdeep	100 %
47	fStat ppdeep	100 %
48	fStat0 ssdeep	100 %
49	fStat0 ppdeep	100 %
50	fStatV2 ssdeep	100 %
51	fStatV2 ppdeep	100 %
52	fStat0V2 ssdeep	100 %
53	fStat0V2 ppdeep	100 %
54	raw jump	91 %
55	raw ssdeep	0 %
56	raw ppdeep	0 %

	Preprocess and Hash	Separation
1	fStat ncd	77 %
2	fStatV2 ncd	77 %
3	skeletons bz	76 %
4	fStat jump	75 %
5	fStatV2 jump	75 %
6	raw ncd	74 %
7	fstSecSkel bz	74 %
8	raw bz	74 %
9	fStat0 bz	74 %
10	fStat0V2 bz	74 %
11	fStat byteBag	73 %
12	fStat0 byteBag	73 %
13	fStatV2 byteBag	73 %
14	fStat0V2 byteBag	73 %
15	skeletons ncd	71 %
16	fstSecSkel ncd	71 %
17	fStat0 ncd	71 %
18	fStat0V2 ncd	71 %
19	fStat0 jump	70 %
20	fStat0V2 jump	70 %
21	fstSecSkel jump	70 %
22	skeletons jump	70 %
23	fStat bz	69 %
24	fStatV2 bz	69 %
25	fStat ppdeep_mod	68 %
26	skeletons byteBag	68 %
27	fstSecSkel byteBag	68 %
28	skeletons lzjd	68 %
29	fstSecSkel lzjd	68 %
30	fStatV2 ppdeep_mod	68 %
31	raw byteBag	66 %
32	fStat ssdeep	66 %
33	fStat ppdeep	66 %
34	fStatV2 ssdeep	66 %
35	fStatV2 ppdeep	66 %
36	raw lzjd	66 %
37	fStat0V2 lzjd	65 %
38	fStat0 lzjd	65 %
39	raw jump	64 %
40	fStat0 ppdeep_mod	61 %
41	fStat0V2 ppdeep_mod	61 %
42	fStat0V2 ppdeep	60 %
43	fStat0 ppdeep	60 %
44	fstSecSkel ppdeep_mod	60 %
45	fstSecSkel ssdeep	60 %
46	fStat0V2 ssdeep	59 %
47	skeletons ppdeep_mod	59 %
48	fStat0 ssdeep	59 %
49	skeletons ssdeep	59 %
50	fstSecSkel ppdeep	59 %
51	skeletons ppdeep	58 %
52	fStat lzjd	57 %
53	fStatV2 lzjd	56 %
54	raw ppdeep_mod	52 %
55	raw ssdeep	28 %
56	raw ppdeep	28 %

Tbl. 6: *smallGroupsByAbi* dataset *qDists* and *separations*

9.4 Chunk Splitting

$\mathcal{H}$ The universal chunk splitting of *ssdeep* can be improved with domain knowledge about *EVM* runtime codes.

$\mathcal{R}$ Splitting by `JUMPI` achieves better results than the context trigger used by *ssdeep* [Fig. 5] [Fig. 6].

$\mathcal{R}$ `JUMPI` has a higher f-statistic value than `JUMP`, `RETURN` and `JUMPDEST` [Tbl. 3], because optimization removes the latter opcodes [Tbl. 2]. The chunk sizes obtained with `JUMPI` strike a good balance.

$\mathcal{C}$ Therefore `JUMPI` is a better splitter than patterns containing `JUMP`, `RETURN` or `JUMPDEST`.

9.5 Pre-Processing

$\mathcal{R}$ Chunk-hashes perform better on the skeleton of the first code section than on the whole code [Fig. 7].

$\mathcal{R}$ Filtering opcodes with *fStat* improves the results for *jump* and *bytebag* [Fig. 8].

	Preprocess and Hash	qDist
1	fStat byteBag	97 %
2	fStat0 byteBag	97 %
3	fStat lev	89 %
4	fStat0 ncd	85 %
5	skeletons ncd	78 %
6	fstSecSkel ncd	76 %
7	raw byteBag	75 %
8	raw ncd	73 %
9	raw lev	68 %
10	skeletons byteBag	68 %
11	fstSecSkel byteBag	68 %
12	raw lzjd	66 %
13	fStat ncd	58 %
14	skeletons lev	57 %
15	fStat0 lev	53 %
16	raw fourbyte	50 %
17	fstSecSkel lzjd	48 %
18	raw ppdeep_mod	45 %
19	fstSecSkel lev	44 %
20	skeletons lzjd	44 %
21	fStat0 lzjd	40 %
22	fStat lzjd	40 %
23	fstSecSkel bz	31 %
24	fStat0 bz	31 %
25	raw bz	25 %
26	skeletons bz	25 %
27	fStat jump	22 %
28	fStat bz	15 %
29	fstSecSkel ppdeep_mod	14 %
30	fStat0 ppdeep_mod	14 %
31	fStat ppdeep_mod	13 %
32	skeletons ppdeep_mod	8 %
33	raw ssdeep	0 %
34	raw ppdeep	0 %
35	raw jump	0 %
36	skeletons ssdeep	0 %
37	skeletons ppdeep	0 %
38	skeletons jump	0 %
39	fstSecSkel ssdeep	0 %
40	fstSecSkel ppdeep	0 %
41	fstSecSkel jump	0 %
42	fStat ssdeep	0 %
43	fStat ppdeep	0 %
44	fStat0 ssdeep	0 %
45	fStat0 ppdeep	0 %
46	fStat0 jump	0 %

	Preprocess and Hash	Separation
1	raw ncd	53 %
2	skeletons ncd	52 %
3	fStat lev	52 %
4	raw lev	50 %
5	fstSecSkel ncd	50 %
6	fStat byteBag	48 %
7	fStat0 byteBag	48 %
8	fStat0 lev	48 %
9	skeletons byteBag	47 %
10	fstSecSkel byteBag	47 %
11	fstSecSkel lev	45 %
12	skeletons lev	44 %
13	fStat jump	42 %
14	fStat ncd	42 %
15	fStat0 ncd	42 %
16	raw byteBag	39 %
17	raw lzjd	39 %
18	skeletons bz	39 %
19	raw ppdeep_mod	36 %
20	skeletons lzjd	36 %
21	skeletons ssdeep	34 %
22	skeletons ppdeep	34 %
23	fstSecSkel ssdeep	34 %
24	fstSecSkel ppdeep	34 %
25	fstSecSkel bz	34 %
26	fStat0 bz	33 %
27	raw bz	31 %
28	fstSecSkel lzjd	31 %
29	raw ssdeep	30 %
30	raw ppdeep	30 %
31	skeletons ppdeep_mod	30 %
32	fstSecSkel ppdeep_mod	30 %
33	fStat bz	30 %
34	fStat0 lzjd	30 %
35	fstSecSkel jump	27 %
36	fStat0 jump	27 %
37	fStat0 ssdeep	25 %
38	fStat0 ppdeep	25 %
39	skeletons jump	23 %
40	fStat lzjd	20 %
41	fStat ppdeep_mod	19 %
42	fStat0 ppdeep_mod	19 %
43	fStat ppdeep	13 %
44	raw fourbyte	11 %
45	fStat ssdeep	11 %
46	raw jump	8 %

Tbl. 7: *proxies* dataset *qDists* and *separations*

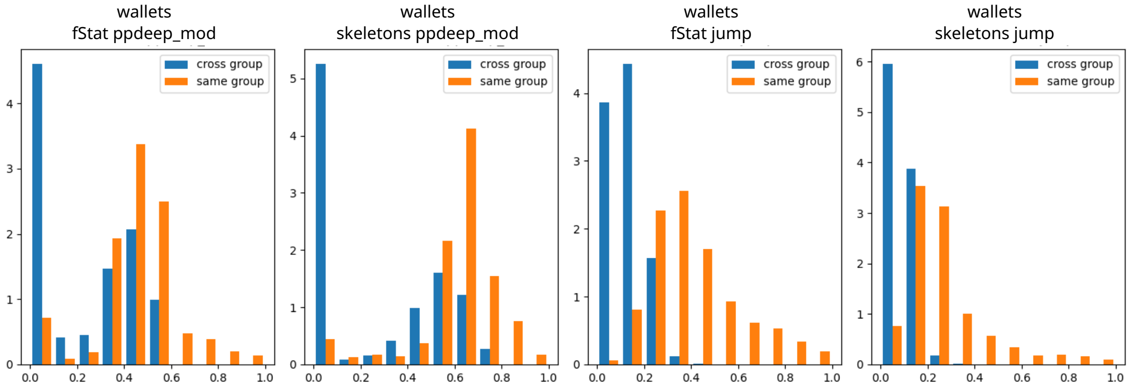


Fig. 5: *ppdeep_mod* vs. *jump* on *wallets*

9.6 ’K’ Sequences

𝒪 *ssdeep* and variants generate long sequences of ’K’ chunk-hashes when hashing code skeletons. E.g. the contract *ACATokenSale* at *0xb68a47aee7b6b79d203125eece1eef8ff19cbf3e* results in the hash *192:fKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKS:fO9SSNFtZT*.

ℛ *ppdeep_mod* without sequence striping performs better than *ssdeep* and *ppdeep* [Fig. 9].

ℛ *Levenshtein* works better than *Jaccard* on *ssdeep* hashes, because *Jaccard* reduces the ’K’ sequences to one entry.

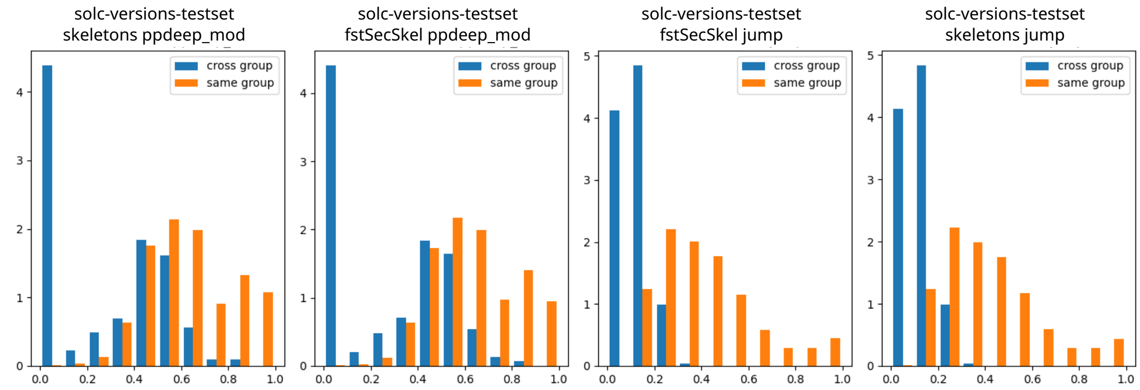


Fig. 6: *ppdeep_mod* vs. *jump* on *solc-versions-testset*

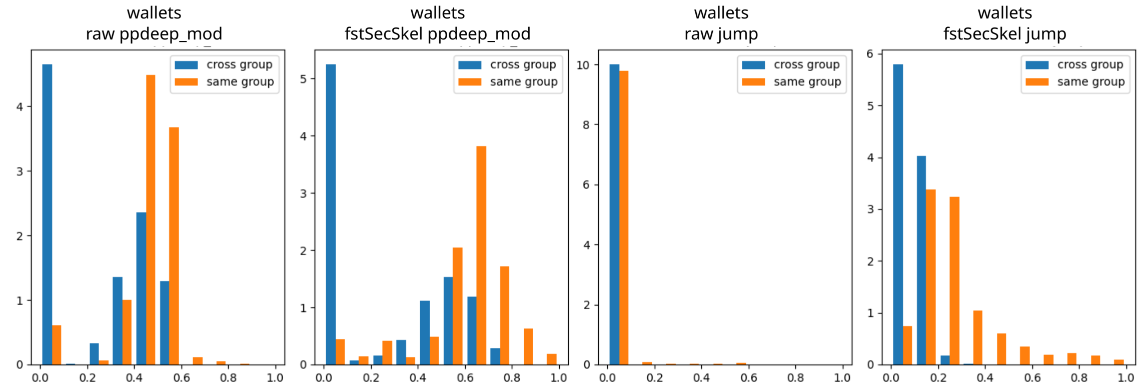


Fig. 7: *fstSecSkel* vs. *raw* on *wallets*

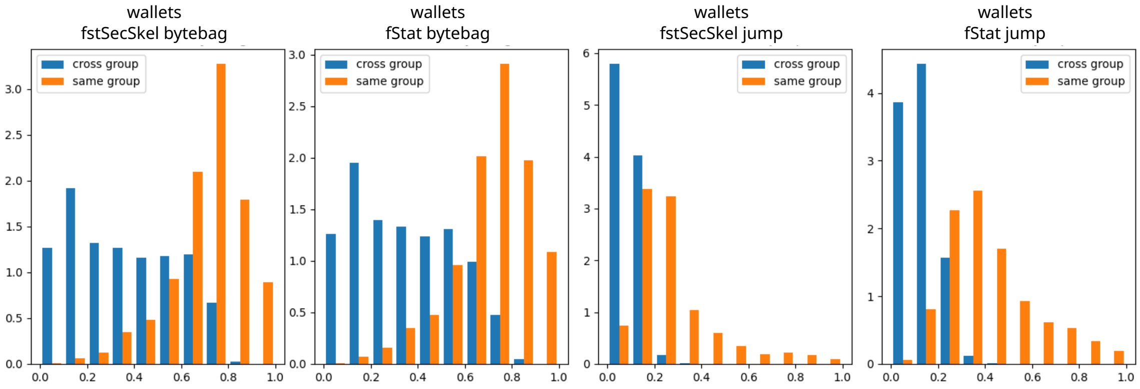


Fig. 8: *bytebag* and *jump* with *fstSecSkel* vs. *fStat* on *wallets*

9.7 NCD

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

$\mathcal{R}$ *ncd* comparison takes a long time, 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.

9.8 Proxies

$\mathcal{R}$ On the short *proxy* codes, the chunk-hashes (*jump*, *ssdeep*, *ppdeep*, *bz*) performance poorly; the binary similarity measures (*bytebag*, *lev*, *ncd*, *lzjd*) show better results.

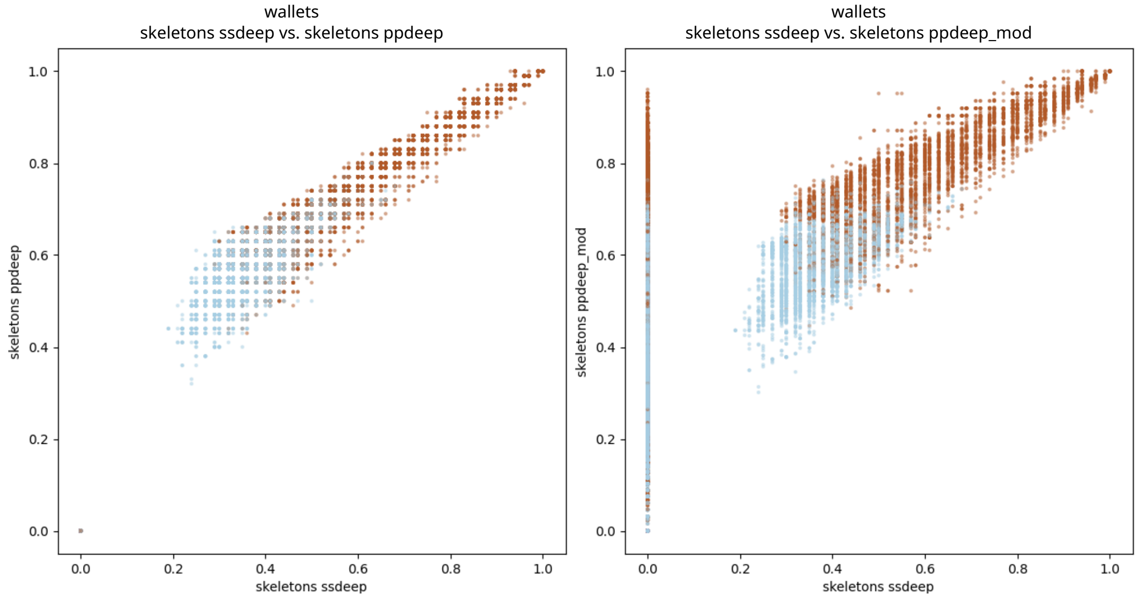


Fig. 9: Scatter plot of ppdeep and ssdeep similarity scores for all wallet code pairs

9.9 Runtime

O When comparing all codes with each other, the runtime of the hashing becomes insignificant, as the comparison of the hashes generally takes much longer, even though individually comparisons are faster than hash calculations.

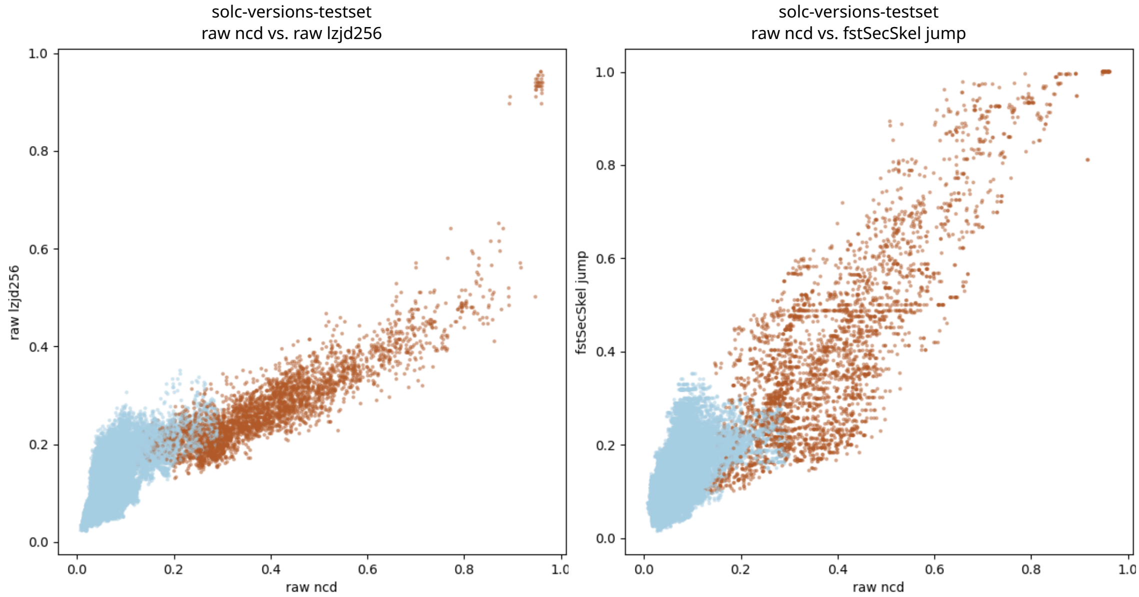


Fig. 10: Scatter plots of *raw ncd* similarity scores for all *solc-versions-testset* code pairs vs. *raw lzjd* (left) and *fstSecSkel jump* (right).

9.10 solc

O Fourbyte signatures in the jump table at the beginning of all codes are sorted and the functions jumped to are sorted as well.

O Optimization removes `JUMPDEST` opcodes.

O Enabling optimization when compiling with *solc* causes bigger changes to the codes than switching *solc* version.

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

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

ℛ *Levenshtein* works better than *Jaccard* with chunk-hashes, because *solc* orders functions by signature in the runtime code.

9.11 jump

ℛ Considering its simplicity *jump* performs surprisingly well, partially due to the fact that the number of `JUMPI` opcodes has a high *fStat* value.

ℛ *jump* correlates strongly with *ncd*, which seams to be more robust to optimization changes, but comparisons take 30 times longer and *jump* separates groups more sharply.

𝒪 The nativ Levenshtein implementation, used for comparison of *jump* and *bz* digests, is the fastest out of all hash similarities used in this work. Only Jaccard applied to the much shorter *fourbyte* signature sets is faster.

9.12 lzjd

ℛ The similarity numbers for 'same' and 'cross' pairs look close [Fig. 11][Fig. 12] but *raw lzjd* scores high in the *separation* and *qDist* comparison [Tbl. 4][Tbl. 5].

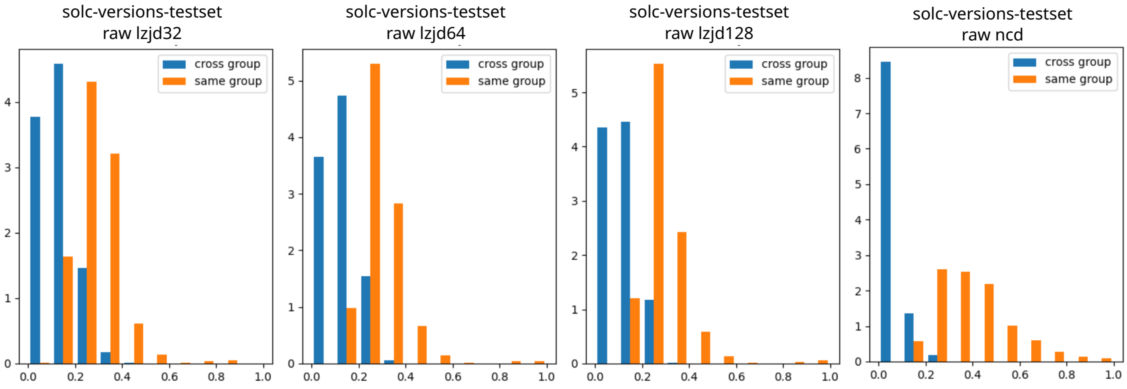


Fig. 11: *lzjd[32,64,128]* on all *raw solc-versions-testset* codes

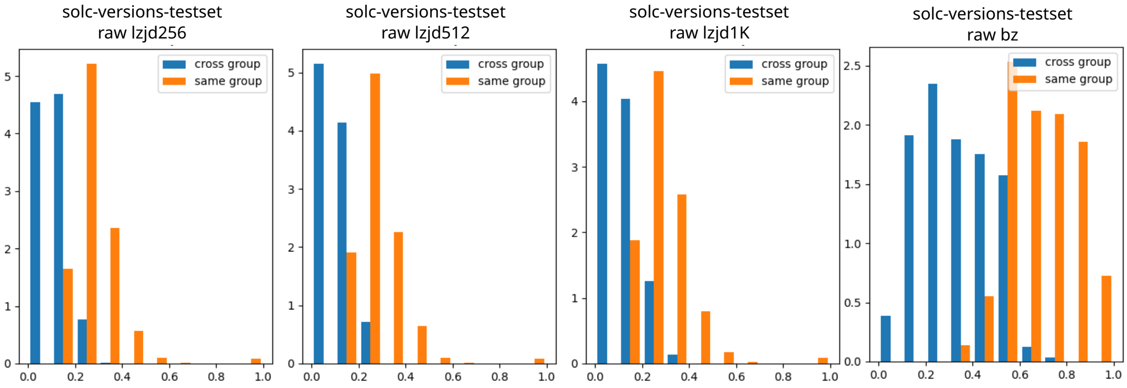


Fig. 12: *lzjd[256,512,1K]* on all *raw solc-versions-testset* codes

ℛ *lzjd* works better with *raw* codes and 256 is a good default *hash_size* setting [Tbl. 8][Tbl. 9][Fig. 13].

9.13 bytebag

ℛ *Bytebag* is simple and fast, jet it seems to be a useful way to determine similarity.

	Preprocess and Hash	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 %

	Preprocess and Hash	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 %

Tbl. 8: *lzjd qDists* and *separations* on all codes from *solc-versions-testset*

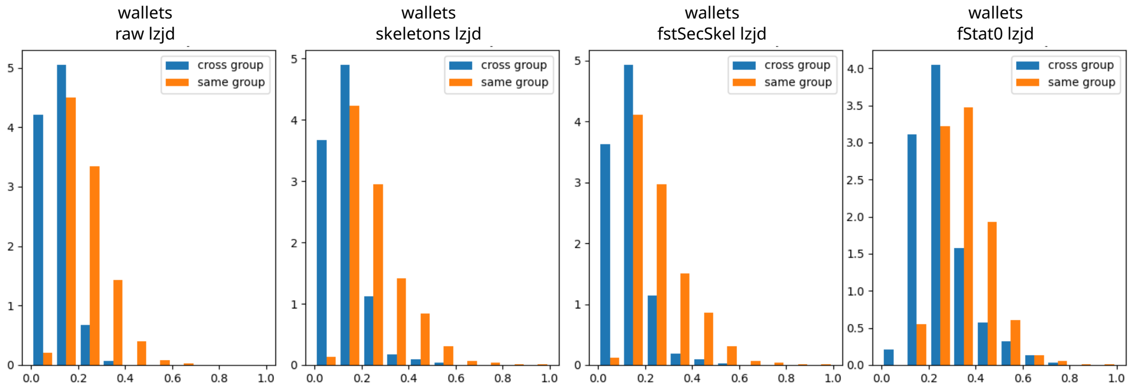


Fig. 13: *lzjd[256]* on *wallets* dataset with different pre-processing methods

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

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

Tbl. 9: *lzjd qDists* and *separations* on all *wallet* codes

10. Remarks

Evaluation Framework

Quickly re-runnable scripts with few dependencies keep things consistent, but splitting data generation and plot rendering would have allowed faster exploration of the data.

Not using any command line arguments and committing out files to the git repository turned out to be a good decision in our opinion, because this way input and output can easily be associated and referenced later.

11. Conclusion

Key Takeaways

Contracts with similar functionality mostly have similar interfaces—usually many of the functions have the same name and arguments, hence *fourbyte* should be tried first, also considering that it has the quickest runtime. If the results are not satisfying the other, more complicated, methods can be applied.

As hypothesized, comparing skeletons yields more meaningful results than comparing the whole code.

Bytecode is different from text, images or audio, because a complete reordering can result in the same execution and many different opcode sequences can push the same bytes onto the stack; but because blockchain storage costs money, short sequences are preferred and similar functions are often ordered in the same way, due to *solc* ordering by signature.

The *solc-versions-testset* has shown that changed compile options can result in widespread changes throughout the runtime code, especially no optimization vs. optimization with high runs settings and *ABI v1* vs. *v2*.

`JUMPI` is a good splitter, which hash or similarity measure is applied to the chunks has less of an impact, provided that the chunks are hashed with a fuzzy method or pre-filtered.

NCD is flexible and robust, but by far the slowest approach tested in this work.

Next Steps

Taking a deeper look into the existing results, e.g. by studying code pairs where different methods yield disagreeing scores and clustering codes known to be similar to see if further subdivision is meaningful.

Combining this work's methods, to obtain combinational similarity scores, starting with fast methods like *fourbyte* and *bytebag*.

Extending *solc-versions-testset* by renaming functions, this results in changed *fourbyte* signatures, in turn leading to a reordering of the code, due to *solc* ordering functions by signature.

Running a few fast clusterings, investigate the results and use the learnings in a big clustering run used to analyze blockchain activity, this could potentially be sped up by storing pre-calculated hashes in a database.

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] Antti Haapala. *The Levenshtein Python C extension module contains functions for fast computation of Levenshtein distance and string similarity*. 2014. URL: `https://github.com/ztane/python-Levenshtein` (visited on 04/11/2022).

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

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

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

[8] 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`.

[9] 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`.

[10] 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`.

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

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

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

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