# Fuzzy Hashing Ethereum Smart Contracts

Bachelor's Thesis in Software and Information Engineering

March 29, 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. [8]

**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. [10]

**ppdeep_mod** Modified version of ppdeep. [5]

**LZJD - Lempel-Ziv Jaccard Distance** Generates a compression dictionary. [7]

**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. $J(A, B) = |A \cap B|/|A \cup B| \in [0, 1]$

**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 [8].

`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 [8]. 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 wich opcode sequences change would most likely require a lot of manual work. To quickly and simply distinguish between opcode wich are likely to change and once that don't change I used the `solc-version-testset` [6] and reduced the codes to bytebags. Then I calculated an f-statistic [Figure 1 on Page 4] for each opcode, using common source-code as grouping criterion and the count of the opcode as value.

$$F = \frac{\text{between group variability}}{\text{within group variability}}$$

Figure 1: One-way ANOVA F-test statistic

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,
)
```

### 5.4 Tests

I tested with the following pre-processing settings.

**raw** Tho whole code unprocessed.

**fstSec** The first code section.

**skel** The skeleton of the whole code.

**fstSecSkel** The skeleton of the fist code section.

**fStat** fstSecSkel filtered of 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.

## 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. 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.

The test-set and the code used to generate it can be found online[6].

#### 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 [Table 1 on Page 5] 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 % |

Table 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 only referenced. 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. For reuse it is available publicly [5].

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 calculated. 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.

## 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[9] wich 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 [10]).

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-version-testet [6] I implemented jumpHash [5].

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 `ord("°")=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

(Raff et al. 2018 [7]) introduces `LZJD` and compares it to `sdhash` and `ssdeep`.

LZ compression is done by generating a dictionary of frequent sequences, this dictionary is the result of this hashing method.

## 9.7 bzHash

bzHash is based on peHash[11] 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 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.
6. Discretize to positive integer values in [0,3].
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

def bzHash(bs:bytes, chunkRes=4) -> str:
  jumpi = b'\x57'
  res = [bzCompRatio(chunk) for chunk in bs.split(jumpi)]
  res = sdFromMean(res)
  return ''.join(chr(0xb0 + 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 $y$.

$$NCD(x,y) = \frac{Z(xy) - \min\{Z(x), Z(y)\}}{\max\{Z(x), Z(y)\}} \quad (1)$$

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 Evaluation with wallets dataset

[Figure 2 on Page 9]

## 10.2 solc-version-testset

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

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.

TODO: how do I know this?

## 10.3 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, wich seams to be more robust to optimization changes, but comparisons take 30 times longer and jumpHash separates groups more sharply.

TODO: sharply? TODO: How do I know this?

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.

TODO: investigate clusters within groups

## 10.4 Bytebag

Combined with skeletonization and opcode filtering Bytebag performs better than ssdeep in some scenarios.

## 10.5 LZJD parameters

[Table 2 on Page 10]

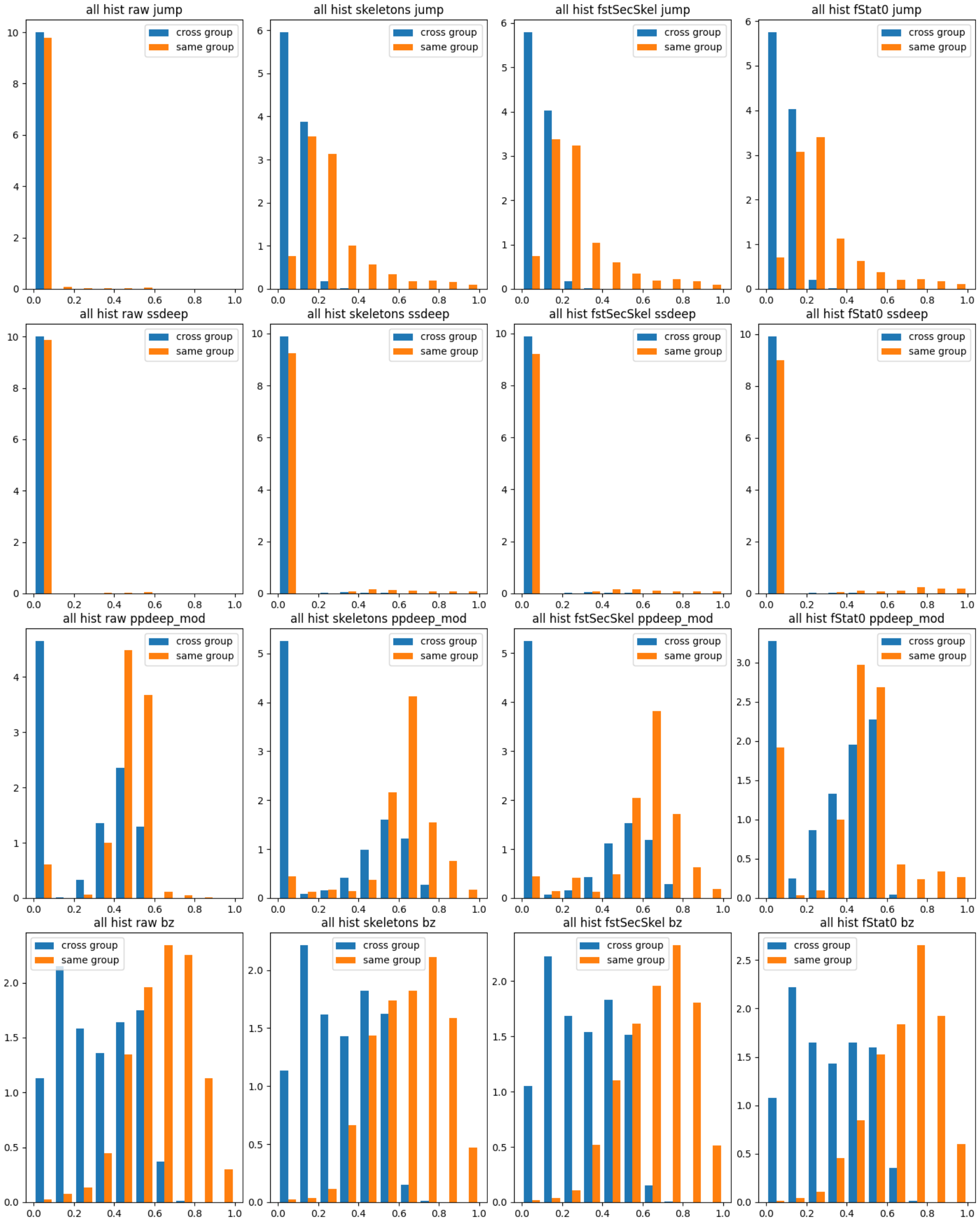

Figure 2: wallets

## 10.6 ssdeep variants

## 10.7 F-Stat filter

## 10.8 Solc Versions clustered with byteBagJaccard

### 10.8.1 Data

Nine Solidity source files compiled with 4 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

|  | Measure | Separation |
|---|---|---|
| 1 | raw ncd | 0.8921856639247944 |
| 2 | skel ncd | 0.7532314923619271 |
| 3 | raw lzjd256 | 0.7223854289071681 |
| 4 | raw lzjd512 | 0.7159224441833137 |
| 5 | raw lzjd128 | 0.699177438307873 |
| 6 | raw lzjd64 | 0.6401292596944771 |
| 7 | skel lzjd32 | 0.5925381903642774 |
| 8 | skel lzjd256 | 0.5801997649823737 |
| 9 | raw lzjd32 | 0.5655111633372503 |
| 10 | skel lzjd64 | 0.5655111633372503 |
| 11 | skel lzjd512 | 0.559048178613396 |
| 12 | skel lzjd128 | 0.5581668625146886 |
| 13 | raw lzjd1K | 0.5455346650998825 |
| 14 | skel lzjd1K | 0.5 |

Table 2: separations

version without altering the function of the contracts.

The runtime codes where segmented and the first segments skeletonized.

### 10.8.2 Similarity

Similarity was calculated via byteBagJaccard after filtering the op-codes with an opcode filter.

Opcode filter

```
(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)
```

### 10.8.3 Clustering

Clustering was done with Gephi (0.9.2), see figure for settings.

### 10.8.4 Observation

The clustering algo first mixed a few similar contracts into one blob, with a little manual help it formed cohesive clusters [Figure 3 on Page 11].

Noticeable is that the Synthetix codes without optimization from all 4 solc versions form a cluster separate from the other Synthetix code.

### 10.8.5 Analysis

When comparing the frequency of the op-codes in ‘Synthetix - no 0.8.4.hex‘ and ‘Synthetix - 999999 0.8.4.hex‘ the following 7 op-codes show the highest change.

| 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 |

Table 3: optimization differences

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

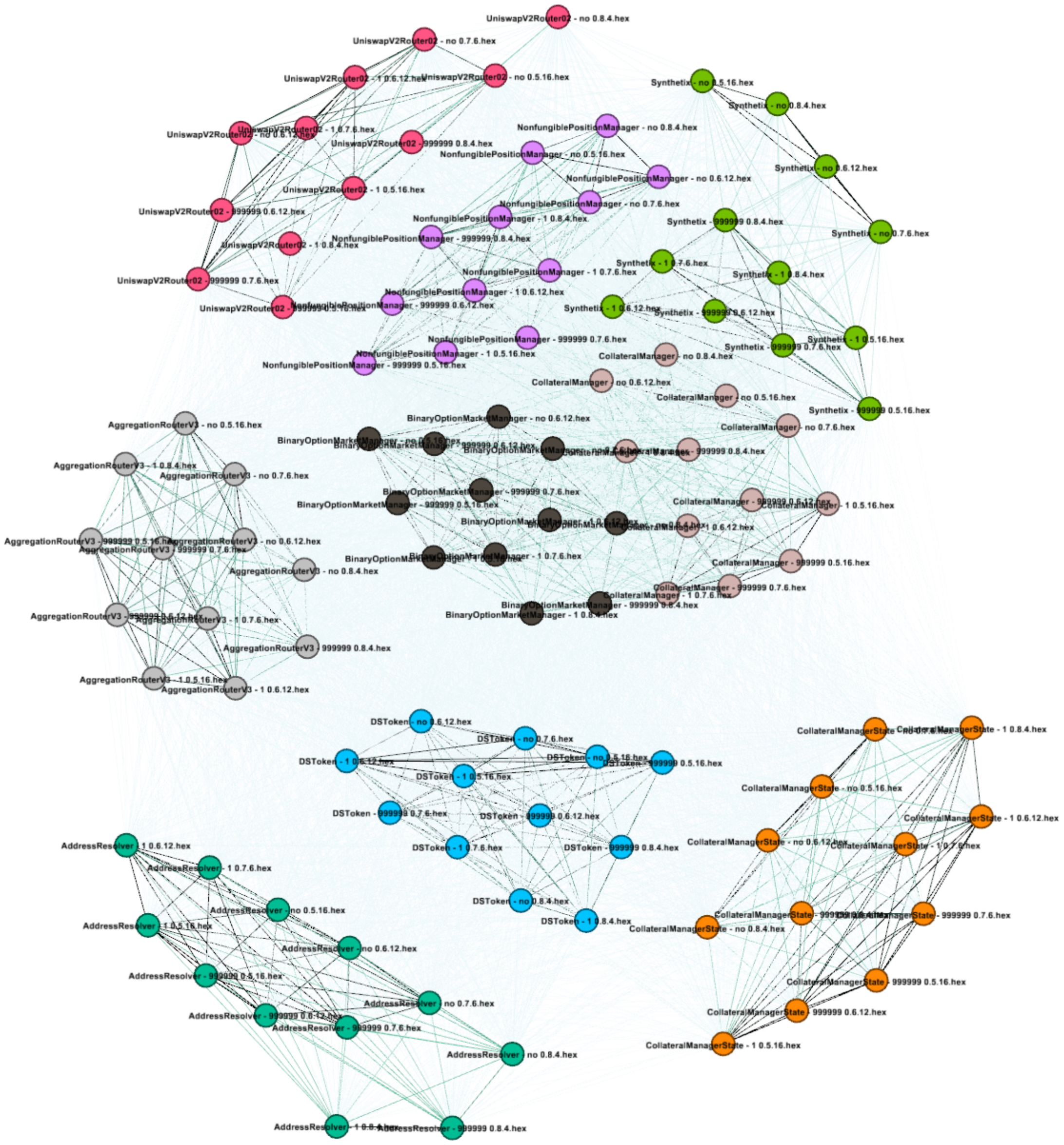
UniswapV2Router02 - no 0.8.4.hex
UniswapV2Router02 - no 0.7.6.hex
UniswapV2Router02 - 1 0.6.12.hex
UniswapV2Router02 - no 0.5.16.hex
UniswapV2Router02 - no 0.6.12.hex
UniswapV2Router02 - 1 0.7.6.hex
UniswapV2Router02 - 999999 0.8.4.hex
UniswapV2Router02 - 1 0.5.16.hex
UniswapV2Router02 - 999999 0.6.12.hex
UniswapV2Router02 - 1 0.8.4.hex
UniswapV2Router02 - 999999 0.7.6.hex
UniswapV2Router02 - 999999 0.5.16.hex
Synthetix - no 0.5.16.hex
Synthetix - no 0.8.4.hex
Synthetix - no 0.6.12.hex
Synthetix - no 0.7.6.hex
Synthetix - 999999 0.8.4.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
Synthetix - 999999 0.5.16.hex
NonfungiblePositionManager - no 0.8.4.hex
NonfungiblePositionManager - no 0.5.16.hex
NonfungiblePositionManager - no 0.6.12.hex
NonfungiblePositionManager - no 0.7.6.hex
NonfungiblePositionManager - 1 0.8.4.hex
NonfungiblePositionManager - 999999 0.8.4.hex
NonfungiblePositionManager - 1 0.7.6.hex
NonfungiblePositionManager - 1 0.6.12.hex
NonfungiblePositionManager - 999999 0.6.12.hex
NonfungiblePositionManager - 999999 0.7.6.hex
NonfungiblePositionManager - 1 0.5.16.hex
NonfungiblePositionManager - 999999 0.5.16.hex
CollateralManager - no 0.8.4.hex
CollateralManager - no 0.6.12.hex
CollateralManager - no 0.5.16.hex
CollateralManager - no 0.7.6.hex
CollateralManager - 999999 0.8.4.hex
CollateralManager - 1 0.5.16.hex
CollateralManager - 1 0.6.12.hex
CollateralManager - 999999 0.5.16.hex
CollateralManager - 999999 0.7.6.hex
CollateralManager - 1 0.7.6.hex
BinaryOptionMarketManager - no 0.6.12.hex
BinaryOptionMarketManager - 999999 0.7.6.hex
BinaryOptionMarketManager - 999999 0.5.16.hex
BinaryOptionMarketManager - 1 0.7.6.hex
BinaryOptionMarketManager - 999999 0.8.4.hex
BinaryOptionMarketManager - 1 0.8.4.hex
AggregationRouterV3 - no 0.5.16.hex
AggregationRouterV3 - 1 0.8.4.hex
AggregationRouterV3 - no 0.7.6.hex
AggregationRouterV3 - 999999 0.5.16.hex
AggregationRouterV3 - no 0.6.12.hex
AggregationRouterV3 - 999999 0.7.6.hex
AggregationRouterV3 - no 0.8.4.hex
AggregationRouterV3 - 1 0.7.6.hex
AggregationRouterV3 - 999999 0.8.4.hex
AggregationRouterV3 - 1 0.5.16.hex
AggregationRouterV3 - 1 0.6.12.hex
DSToken - no 0.6.12.hex
DSToken - no 0.7.6.hex
DSToken - no 0.5.16.hex
DSToken - 1 0.6.12.hex
DSToken - 999999 0.5.16.hex
DSToken - 1 0.5.16.hex
DSToken - 999999 0.7.6.hex
DSToken - 999999 0.6.12.hex
DSToken - 1 0.7.6.hex
DSToken - 999999 0.8.4.hex
DSToken - no 0.8.4.hex
DSToken - 1 0.8.4.hex
CollateralManagerState - 1 0.8.4.hex
CollateralManagerState - no 0.5.16.hex
CollateralManagerState - 1 0.6.12.hex
CollateralManagerState - no 0.6.12.hex
CollateralManagerState - 1 0.7.6.hex
CollateralManagerState - 999999 0.7.6.hex
CollateralManagerState - no 0.8.4.hex
CollateralManagerState - 999999 0.5.16.hex
CollateralManagerState - 999999 0.6.12.hex
CollateralManagerState - 1 0.5.16.hex
AddressResolver - 1 0.6.12.hex
AddressResolver - 1 0.7.6.hex
AddressResolver - no 0.5.16.hex
AddressResolver - 1 0.5.16.hex
AddressResolver - no 0.6.12.hex
AddressResolver - 999999 0.5.16.hex
AddressResolver - no 0.7.6.hex
AddressResolver - 999999 0.6.12.hex
AddressResolver - 999999 0.7.6.hex
AddressResolver - no 0.8.4.hex
AddressResolver - 1 0.8.4.hex
AddressResolver - 999999 0.8.4.hex

Figure 3: bytebag cluster with force-atlas

- ‘59 0x3B EXTCODESIZE‘ is also very consistent. [csv]
- ‘90 0x5A GAS‘ has a high absolute difference between the Synthetix contracts but it has higher differences across contracts.

### 10.8.6 Interpretation

The ‘RETURN‘ code should definitely be filtered out since optimization drastically reduces its prevalence.

Optimization options change the codes more than solc version changes.

## 11 Remarks

Section um alles zu notieren, was Ihnen an Absonderlichkeiten oder Schwierigkeiten untergekommen ist, inklusive Lessons learned (also was Sie beachten würden, wenn Sie nochmals beginnen würden); wenn es da viel zu berichten gibt, können Sie es natürlich weglassen. Der Sinn einer eigenen Section ist, dass Sie hier Ihre Eindrücke informell ohne tiefere Begründungen wiedergeben können, während die Aussagen in den anderen Abschnitten begründet sein sollten.

# 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 and Charles Nicholas. “Lempel-Ziv Jaccard Distance, an effective alternative to ssdeep and sdhash”. In: *Digital Investigation* 24 (2018), pp. 34–49.

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

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

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

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