Fuzzy Hashing Ethereum Smart Contracts

Bachelor's Thesis in Software and Information Engineering

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 date. 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. Transaction change the state.

Blockchain Complete record of all transaction.

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

Ethereum account An account has an address, a 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 Set of contract functions callable by other accounts.

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

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

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

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.

3 Introduction

`TODO: This is not an introduction`, it is related work / overview and needed explanations / definitions.

Skeletons are specific but not sensitive and interfaces are sensitive but not specific, other methods are needed to fill the gap.

Starting points where the studies (He et al. 2020 [3]) and (Norvill et al. 2017 [7]). They both use ssdeep[5] described in the paper (Kornblum 2006 [4]).

I'am evaluating ssdeep, possible modifications and other Fuzzy Hashing methods.

3.1 Fuzzy Hashing Categories according to (Naik et al. 2020 [6])

- Context-Triggered Piecewise Hashing (CTPH)
- Statistically Improbable Features (SIF)
- Block-Based Hashing (BBH)
- Block-Based Rebuilding (BBR)

I will be evaluation CTPH and BBH.

3.1.1 Binary Hashing (Information Theory based Methods)

- Normalized Compression Ratio (NCD) - Not really a hashing method, because it's applied to the whole files.
- Compression Ratio
- LZJD

I will also look at these methods.

3.2 Other Ways to Cluster Contracts

I'am not looking at these methods in my thesis.

Control Flow Graph Generate CFGs with radare2.

n-grams Method from natural language processing the looks at contiguous sequences of n items (letters, opcodes).

Neural Network Use call-graph date for learning of hash similarity.

4 Pre-Processing

4.1 Segmentation and Skeletonization [10]

4.1.1 Segment selection

Based on the segmentation there are various ways to apply the fuzzy hashing methods.

1. On the whole code as is.
2. On the whole code excluding meta-data.
   - Meta-data has no effect on execution and changes between compilations.
3. On all code-sections concatenated excluding data-sections especially 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.

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

4.1.2 Skeletonization

In all selection cases, there is the option, of ignoring the arguments to push-operations.

- 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.
- This alternative boils down to only including the opcodes of the push-instructions.
- Setting the push arguments to zero has the benefit of preserving the ability to disassemble the code.

4.2 Opcode filtering

The same externally observable behavior can be achieved via different opcode sequences.

Some opcodes are more like to change with solc-versions or compile-options than others.

Removing less significant opcodes before hashing should yield more meaningful similarity scores.

4.2.1 F-Statistic Filter

To determine significant opcodes I used the solc-version-testset[9].

1. Calculate the Bytebag of all codes.
2. Calculate an f-statistic for each opcode, using common source-code as grouping criterion and the count of the opcode as value.
3. Order Opcodes by f-statistic and select the top 30.

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

Figure 1: One-way ANOVA F-test statistic

5 Data sets

5.1 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[9].

5.1.1 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 4] 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

5.2 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.

5.3 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.

5.4 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.

6 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 [8].

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 the number of same-group pairs.

7 Hashing Methods

`TODO: reference to exact page number or figure`

7.1 Signature similarity

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

7.2 ssdeep - Context Triggered Piecewise Hashes (CTPH)

ssdeep is based on spamsum[12] wich was written for email spam detection.

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).

7.2.1 Modified ssdeep

To make modifications easy I used a version of ssdeep implemented in pure python (ppdeep [13]).

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.

7.3 JUMP-Hash

1. Split code by `JUMPI` into chunks.
2. Hash each chunk with `sha1`.
3. Map the first byte 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)
```

7.4 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))
```

7.5 LZJD - Lempel-Ziv Jaccard Distance

This is how a cite[11] looks like.

7.6 BZ-Hash

BZ-Hash is based on ImpHash **TODO: cite ImpHash** that calculates the compression ratio for equal sized chunks 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()
```

7.7 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.

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

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))
```

8 Results

8.1 Evaluation with wallets dataset

[Figure 2 on Page 8]

8.2 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?

8.3 JUMP-Hash

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 JUMP-Hash 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

8.4 Bytebag

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

8.5 LZJD parameters

[Table 2 on Page 9]

Figure 2: wallets

8.6 ssdeep variants

8.7 F-Stat filter

8.8 Solc Versions clustered with byteBagJaccard

8.8.1 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.

8.8.2 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)
```

8.8.3 Clustering

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

8.8.4 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 10].

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

8.8.5 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.

TODO: latex table

```
|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|
```

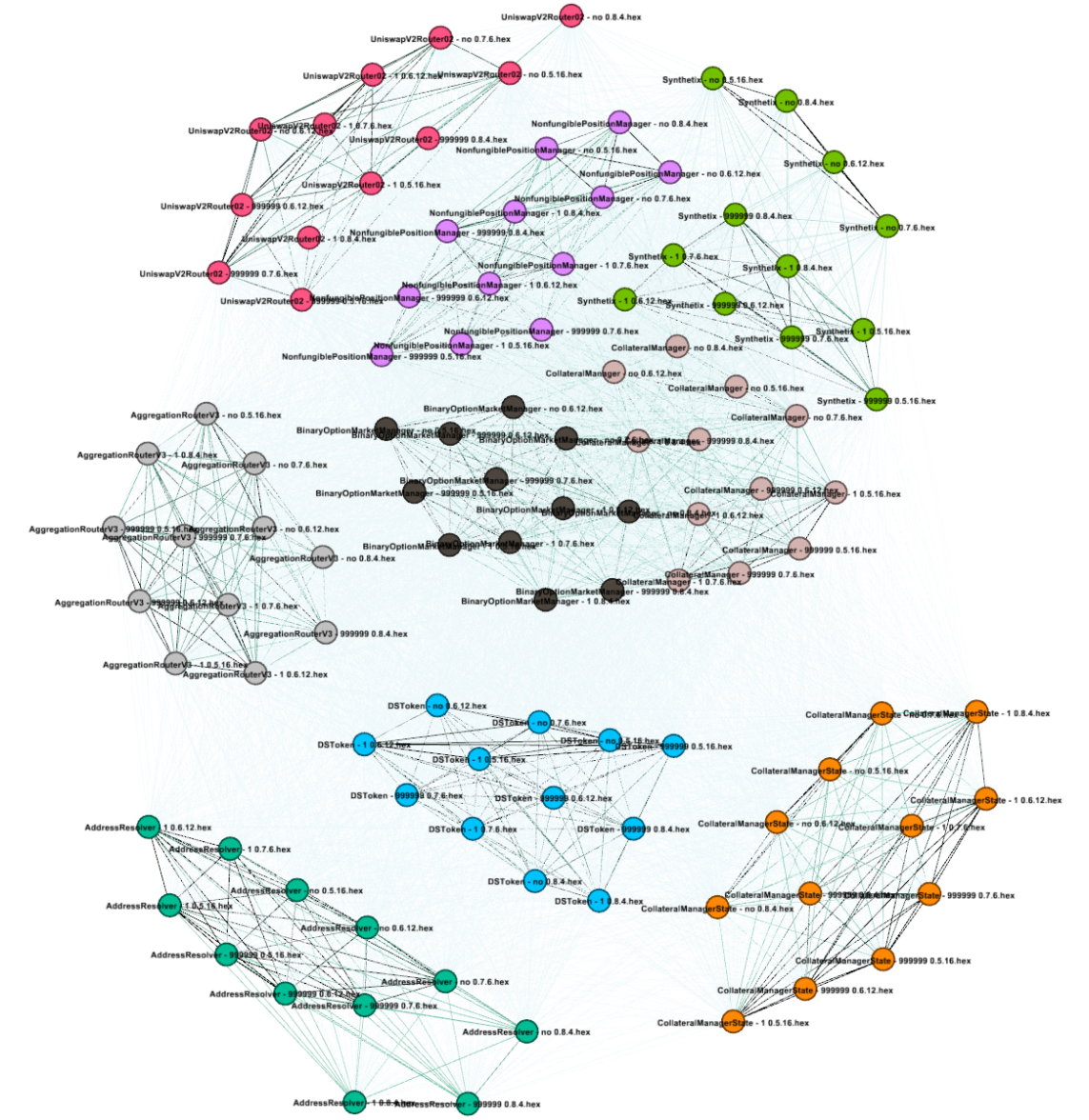

Figure 3: bytebag cluster with force-atlas

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

8.8.6 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.

9 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.

10 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] Ningyu He et al. “Characterizing code clones in the ethereum smart contract ecosystem”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2020, pp. 654–675.

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

[5] Jesse Kornblum. *ssdeep - Fuzzy hashing program*. 2017. URL: https://ssdeep-project.github.io/ssdeep/index.html (visited on 03/09/2022).

[6] Nitin Naik et al. “Embedding Fuzzy Rules with YARA Rules for Performance Optimisation of Malware Analysis”. In: *2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE. 2020, pp. 1–7.

[7] Robert Norvill et al. “Automated labeling of unknown contracts in ethereum”. In: *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE. 2017, pp. 1–6.

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

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

[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 H Sung et al. “Static analyzer of vicious executables (save)”. In: *20th Annual Computer Security Applications Conference*. IEEE. 2004, pp. 326–334.

[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).