Evaluation of Ethereum Smart Contract Similarity Measures

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

This thesis explores fuzzy hashing methods to compute similarities between Ethereum Smart Contracts. For this purpose a set of python utilities was implemented and an data-set for evaluation generated.

2 Goal

The goal was to compare different similarity measures and pre-processing steps and evaluate how they respond to changes in the codes. This should be accomplish with a set of reusable python utilities.

3 Introduction

Starting points where the studies (He et al. 2020 [2]) and (Norvill et al. 2017 [5]).

They both use ssdeep[4] described in the paper (Kornblum 2006 [3]).

4 Terms

Ethereum account
: An account has an eth balance and can execute transactions.

Smart Contract
: An ethereum account with associated runnable code and data stored on the Ethereum Blockchain. Can refer to just the runtime code or the source code or one Contract interface in the source code.

Runtime Code
: The runnable code stored on the blockchain as a string of opcode bytes. Used synonymously with code or bytecode.

EVM
: Ethereum Virtual Machine. Transaction base stack machine.

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

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

Contract interface

Set of contract functions callable by other accounts.

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.

5 Data sets

5.1 solc-versions-testset [7]

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.

The following solc versions where used:

1. `0.5.16`
2. `0.6.12`
3. `0.7.6`
4. `0.8.4`

To evaluate the effect of optimization I applied the following options:

1. `{ enabled: false, runs: 200 }`
2. `{ enabled: true, runs: 0 }`
3. `{ enabled: true, runs: 200 }`
4. `{ enabled: true, runs: 999999 }`

Further the contracts where compiled with ABI encoding v1 and v2.

5.1.1 Optimization statistics

To determine the relevant optimization options I calculated the following statistic 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 %

5.2 Wallets

820 wallet contract codes categorized into 38 types.

Wallets with the same type have the same interface.

`TODO:` How where the types assigned?

5.3 Proxies

32 proxy contract codes categorized into 9 types.

`TODO:` How where the types assigned?

6 Pre-Processing

6.1 Segmentation and Skeletonization [8]

6.1.1 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.

6.1.2 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.

6.2 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.

6.2.1 F-Statistic Filter

To determine significant opcodes I used the solc-version-testset[7].

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

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 [1])

7.2 ssdeep - Context Triggered Piecewise Hashes (CTPH)

ssdeep is based on spamsum[10] 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 [11]).

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.

7.2.2 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)
```

7.3 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))
```

7.4 LZJD - Binary-Hashing

This is how a cite[9] looks like.

7.5 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 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[6].

9 Results

`TODO:` artifacts, hypothesis

9.1 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?

9.2 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:` infestigate clusters within groups

9.3 Bytebag

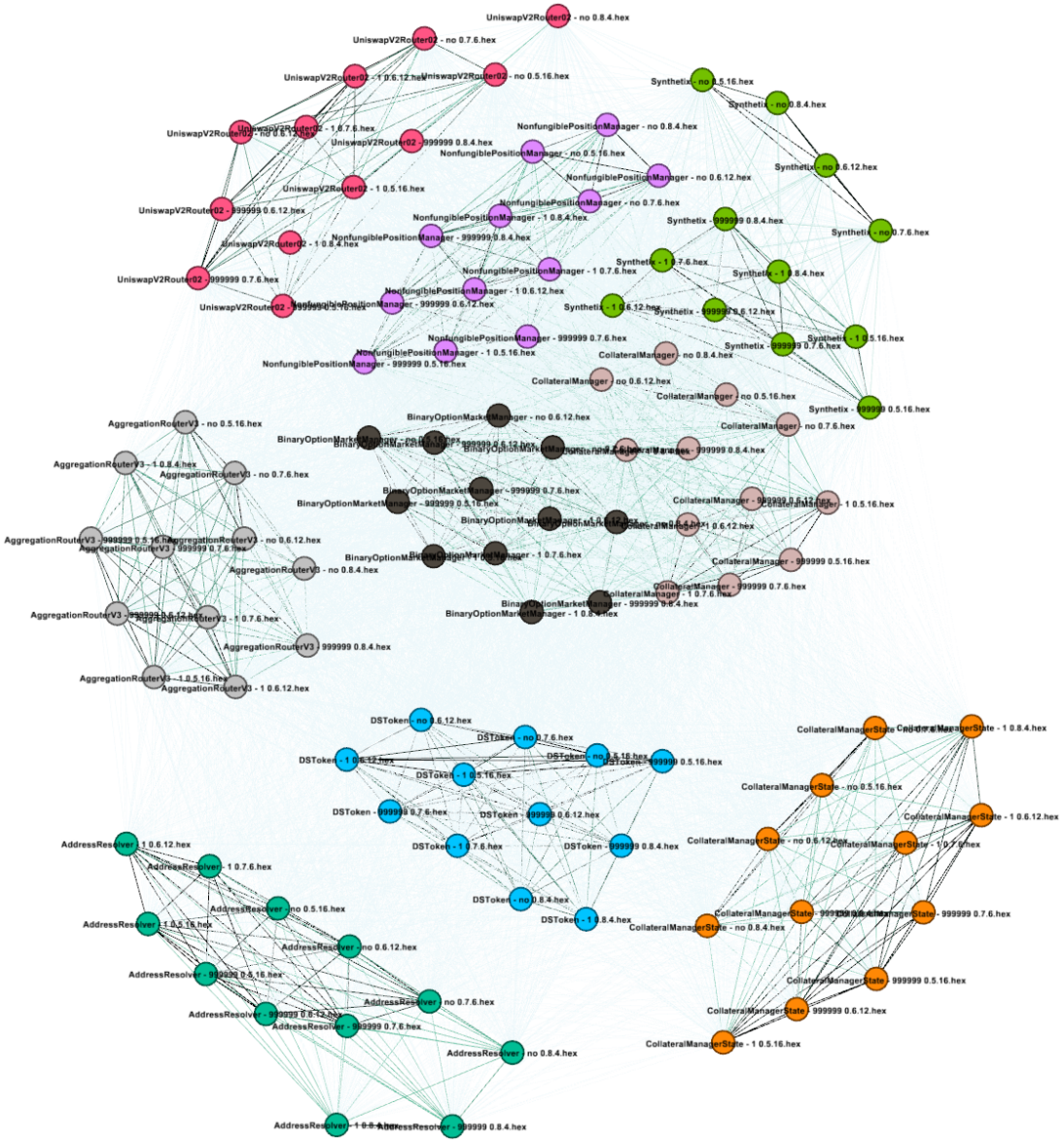
Combined with skeletonization and opcode filtering Bytebag performs better than ssdeep in some scenarios.

9.4 LZJD parameters

	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

9.5 ssdeep variants

9.6 F-Stat filter



10 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.

11 Conclusion

next steps

goals for followup papers

References

[1] 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.

[2] 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.

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

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

[5] 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.

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

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

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