Exploring Ethereum's Blockchain Anonymity Using Smart Contract Code Attribution

Shlomi Linoy
University of New Brunswick
Fredericton, Canada
slinoy@unb.ca

Natalia Stakhanova
University of Saskatchewan
Saskatoon, Canada
natalia@cs.usask.ca

Alina Matyukhina
University of New Brunswick
Fredericton, Canada
amatyukh@unb.ca

***Abstract*—Blockchain users are identified by addresses (public keys), which cannot be easily linked back to them without out-of-network information. This provides pseudo-anonymity, which is amplified when the user generates a new address for each transaction. Since all transaction history is visible to all users in public blockchains, finding affiliation between related addresses can hurt pseudo-anonymity. Such affiliation information can be used to discriminate against addresses that were found to be related to a specific group, or can even lead to the de-anonymization of all addresses in the associated group, if out-of-network information is available on a few addresses in that group. In this work we propose to leverage a stylometry approach on Ethereum's deployed smart contracts' bytecode and high level source code, which is publicly available by third party platforms. We explore the extent to which a deployed smart contract's source code can contribute to the affiliation of addresses. To address this, we prepare a dataset of real-world Ethereum smart contracts data, which we make publicly available; design and implement feature selection, extraction techniques, data refinement heuristics, and examine their effect on attribution accuracy. We further use these techniques to test the classification of real-world scammers data.**

***Keywords*-blockchain, Ethereum, smart contracts, distributed ledger, Ethereum, traceability, authorship attribution**

I. Introduction

With the rising popularity of blockchain technologies, their practical applications in the industry sectors are expanding as well (healthcare [1], government [2], IoT [3], insurance policies [4], and securities trading [5]).

Blockchain technologies enable different parties who do not trust each other to share information through the use of a robust consensus protocol which eliminates the need for a central authority. Although numerous blockchain platforms are currently available, Bitcoin [6] and Ethereum [7] remain the most popular, with a market capitalization of over $US\$15.5bn$ as of December 23rd, 2018 [8].

While Bitcoin is mainly a cryptocurrency, Ethereum enables, in addition to cryptocurrency, the creation and running of smart contracts (abbreviated as contracts) in a decentralized way. An Ethereum contract is a computer code that enables users to create their own arbitrary rules for ownership and state transition functions. The contract is written in a high level language such as Solidity and is compiled into EVM (Ethereum Virtual Machine) bytecode.

Since blockchain technologies are offering monetary value to their users, attacks on these platforms are mounting [9] [10]. The majority of these attacks are profit driven and leverage the fact that the identity of an adversary is hidden behind the account address. In fact, the pseudo-anonymity of an account address is one of the main premises of public blockchain platforms such as Ethereum, which is paramount since all transactions are publicly available on the blockchain. At this point, the only measure that allows combating the blockchain abuse is to drop transactions that contain suspicious account addresses from malicious users during the transaction validation. Yet, as users can and are encouraged to generate a new account address per transaction, detecting suspicious account addresses remains challenging.

In this work, we propose to leverage a stylometry approach to explore the extent to which a deployed contract's bytecode and its high level Solidity source code can contribute to the affiliation of account addresses. Since a deployed contract contains only its bytecode and does not preserve its source code, we obtain the contract's Solidity source code from `etherscan.io` [11], which enables a contract's author to upload the Solidity source code to their platform and link it to the deployed contract address on the blockchain.

Previous research on de-anonymization explored the affiliation of Bitcoin addresses by using out-of-network information such as IP addresses [12] [13], geo-locations [14], inner network information using graph analysis [15], and Bitcoin address classification techniques [16] [17]. We take an alternative approach and leverage the coding style of Ethereum smart contracts to attribute the deployed contracts' code to their developers' account addresses. Within this analysis, we take an insight from code authorship attribution techniques that allow the identification of a code's anonymous author based on the unique characteristics that describe the author's coding style and allow one to distinguish this author's code. This style can be expressed through layout (e.g., indentations, white spaces), lexical (e.g., function lengths, variable names) and semantic (e.g., control flow structure, AST depth) levels. Research on authorship attribution demonstrates the effectiveness of these types of features on accurate attribution of source codes [18] [19].

In Ethereum, the contract's bytecode preserves some of the stylistic properties, while the contract's Solidity source

978-3-903176-24-9

code preserves the full stylistic properties including the layout and lexical properties. However, the nature of Ethereum's source code introduces some challenges, including sensitivity to the source code's length and extensive code reuse (e.g., through common interfaces which are used extensively in smart contracts). To address these concerns, we introduce multiple data refinement heuristics which are based on the contract's Solidity source code components and a feature selection technique to extract a small portion of the features. We prepare a dataset of verified real-world contract data to which we apply these methods and analyze their effect on the attribution accuracy. In the analysis phase we focus on two commonly used approaches in code attribution: attribution based on n-grams and attribution based on a state-of-the-art feature set proposed by Caliskan et al. [19]. Our experimental results show that it is feasible to attribute Ethereum contracts to their corresponding deployers. We were able to achieve 93.5% accuracy in attributing source codes, using 3% of the total features, and 80% in attributing bytecodes using 13% of the total features. In addition, we apply one of our best performing heuristics to real world scammers data [20] and successfully attribute all of the contracts associated with a specific account address.

II. Related work

Blockchain security issues are manifold. Mauro Conti et al. [9] surveys attacks in Bitcoin which include: double spending, bribary attacks, brute force, transaction malleability, 50% hashpower, selfish mining, DDoS attack, and routing attacks to name a few. Xiaoqi Li et al. [10] surveys attacks in Ethereum which include: use of criminal smart contracts, exploitation of smart contracts' security vulnerabilities, and the DAO attack. In this paper we focus on privacy and anonymity issues in blockchain.

a) Blockchain addresses clustering: Reid et al. [21] analyzed the Bitcoin network with the use of two abstractions: "transaction network" and "user network". The "transaction network" shows the flow of Bitcoins from one transaction to the next over time, where each input edge to one transaction node is the output edge of the previous transaction. On the other hand, the "user network" shows the flow of Bitcoins from one user (payer) to another user (payee); each user is represented by a collection of his Bitcoin addresses. Since a user can have multiple unconnected Bitcoin addresses, there is no accurate process to determine which user is connected to which Bitcoin address. Spagnuolo et al. [22] propose to use a "change address", which is the address used to send any funds that remain at the end of a transaction, as a method to cluster bitcoin addresses to the same user. Chan et al. [17] explore the feasibility of affiliating Ethereum addresses using transaction graph analytics where transaction data are transferred into a graph database and tags are collected from sources such as `etherscan.io`. Addresses are considered to be affiliated if they share a transaction. Norvil et al. [23] explore unsupervised clustering techniques based on the ssdeep hash similarity of Ethereum contracts' bytecode. To find the context of a specific cluster the authors analyze the Solidity source code for each contract in the cluster and extract the most frequent tokens. While hash similarity operates on the contract code as a string and can be sensitive to code structure changes, our proposed approach considers code stylistic characteristic similarities. Attribution based on stylistic characteristics can provide high classification accuracy for contract codes that seem very different, as long as the stylistic features are similar. In addition, as the results of our work suggest, code reuse is prevalent in contracts' code, which can contribute to hash similarity bias. We further make use of heuristics that refine code similarities to reduce attribution bias.

b) De-anonymization of blockchain addresses using out-of-network information: Santamaria et al. [12] showed that publicly available metadata which are associated with the Bitcoin network can be used to obtain additional information regarding a Bitcoin address, e.g., in forums, users provide their Bitcoin address in a posted question or as part of their message signature, which associates the user's forum identity with his Bitcoin address. Meiklejohn et al. [16] use Bitcoin addresses from verifiable sources, e.g., goods vendors or exchanges, to follow the transactions and trends related to these Bitcoin addresses. Since Bitcoin operates on a P2P network, the transaction's issuer information can be obtained from the network infrastructure underlying the peer nodes, combined with information on the nodes that participate in the transaction relay. Koshy et al. [13] showed that anomalous transaction relays can be used to help connect an IP address to a Bitcoin address. Kaminsky [24] presents a method of de-anonymization that connects the transaction's input Bitcoin address to the IP address of the first relayer. This method can be actualized if the attacker can connect to all nodes in the Bitcoin network. However the anomalous transaction behaviour approach used by Koshy et al. [13] produces better de-anonymization results than that of the first relayer approach, although anomalous transaction behavior is much less frequent than non-anomalous. Biryukov et al. [25] discuss a de-anonymization method that uses entry nodes (all the peer nodes that the Bitcoin client is connected to) where, if an attacker is connected to an entry node, the IP address can be forwarded to him.

While the majority of research has been conducted on Bitcoin, it could be applicable to Ethereum as well. Klusman et al. [26] explored how the approaches proposed by Biryukov et al. [25] and Spagnuolo et al. [22] (both were discussed above) can be applied to Ethereum, with some changes to the Ethereum network.

No studies were conducted on de-anonymizing Ethereum addresses using authorship attribution on Ethereum contracts' code.

c) Authorship attribution: State-of-the-art methods in source code authorship attribution rely on low-level information such as word or character n-grams. Such features have been widely used in [27] [28] as they are able to capture stylistic information. At the binary level, the corresponding byte-level n-grams have been successfully explored [29] [30].

More complex features which require additional parsing of code were also explored although less frequently due to the overhead and complexity. There have been several attempts to utilize syntactic (structural) features for author attribution tasks. Most notable among these works is the study by Caliskan et al. [19] that adopted ASTs (abstract syntax trees) for attribution of code. In addition to syntactic features derived from AST, the authors use unigrams term frequency. In this work, we use a similar feature set, which is modified to support the Solidity source code. We refer to it as "Caliskan features". For more information on various authorship attribution methods and challenges we refer the interested reader to a survey by Vaibhavi et al. [31].

III. Smart contract attribution

Since anonymity is the main feature of a public blockchain technology, the Ethereum platform (as other blockchain implementations) is designed to maintain no personal identifiable information. The only source of data that can be directly linked to a blockchain user beyond the cryptographic keys are transactions information. Each transaction contains the addresses of the transaction's issuer and receiver. In the case when an account address executes a transaction to deploy a smart contract, the transaction will contain the account address and the deployed contract address. In our approach, we explore the feasibility of attributing deployed contracts' source codes and bytecodes to their deployers' account addresses. We turn our attention to code attribution research that showed the effectiveness of attribution techniques in their ability to identify an author of a given code based on extracted coding style. We examine the performance of two commonly used approaches in code attribution: attribution based on n-grams and attribution based on a customized feature set proposed by Caliskan et al. [19].

The process is shown in Figure 1 and includes feature extraction, feature selection, data refinement, and classification of bytecodes as well as their corresponding source codes. We note that data refinement in both source code and bytecode is a key step in reducing classification bias and depends exclusively on the contract's Solidity source code.

A. Ethereum code extraction

When a contract is deployed to the blockchain, only its bytecode is retained on the chain. To obtain the original source code, reverse engineering techniques can be applied on the bytecode to a limited degree, which is affected by the compilation process into the EVM bytecode, e.g., optimizing the contract's bytecode for performance may change the contract's code structure while maintaining the same functionality, human readable variable names are not required by the EVM and are encoded, and the layout information of the source code is removed. This makes the reverse engineering of the exact original source code challenging. Some tools were proposed to reverse engineer a contract's bytecode to a human readable source code with very limited results, e.g., porosity [32], and radare2 [33]. `etherscan.io` is an Ethereum blockchain explorer platform [11] which provides a range of capabilities that include the retrieval of contracts' source codes in specific cases, without the use of reverse engineering. It does so by providing an API to upload the source code to `etherscan.io`. The source code is then compiled into bytecode and is compared with the deployed contract's bytecode. If the two are equal, the contract's source code is considered verified. This manual pairing is entirely optional, and is not reflected in the Ethereum blockchain. In our analysis, we rely exclusively on verified contracts that provide both bytecode and source code representations. We further disassemble the bytecode into opcodes for easier parsing. Since Solidity is the most commonly used programming language for writing Ethereum contracts, we focus our analysis exclusively on contracts written in Solidity.

B. Data refinement heuristics

The tendency of programmers to reuse their own code components and those written by others can negatively affect the code's classification accuracy [34] [35]. Solidity, like most programming languages, enables the creation of code libraries for common algorithms reuse [36]. These libraries can be created in a local or a remote contract. In the various Solidity contracts' source codes that we examined, we located only a handful of library calls. Instead, the common approach is to copy necessary code into a contract and modify it as required. We examined the top 10 similarities in our Solidity source codes' dataset and found that the common components include (in descending order) ERC23 contract interfaces, safe math libraries, ownership contract implementations, ERC20 contract implementations, token recipient interface implementations, and pausable interface implementations. The results show that most similarities are attributed to token standards implementations (e.g., the evolution of token standards from EC20 to EC23 which introduced a new code template to copy and reuse) as well as a generic safe math functionality. This introduces a large amount of duplicate code and increases the similarity between contracts.

To avoid unnecessary attribution bias and reduce the amount of common code in the contracts, we consider several data refinement heuristics. These heuristics depend on the contract's Solidity source code and provide the code similarity assessment at the component granularity level, such as contracts, libraries, and interfaces. Note that Ethereum supports inheritance and therefore a single program's source code may include multiple contracts that inherit characteristics from each other.

Before the refinement is applied, each contract's source code is split into its individual components. Similarity between splits is assessed using the Levenshtein distance metric. To reduce the number of pairwise comparisons, each split is compared to another split only if their size differs by at least 10%. Two compared splits are considered similar if their similarity score is 80% or higher. From each similarity group we retain a single split in an arbitrary way and delete all other similar splits according to one of the following heuristics:

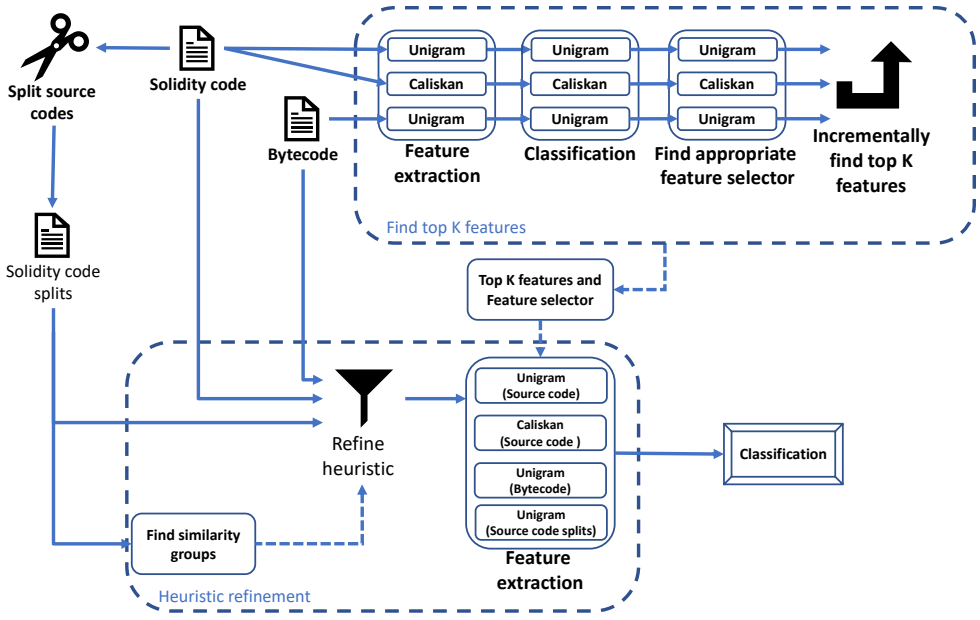


Fig. 1: The high-level flow of attributing Ethereum smart contacts

1) Considers splits' similarity between authors as well as within an author - if any contract's split was found to be similar, and hence removed, all of the contract's splits are removed as well. The remaining splits belong to contracts without any similarities. All the remaining splits per contract are merged to produce a single compilable contract source code.
2) Considers splits' similarity between authors as well as within an author - if any contract's split was found to be similar, and hence removed, all of the contract's splits are removed as well. The remaining splits belong to contracts without any similarities. Each split is treated as a full contract source code for classification purposes.
3) Considers splits' similarity between authors as well as within an author - Only contracts which had all of their splits removed (due to being similar to other splits) are deleted. The remaining splits represent full or partial contracts without any similarities. All the remaining splits per contract are merged to produce a full or partial contract source code which may be non-compilable due to its partialness. This can prevent the extraction of ASTs that are required for the Caliskan feature extraction. For this reason we only extract unigram features. For each merged source code we retrieve the opcodes corresponding to the original full source code before the splits' removal.
4) Considers splits' similarity between authors as well as within an author - Only contracts which had all of their splits removed (due to being similar to other splits) are deleted. Each split is treated as a full contract source code for classification purposes.
5) Considers splits' similarity only between authors - if any contract's split was found to be similar, and hence removed, all of the contract's splits are removed as well. The remaining splits belong to contracts without any similarities. All the remaining splits per contract are merged to produce a single compilable contract source code.
6) Considers splits' similarity only between authors - if any contract's split was found to be similar, and hence removed, all of the contract's splits are removed as well. The remaining splits belong to contracts without any similarities. Each split is treated as a full contract source code for classification purposes.
7) Considers splits' similarity only between authors - Only contracts which had all of their splits removed (due to being similar to other splits) are deleted. The remaining splits represent full or partial contracts without any similarities. All the remaining splits per contract are merged to produce a full or partial contract source code which may be non-compilable due to its partialness.
8) Considers splits' similarity only between authors - Only contracts which had all of their splits removed (due to being similar to other splits) are deleted. The remaining splits represent full or partial contracts without any similarities. All the remaining splits per contract are merged to produce a full or partial contract source code which may be non-compilable due to its partialness.

C. Feature extraction

For our analysis we adopt two feature sets that are widely used in code attribution studies and which are treated as benchmark sets: n-gram features employed by [37]–[40], and a feature set derived by Caliskan et al. [19], which is used in the majority of recent studies [18] [41].

N-grams are derived at the lexical level and thus primarily

TABLE I: Dataset statistics

	Author count	Contract count	Avg samples per author/(std)	Avg source code len/(std)	Avg source code LOC/(std)	Avg byte code len/(std)	Min contracts/ author
Selected4contracts	1071	8915	8.32 / (12.19)	114767.76 / (196095.32)	3330.93 / (5694.51)	185421.86 / (287453.45)	4

represent layout characteristics of code. While the Caliskan et al. [19] feature set includes lexical features (e.g., indentation, white space use, braces placements, statistical distribution of variable lengths and capitalization), and syntactic features that outline the external structural organization of the code and include features which are derived from an AST, e.g., code length, nesting levels, and branching.

To obtain n-gram features we tokenize both source code files and opcode files (extracted from bytecodes) using space, carriage return, new line, and tab. The Caliskan et al. feature set was originally developed for C and C++ programs; we thus map these features to the corresponding Solidity features.

D. Classification

The Random Forest algorithm has recently been gaining popularity in code authorship attribution [19]. The algorithm, which is an ensemble of decision trees, performs well in the scenarios where the goal is to identify the most likely author of a code fragment. Further, once trained, it provides the best trade-off between accuracy and processing time, hence outperforming other models, including neural networks.

IV. Data

Since anonymity is the main feature of all blockchain technologies, there are no datasets available for research that identifies users and their corresponding transactions or contracts (in the case of Ethereum). As a result there is no "ground truth" data for our experimentation purposes. To ensure a comprehensive evaluation of the feasibility of the proposed approach, we constructed a validation dataset with known relations between users (as represented by their account address) and contracts. Generally, it is possible for an individual user to generate multiple transactions under different keys. In this work we adopt a conservative approach and only collect users with multiple contracts issued under the same key. For our analysis, we collected 21,825 verified contracts by crawling `etherscan.io`, which we made publicly available[1]. We employed a web crawler to scan the verified contracts' HTML and retrieved their source code with the retained layout and lexical features, which are critical for the source code authorship attribution. For each of the retrieved contracts, we also extracted the corresponding bytecode and disassembled it to obtain its opcodes.

Our dataset contained a large number of authors with a single contract. Since having such a limited number of contracts limits the possibility of validating attribution results, we filter out authors that have fewer than 4 contracts. Table I shows the statistics of our resulting dataset.

[1]https://github.com/shomzy/Lino1910

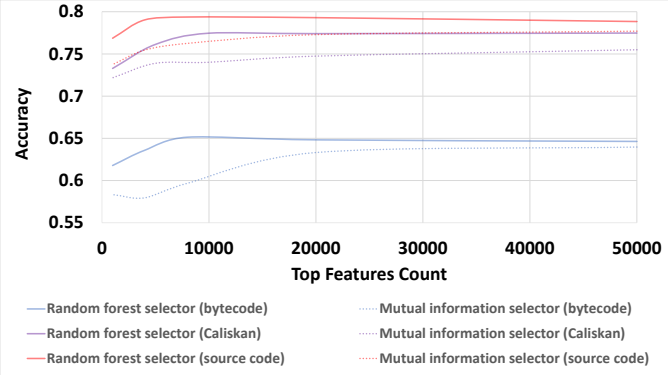


Fig. 2: Feature selectors comparisons

V. Evaluation

We implement the proposed attribution approach using Python's scikit-learn module [42]. We use a Random Forest classifier with a 4-fold stratified cross validation strategy.

The objectives of our experimental evaluation are:

1) Understand the effectiveness of attribution features for Ethereum contracts' attribution.
2) Explore the ability of our approach to accurately attribute contacts to known authors.

A. Data refinement

Table II shows the effects of the refinement heuristics on our dataset. One noticeable difference between each heuristic is the amount of retained data. Heuristics in which similarities are considered only between authors find less common code between contracts and as a result retain more splits. On the other hand, heuristics in which similarities are considered within authors find more common code between contracts. A possible explanation is that authors are more likely to reuse their own code, e.g., to write and deploy different versions of the same contract, or to reuse their own components.

The least amount of code is retained with *heuristic 1*. This is the most conservative approach that removes all similarities between, as well as within authors and as a result maintains the least number of authors. Note that for our analysis we choose to retain authors that have at least 4 contracts after applying the heuristic.

On the other extreme, *heuristic 8* provides the highest data retention. In this approach, only similarities between authors are removed. Each of the remaining splits is being treated as a "full contract", which results in retaining more authors with at least 4 contracts. This heuristic can be used in cases where obtaining sufficient data is not feasible. Yet, even with this

TABLE II: Heuristics refinement details

Heuristic	Program type	Feature type	Retained data %	Author count	Avg samples per author/(std)	Source code avg size/(std)	Source code avg LOC/(std)	Top/total features
1	Bytecode	Unigram	0.50%	34	6 / (2.48)	94371 / (80672.32)	-	1500 / 11590
	Source code	Unigram	0.50%	34	6 / (2.48)	45162.41 / (53276.11)	1208 / (1345.128)	647 / 21550
	Source code	Caliskan	0.50%	34	6 / (2.48)	45162.41 / (53276.11)	1208 / (1345.128)	700 / 34987
2	Source code splits	Unigram	1%	76	6.88 / (3.433)	22917.43 / (26892.87)	663.67 / (761.45)	1000 / 29623
3	Bytecode	Unigram	3.50%	283	6.24 / (3.723)	149305.82 / (127928.95)	-	6000 / 42768
	Source code	Unigram	3.50%	283	6.24 / (3.723)	41467.36 / (39868.53)	1186 / (1070.32)	5000 / 135563
4	Source code splits	Unigram	9%	475	9.65 / (7.893)	33281.7 / (34426.26)	958.05 / (929.51)	6000 / 175427
5	Bytecode	Unigram	1.80%	106	8.39 / (9.173)	144171.77 / (195275.03)	-	3000 / 21422
	Source code	Unigram	1.80%	106	8.39 / (9.173)	76077.88 / (142905.22)	2091.55 / (3900.64)	1000 / 49030
	Source code	Caliskan	1.80%	106	8.39 / (9.173)	76077.88 / (142905.22)	2091.55 / (3900.64)	3000 / 91315
6	Source code splits	Unigram	4.20%	136	15.48 / (23.673)	59369.63 / (127319.81)	1660.75 / (3498.86)	1600 / 52515
7	Bytecode	Unigram	9.13%	599	7.66 / (7.028)	180939.86 / (212620.36)	-	8000 / 57508
	Source code	Unigram	9.13%	599	7.66 / (7.028)	66135.62 / (108244.7)	1894.51 / (3138.6)	6000 / 196478
8	Source code splits	Unigram	23.50%	658	17.94 / (29.373)	61984.62 / (104176.03)	1787.22 / (3040.22)	7000 / 203899

liberal approach, the resulting set contains only 658 authors which are 61% of the original dataset.

B. Feature selection

All feature sets that we employ in our study generate large and sparse feature vectors mostly due to their heavy use of unigram term frequencies. In many cases, such feature vectors lead to over-fitting. To avoid biased classification results, we examine two feature selection methods: Mutual information analysis and Random Forest importance-based.

Figure 2 shows that Random Forest importance-based feature selection consistently provides up to 5% higher classification accuracy compared to mutual information analysis. We therefore apply this method to further reduce the number of features for our analysis.

In order to find the minimal feature set that provides sufficient classification accuracy, we employ the following method: we incrementally select the top K ranked features and calculate the classification's accuracy based on 4-fold cross validation. The results are given in Figure 3. It is clear from the analysis that the accuracy quickly plateaus for all feature sets. The best trade-off between the number of features (hence size of corresponding feature vectors) and accuracy happens at 8000 top ranked features for opcode (bytecode) unigrams (13% of total features), 8000 top ranked features for source code unigrams (3% of total features), and 10000 top features for the Caliskan feature set (2% of total features). Table III shows the classification results, before and after applying a feature selection, per feature set. Although the accuracy does not change, the number of features retained for attribution analysis is significantly smaller. We use these top ranked feature ratios for our following analysis.

TABLE III: Evaluation results before and after feature selection

Program type	Feature set	Total features	Accuracy	Top ranked features	Accuracy
Source code	Unigram	231553	78.04%	8000 (3%)	79.77%
Source code	Caliskan	441963	76.52%	10000 (2%)	77.87%
Bytecode	Unigram	63363	64.05%	8000 (13%)	65.69%

C. Attribution results

We perform the following per heuristic: refine the dataset as dictated by the heuristic, extract the feature sets using the Random Forest importance-based feature selector, and refine the top features according to the ratios described above. Table II provides the statistics' details.

We next perform a 4-fold stratified cross validation on each of the refined feature sets, using a Random Forest classifier. Table IV shows the attribution accuracy results per heuristic. The column "Accuracy after feature selection" refers to the accuracy on the refined feature set which was extracted from Selected4contracts (rightmost column in Table III).

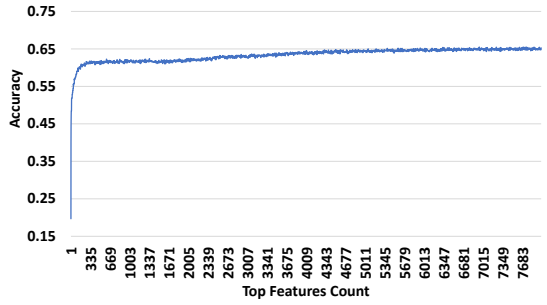
In most heuristics, the removal of similar code from the contracts results in a significantly higher attribution classification rate as compared to before applying the heuristics (as summarized in column "accuracy after feature selection"). The highest rate (93.5%) was achieved with source code unigrams, and the second best result was obtained with the Caliskan's feature set (90.9%). The heuristics that produce partial contracts provide significantly more data. However, the type of analysis that can be performed is limited since no corresponding partial bytecode, or ASTs (used by the Caliskan's feature set) can be extracted. These cases are labelled with a dash '-'.

Heuristic 5 shows the highest accuracy in the overall categories (excluding source code splits). It does this at the cost of removing all contracts that have any similarities, which results in the removal of many authors. *Heuristic 7* has the second best performance. It removes only the similar components of the source code and merges the remaining components. Although it provides a slightly lower accuracy than *Heuristic 5*, it retains 6 times more authors.

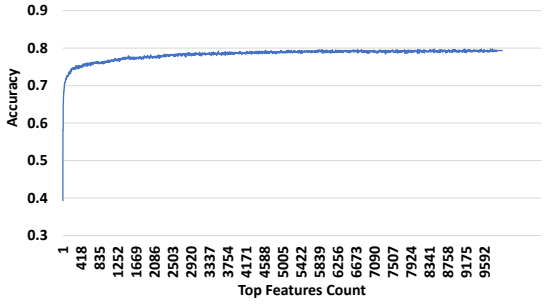
Comparing heuristics, we see that caution should be exercised as to which similarities should be removed. The single refinement property that distinguishes *heuristics 1-4* and *heuristics 5-8* is where to search for similarities. The former searches for similarities between, as well as within authors while the latter searches for similarities only between authors.

TABLE IV: Classification accuracy per heuristic

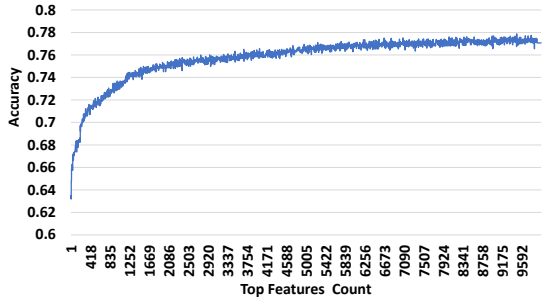
Program type	Feature set	Accuracy after feature selection	Heuristic accuracy							
			1	2	3	4	5	6	7	8
Bytecode	Unigram	65.69%	57.69%	-	57.09%	-	79.98%	-	75.18%	-
Source code	Unigram	79.77%	81.79%	-	72.31%	-	93.51%	-	88.47%	-
Source code	Caliskan	77.87%	78.61%	-	-	-	90.91%	-	-	-
Source code splits	Unigram	-	-	68.78%	-	60.38%	-	90.62%	-	87.23%



(a) Byte code



(b) Source code



(c) Caliskan

Fig. 3: Finding top ranked features using incremental steps

The preservation of similarities within the authors results in features that more accurately capture each author's preference of using specific code constructs over others. Further, refining less data results in additional and larger samples, which also contributes to higher accuracy.

We note that, while the approach that is based on the n-gram features provide the higher accuracy, it is also susceptible to manipulations of coding constructs' names. In contrast, the approach that is based on the Caliskan features supports more robust features such as ASTs which can be more resilient in such cases at the cost of a slightly lower classification accuracy.

VI. Attribution of Scams

The results show the effectiveness of our approach for attributing unknown smart contracts to their corresponding authors. To examine our approach in the underground scammers' community, we further explore the attribution of real-world Ethereum scams.

Etherscamdb [20] is an open source database that keeps track of all the current Ethereum scams. The scam information in the site is diverse and contains scam categories like Fake ICO, Phishing, Scamming, and Scam. We extract contracts based on the account's address and the contract's address. For a given malicious contract's address, we extract all contracts which were deployed by the contract's deployer. Symmetrically, for each malicious account address, we extract all of its deployed contracts (if any) and consider them to be malicious (see Algorithm 1).

Algorithm 1 Retrieve scammers data from EtherscamDB

Require: $ScmDB_{addresses} = (a_1, a_2, ...)$ - Etherscamdb's malicious addresses;
Ensure: $Scm_{authors} = (au_1, au_2, ...)$ - contract related malicious authors. Each author contains $SC = (sc_1, sc_2, ...)$ were sc_i is a contract's address;
$Scm_{authors} \Leftarrow \{\}$
$A \Leftarrow \{\}$
for all a_i in $ScmDB_{addresses}$ **do**
  if a_i is account address **then**
    $A.Add(a_i)$
  else if a_i is a contract's address **then**
    $T \Leftarrow etherscan.get\_all\_transactions(a_i)$
    $a_{addr} \Leftarrow get\_contract\_creator\_account\_address(T)$
    $A.Add(a_{addr})$
  end if
end for
for all a_i in A **do**
  $T \Leftarrow etherscan.get\_all\_transactions(a_i)$
  $a_{contracts} \Leftarrow get\_contract\_creation\_addresses(T)$
  $Scm_{authors}.Add(\{a_i, a_{contracts}\})$
end for
return $Scm_{authors}$

To fetch the account address of a malicious contract's deployer or the contract's addresses deployed by a malicious account address, we use `etherscan.io`. To implement

TABLE V: Scammers dataset

	Num of authors	Total num of contracts	Max contracts/ author	Avg num of contracts/ author	Source code avg bytesize	Source code avg LOC
source code/ opcode	22	68	20	3.09	7379.98	212.91

the algorithm, for each address that was extracted from Etherscamdb, we use `etherescan.io`'s API[2] to find all the transactions that are related to a requested address. We provide a malicious address and receive a transaction list. For each transaction we examine the results field. An empty "To" field indicates that the transaction is used to deploy a contract whose address is provided in the "contractAddress" field. When "contractAddress" is the same as the provided malicious address, this indicates that the provided address is a contract address. If it is different, this indicates that the provided malicious address is an account's address and the "contractAddress" value is the deployed contract's address. If the results field is empty, an account or contract was not found at the specified address and is ignored.

The algorithm's result set contains all malicious related authors with their deployed contracts' addresses. For each of the contract's addresses, we extract its opcode and source code (if available in `etherscan.io`'s verified contracts). The resulting scammers dataset are shown in Table V.

We cross referenced the scammers data and our Selected4contracts, presumably, benign dataset. One account's address (*0x0042bd345e43bd151fa563c2bc8fa22bda507104*) was contained in both datasets. It contained 8 verified contracts in our collected scammers dataset of which, 5 contracts were found in our Selected4contracts' benign dataset. The 5 shared contracts as well as the 3 unseen contracts were attributed correctly in the source code and bytecode datasets.

VII. Conclusions

Blockchains provide pseudo-anonymity for users by providing public addresses which can not be easily linked back to the users. In this work, we propose to leverage a stylometry approach to explore the extent to which a deployed smart contract's source code can contribute to the affiliation of account addresses. To address this, we prepare a dataset of real-world contract data; design and implement feature selection, extraction techniques, data refinement heuristics; and examine their effect on attribution accuracy. We further use these techniques to test the classification of real-world scammers data. Our experimental results show that it is feasible to attribute Ethereum contracts to their corresponding deployers. We were able to achieve 93.5% accuracy in attributing source codes, using 3% of the total features, and 80% in attributing bytecodes using 13% of the total features. In addition, we apply one of our best performing heuristics to real world scammers data and successfully attribute all of the contracts associated with a specific account address.

[2]api.etherscan.io/api?module=account&action=txlist&address=$\langle address \rangle$

References

[1] Blockchain applications for the modern nation. [Online]. Available: https://cryptodaily.co.uk/2018/03/blockchain-applications/

[2] How ibm blockchain can improve government services and ensure trust. [Online]. Available: https://www.ibm.com/downloads/cas/2VNRQX9V

[3] Blockchain has grabbed the attention of investors. [Online]. Available: https://www.cnbc.com/2018/04/02/blockchain-has-grabbed-the-attention-of-investors.html

[4] V. Gatteschi, F. Lamberti, C. Demartini, C. Pranteda, and V. Santamaría, "Blockchain and smart contracts for insurance: Is the technology mature enough?" *Future Internet*, vol. 10, no. 2, p. 20, 2018.

[5] Three near-term applications for blockchain technology. [Online]. Available: https://www.forbes.com/sites/forbesfinancecouncil/2018/03/28/three-near-term-applications-for-blockchain-technology/2/#6c9f49c6310d

[6] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[7] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[8] coinmarketcap. coinmarketcap. [Online]. Available: https://coinmarketcap.com/

[9] M. Conti, S. Kumar, C. Lal, and S. Ruj, "A survey on security and privacy issues of bitcoin," *IEEE CST*, 2018.

[10] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *FGCS*, 2017.

[11] etherscan.io. [Online]. Available: https://etherscan.io/

[12] M. Santamaria Ortega, "The bitcoin transaction graph anonymity," 2013.

[13] P. Koshy, D. Koshy, and P. McDaniel, "An analysis of anonymity in bitcoin using p2p network traffic," in *ADCS*. Springer, 2014, pp. 469–485.

[14] J. DuPont and A. C. Squicciarini, "Toward de-anonymizing bitcoin by mapping users location," in *CODASPY*. ACM, 2015, pp. 139–141.

[15] D. Ron and A. Shamir, "Quantitative analysis of the full bitcoin transaction graph," in *ADCS*. Springer, 2013, pp. 6–24.

[16] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage, "A fistful of bitcoins: characterizing payments among men with no names," in *IMC*. ACM, 2013, pp. 127–140.

[17] W. Chan and A. Olmsted, "Ethereum transaction graph analysis," in *12th ICITST*. IEEE, 2017, pp. 498–500.

[18] E. Dauber, A. Caliskan, R. Harang, and R. Greenstadt, "Poster: Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments," in *IEEE/ACM 40th ICSE-Companion*, 2018, pp. 356–357.

[19] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX*, 2015, pp. 255–270.

[20] Etherscamdb. [Online]. Available: https://etherscamdb.info/

[21] F. Reid and M. Harrigan, "An analysis of anonymity in the bitcoin system," in *Security and privacy in social networks*. Springer, 2013, pp. 197–223.

[22] M. Spagnuolo, F. Maggi, and S. Zanero, "Bitiodine: Extracting intelligence from the bitcoin network," in *ADCS*. Springer, 2014, pp. 457–468.

[23] R. Norvill, B. B. F. Pontiveros, R. State, I. Awan, and A. Cullen, "Automated labeling of unknown contracts in ethereum," in *26th ICCCN*. IEEE, 2017, pp. 1–6.

[24] D. Kaminsky, "Black ops of tcp/ip," *Black Hat USA*, p. 44, 2011.

[25] A. Biryukov, D. Khovratovich, and I. Pustogarov, "Deanonymisation of clients in bitcoin p2p network," in *SIGSAC CCS*. ACM, 2014, pp. 15–29.

[26] R. Klusman and T. Dijkhuizen, "Deanonymisation in ethereum using existing methods for bitcoin," 2018.

[27] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas, "Source code author identification based on n-gram author profiles," *AIAI*, pp. 508–515, 2006.

[28] G. Frantzeskou, S. MacDonell, E. Stamatatos, and S. Gritzalis, "Examining the significance of high-level programming features in source code author classification," *Journal of Systems and Software*, vol. 81, no. 3, pp. 447–460, 2008.

[29] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas, "Effective identification of source code authors using byte-level information," in *ICSE*. ACM, 2006, pp. 893–896.
[30] G. Frantzeskou, E. Stamatatos, S. Gritzalis, C. E. Chaski, and B. S. Howald, "Identifying authorship by byte-level n-grams: The source code author profile (scap) method," *IJDE*, vol. 6, no. 1, pp. 1–18, 2007.
[31] V. Kalgutkar, R. Kaur, H. Gonzalez, N. Stakhanova, and A. Matyukhina, "Code authorship attribution: Methods and challenges," *CSUR*, vol. 52, no. 1, p. 3, 2019.
[32] Porosity. [Online]. Available: https://github.com/comaeio/porosity
[33] Radare2. [Online]. Available: https://github.com/radare/radare2
[34] S. Burrows, A. L. Uitdenbogerd, and A. Turpin, "Comparing techniques for authorship attribution of source code," *spe*, vol. 44, no. 1, pp. 1–32, 2014.
[35] N. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? identifying the authors of program binaries," in *ESORICS*. Springer, 2011, pp. 172–189.
[36] Solidity library calls. [Online]. Available: https://solidity.readthedocs.io/en/develop/contracts.html#libraries
[37] J. Kothari, M. Shevertalov, E. Stehle, and S. Mancoridis, "A probabilistic approach to source code authorship identification," in *ICIT*. IEEE, Apr. 2007, pp. 243–248.
[38] M. F. Tennyson and F. J. Mitropoulos, "Choosing a profile length in the scap method of source code authorship attribution," in *SoutheastCon*. IEEE, Mar. 2014, pp. 1–6.
[39] M. Shevertalov, J. Kothari, E. Stehle, and S. Mancoridis, "On the use of discretized source code metrics for author identification," in *SSBSE*. IEEE, May 2009, pp. 69–78.
[40] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas, *Source code author identification based on N-gram author profiles*, ser. IFIP. Springer, Jun. 2006, vol. 204.
[41] L. Simko, L. Zettlemoyer, and T. Kohno, "Recognizing and Imitating Programmer Style: Adversaries in Program Authorship Attribution," *PoPETs*, no. 1, pp. 127 – 144, 2018.
[42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.