View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Open Repository and Bibliography - Luxembourg

Standardising smart contracts: Automatically inferring ERC standards

Robert Norvill
SEDAN group, SnT
University of Luxembourg
29 Avenue John F. Kennedy
L-1855 Luxembourg
robert.norvill@uni.lu

Beltran Fiz
SEDAN group, SnT
University of Luxembourg
29 Avenue John F. Kennedy
L-1855 Luxembourg
beltran.fiz@uni.lu

Radu State
SEDAN group, SnT
University of Luxembourg
29 Avenue John F. Kennedy
L-1855 Luxembourg
radu.state@uni.lu

Andrea Cullen
EECS
The University of Bradford
Richmond Road Bradford
BD7 1DP, UK
a.j.cullen@bradford.ac.uk

***Abstract*—Ethereum smart contracts have become common enough to warrant the need for standards to ensure ease of use. The most well known standard was created for the emerging token ecosystem and the exchanges serving it: the ERC20 standard.**

In this work we use the function selectors present in Ethereum smart contract bytecode to define contract purpose. Contracts are clustered according to the selectors they have. A Reverse look-up from selectors to function names is used to label clusters. We use the function names in clusters to suggest candidates for ERC standardisation.

I. Introduction

Ethereum is the world's most popular blockchain for smart contract utilisation. It can be thought of as a blockchain based, decentralised computer, with smart contracts as the programs it is capable of executing. Ethereum smart contracts are widely used and can hold a high value in tokens or ether. The results of smart contract execution are agreed upon by consensus.

Ethereum bytecode consists of opcodes and static values. The Ethereum Virtual Machine (EVM) reads and operates on these opcodes and values.

The availability of smart contract source code is entirely at the discretion of the developer. The bytecode is always publicly visible, so that nodes can execute it. However, the source code is unavailable in the majority of cases. At the time of writing [1] has verified source code for 49570/46338837 contracts on the public blockchain.

Ethereum has a growing number of Ethereum Improvement Proposals (EIP's). Ethereum Request for Comments (ERC's) are a subset of EIP's, most of which define templates for contracts, for a given purpose. The most popular is currently ERC20 [2]. It provides a standardised set of functions that must be implemented for tokens in Ethereum. ERC's only detail the functions that must be implemented, not the semantics of implementation. In keeping with this approach, we consider function names, and not function content.

In this paper we aim to provide insight into the purpose of contracts for which the source code is unavailable. Standardisation can be a lengthily process, and is often a reaction to an existing need. We aim to provide a method to speed up the process, by early identification of potential candidates for new ERC's. Our approach can be used to identify where the community could focus its efforts.

In Ethereum smart contracts, function selectors are 4 byte identifiers created from the keccak256 hash of the function name and parameter types. They are stored in a contract's bytecode as static values and used to identify which function is being called. We utilise selectors by treating those derived from functions in ERC's and those present in contracts, as sets which define contracts. The sets of selectors are used to cluster similar contracts. A reverse look-up is used to label clusters. Selectors are guaranteed to be unique within a contract and are universally uniform. A contract implementing an ERC must have the selectors of that ERC.

II. Related Work

Ethereum's yellow paper is frequently updated. It clearly defines the opcodes and their behaviour [3]. Such information is necessary when working with bytecode.

Etherscan is a web-based blockchain explorer for Ethereum [1]. It verifies contracts by compiling the source code provided by developers and checking that the bytecode matches that which is stored on the blockchain. Etherscan can label smart contracts as ERC20 tokens without the source code. We speculate that they may identify these contracts using features common to all ERC20 contracts, such as the function selectors.

Various projects and papers have focused on Ethereum bytecode. They fall into two main categories, those dealing with the monetary cost of executing smart contracts, and those dealing with contract security. Those dealing with cost include: [4], [5], [6], [7], [8]. The security focused projects include: [9], [10], [11].

In this work we make use of the function selectors found in complied smart contracts. The website found at [12] provides access to a database of function selectors. It is populated with user provided functions which match a given selector. The entire database is available at [13]. We use this database to perform the reverse look-up from selectors to function names. We use the acquired function names to define cluster purpose.

978-1-7281-1328-9/19/$31.00 ©2019 IEEE

Lastly, in our previous work we used clustering to identify contract purpose [14]. The whole bytecode of each contract was used. We employed identification methods used in malware detection and used Etherscan's verified contracts to label clusters. In this work we improve our accuracy by focusing on the selectors.

III. Dataset

The raw data consists of the 1499926 contracts from Ethereum's public chain, between blocks 46402 and 5609706. We extract the function selectors from each contract and treat them as elements of a set which represents the contract. Duplicates and contracts without selectors are removed. Duplicates are defined as contracts sharing the exact same set of selectors. If A and C are sets of selectors defining contracts then C is a duplicate if $A = C$

Timestamps in the raw data range from Aug-07-2015 04:42:15 AM +UTC to May-14-2018 01:52:40 AM +UTC. The raw data contains 125611 contracts which implement ERC's. There are 68 ERC's listed at [15], of which our dataset contains 3. In order of prevalence they are: ERC20, ERC173, ERC721. ERC's 20 and 721 have both required and optional functions in their definitions. We represent this as two sets. One, a required set of selectors, that a contract must have in order to implement the ERC. The other, a set of optional selectors, that a contract may contain zero or more of. ERC20 and ERC721 are finalised, while ERC173 is still a draft.

With duplicates removed we have a total of 22800 unique contracts, which we use for clustering. The number of duplicates each contract has is recorded and used as weight when clustering.

IV. Methodology

In this section we discuss the steps taken to build our dataset, and the clustering methods employed.

In Ethereum, function selectors are used to identify which function is being called. They are generated from the canonical form of the function, which is the function name followed by the type of each parameter, separated by commas, with no white space. For example, for ERC20:

```
function transfer(address _to, uint256 _value)
public returns (bool success)
```

has the canonical form: `transfer(address,unit)`

The canonical form must be hashed using the keccak256 function (SHA3). The selector is taken to be the first 4 bytes of the hash, encoded as a hexadecimal number. We generate the functions selectors for ERC's to check for their presence in our dataset. Selectors are stored in contract bytecode to allow the given function to be selected when calling the contract.

The input for the hash includes the data types of parameters, allowing for overloading. This ensures that functions with the same name and different parameters will have unique selectors. As the process produces uniform output, selectors will be the same for functions with the same name and parameters across contracts. This makes selectors a natural fit for representation as elements of sets. Set elements must be unique, a property each contract already has. As such, each ERC and contract can be represented by a set of its selectors. We are able to validate our clustering through the use of set operations. Any contract implementing an ERC must contain the selectors of that ERC. In set theory terms a contract selector set A must be a superset, of an ERC selector set C. If A implements C then: $A \supseteq C$

Our dataset consists of a set of selectors per unique contract in the raw data. Selectors are extracted from the bytecode using pattern matching.

```
PUSH4 0xdd62ed3e EQ PUSH2 0x0116 JUMPI DUP1
```

We identify the selectors using regular expressions to match the bytecode around each selector. An example with the selector of the first ERC20 function can be seen above, the selector is highlighted in bold. The extracted selectors are stored as one set per contract.

The K-means clustering algorithm is used to cluster contracts based on the selectors present in their bytecode. The contracts are weighted according to the number of duplicates they have.

We run K-means with different values of k: 2, 5 and 9. ERC20 is used to validate our results. The ERC20 contracts should be clustered together. We record the percentage of each cluster which is made up of ERC20 contracts. For each cluster the top 10 most commonly occurring selectors are extracted, and a reverse look-up from selectors to functions names is carried out using the database available from the 4byte website [13]. The top selectors are used to provide suggestions for new ERC's.

We use Single Value Decompression (SVD), as implemented in the Python's sklearn library [16], to visualise the result of the clustering process.

V. Experimental Results

In this section we discuss the results of the clustering, and label clusters according to their member's purpose. Purpose is derived from the function names in each cluster. We detail the results for k=2 and show how we are able to make further recommendations by increasing the value of k. When discussing the contracts in a cluster we refer to the unique contracts that make up our dataset, unless stated otherwise.

A. k=2

For k=2, c0 represents non-ERC20 contracts and c1 contains the ERC20 contracts. c0 contains 16471 contacts and c1 contains 5817. The percentage of ERC20 contracts in each cluster is 1.3 and 93.3, respectively. With k=2 we can identify ERC20 contracts with a high level of accuracy.

B. k=5

By increasing the value of k to 5 we begin to identify candidates for new ERC's. Fig. 2 shows the new clusters. Notably, the cluster on the left hand side has split in two. c4 contains the ERC20 contracts. It is larger than its equivalent for k=2, containing 7490 unique contracts, 75.4% of which are

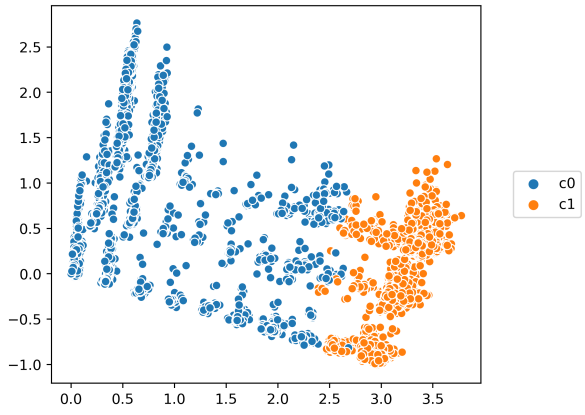

Fig. 1: Visualisation for clustering for k=2

full ERC20 implementations. It contains a number of partial ERC20 implementations, with 7399 contracts containing the required ERC20 function `balanceOf()` and 6174 containing the required function `allowance(address,address)` (1225 less). The optional ERC20 function `symbol()` appears 7209 times, 1035 times more than the required allowance function. This suggests a large number of different ERC20 hybrids have been implemented.

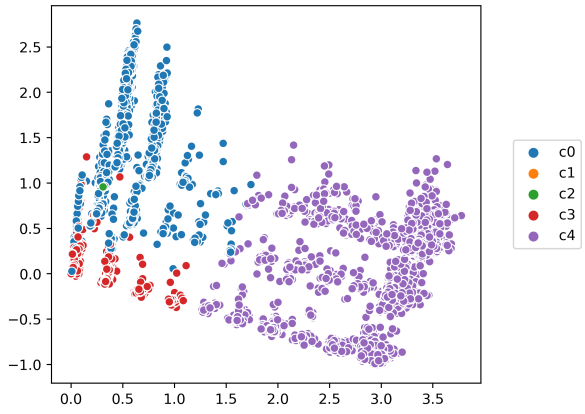

Fig. 2: Visualisation of clustering for k=5

Cluster c0 contains contracts which have functionality related to ownership: 5831/6330 contain `owner()` and 3444/6330 contain `transferOwnership(address)`. These two functions comprise the required functions defined in ERC173, suggesting that we can identify the the usage of this ERC. It appears that a great many contracts implement half of ERC173, by defining only the owner function. In c4; after the ERC20 functions, the most prevalent selectors are `owner()` and `transferOwnership(address)` which occur in 58.3% and 42.3% of the contracts respectively.

c1 contains one unique contract with a weight of 363285. This makes it a very commonly occurring contract. It contains all the ERC20 required and optional selectors. A number of its functions are not found in the 4bytes database. Removing all ERC20 and unknown selectors, we find `grant(address,uint256)`, `owner()` and `transferOwnership(address)`. The presence of ERC173 in such a heavily weighted cluster shows it is in frequent use. The presence of `grant(address,uint256)` shows a very commonly occurring function not present in any ERC. ERC1207, has a grant function, however it takes an extra string as a parameter.

c2 provides our first candidate for an ERC. It contains 16 unique contracts, and represents 56 contracts in the raw data. All the contracts contain:
`token()`, `bought_tokens()`, `sale()`, `set_token_address()`
All but one contain:
`set_sale_address(address)`, `change_min_amount(uint256)`, `set_percent_reduction(uint256)`, `change_max_amount(uint256)`, `change_owner(address)`.
ERC900 has `token()` as one of its required functions, but none of its other functions are present. This cluster is a candidate for a new ERC. The function names suggest contracts used for crowd sales. As they do not contain any of the ERC20 selectors it is likely there is unstandardised, but desired, functionality. We suggest a crowd sale ERC with required functions defined as those listed above.

c3 contains 8451 unique contracts. The most commonly occurring function is `kill()`, which occurs in only 1191 of the contracts. This cluster seems to catch contracts that do not display a strong similarity any others.

C. k=9

lastly, we look at the results for k=9, the clusters for which can be seen in fig 3. At a glance, boundaries between the major clusters have not vastly changed. However, a detailed look at the contents of the clusters allows us to draw out more interesting information.

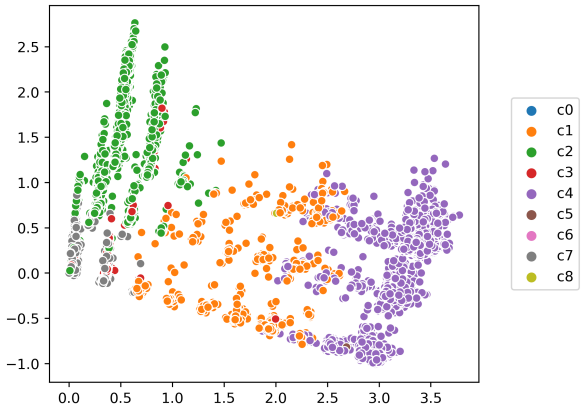

Fig. 3: Visualisation of clustering for k=9

As before, c4 identifies the ERC20 contracts; 92.5% of the cluster is made up of contracts implementing all the required ERC20 functions. The hybrid implementations of ERC20 have been separated out into c1, with full ERC implementations making up only 5.3% of the cluster. This shows that we are able to distinguish between full ERC20 implementations and partial ones. In c1 optional functions have a greater presence than some of the required functions, with `symbol()` appearing more frequently than some required functions. This shows the same trend for hybrid ERC20 implementations.

c6 identifies the same heavily weighted, unique ERC20 contract as c1 for k=5. The c7 cluster is extremely similar to c3 for k=5. The number of contracts is almost identical and they display the same level of heterogeneity. As such, this cluster is still catching contracts which do not fit elsewhere.

c2, like c0 for k=5, identifies contracts whose shared attributes are the functions defined in ERC173, with 95.9% containing the owner function. The trend for partial implementations of ERC173 is also visible here with only 55% of the contracts having implemented the required `transferOwnership(address)` function.

c0 contains 7 unique contracts, with a combined weight of 140. All the contracts contain: `signers`, `(uint256)`, `activateSafeMode()`, `(isSigner(address)`, `(safeMode()` All but one contain:
`createForwarder()`, `getNextSequenceId()`, `sendMultiSig(address,uint256, bytes,uint256,uint256,bytes)` None of these functions appear in any of the current ERCs. The names suggest an agreement schema. We put forward our second ERC recommendation based on the above functions.

c3 has the following functions in more than 70% of its contracts: `amountRaised()`, `0x6e66f6e9`, `beneficiary()`, `balanceOf(address)`. With the exception of the selector for which no name is available, these functions suggest contracts for fund raising. Given the number of functions pertaining to funding, we posit that a candidate for an ERC is a template for the kind of fund raising suggested by the functions in this cluster. An extension of ERC20 could include these functions to allow users to check how much has been raised and to whom ether or tokens are being given.

c5 consists of 45 unique contracts with a combined weight of 124360. All the contracts have:
`isOwner(address)`, `revoke(bytes32)`, `hasConfirmed(bytes32,address)`
All but one have:
`addOwner(address)`, `m_numOwners()`, `changeOwner(address,address)`, `changeRequirement(uint256)`, `removeOwner(address)`, `m_required()` These functions very strongly suggest smart contracts used to record and update ownership. Given the homogeneity and weight of this cluster, we suggest an ERC to provide a generic way to record ownership for tokens, tangible and intangible assets, or other contracts. Optional functions could be defined for different types of asset.

Lastly, c8 contains 2 unique contracts with a combined weight of 5. Although few in number, it is interesting that these contracts were given their own cluster. On inspection, the functions describe a game.

VI. Conclusion

In this paper we detail the results of clustering Ethereum smart contracts when each contract is treated as the set of the function selectors found in its bytecode. Clusters are labelled according to the results of a reverse look-up, from function selectors to function names. We cluster using K-means with different values of k. From k=2 onwards the major clusters begin to emerge. We maintain accuracy for increased values of k: k=5 and k=9. New clusters which emerge when increasing the value of k allow us to highlight contract purposes for which no ERC currently exists, and make ERC recommendations. We validate our method by showing that ERC20 contracts can be accurately clustered.

Future work includes taking contract transaction volume into account and investigating the effectiveness of using a custom distance metric for clustering.

References

[1] Ethereum, "Ethereum (ETH) Blockchain Explorer," https://etherscan.io/, 2018, [Accessed 8th November 2018].

[2] F. Vogelsteller and V. Buterin, "ERC-20 Token Standard," https://eips.ethereum.org/EIPS/eip-20, 2015, [Accessed 8th November 2018].

[3] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[4] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 442–446.

[5] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, 2018, pp. 81–84.

[6] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 116, 2018.

[7] M. Marescotti, M. Blicha, A. E. Hyvärinen, S. Asadi, and N. Sharygina, "Computing exact worst-case gas consumption for smart contracts," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018, pp. 450–465.

[8] melonproject, "Oyente," 2018, [Accessed 8th November 2018].

[9] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," *arXiv preprint arXiv:1806.01143*, 2018.

[10] ConsenSys, "Mithril Classic," https://github.com/ConsenSys/mythril-classic, 2018, [Accessed 8th November 2018].

[11] usyd blockchain, "Vandal," https://github.com/usyd-blockchain/vandal, 2018, [Accessed 8th November 2018].

[12] 4byte, "4byte Directory," https://www.4byte.directory, 2018, [Accessed 11th December 2018].

[13] ethereum lists, "List of 4byte identifiers to common smart contract functions," https://github.com/ethereum-lists/4bytes, 2018, [Accessed 13th December 2018].

[14] R. Norvill, B. B. F. Pontiveros, R. State, I. Awan, and A. J. Cullen, "Automated labeling of unknown contracts in ethereum," pp. 1–6, 2017.

[15] etherum, "ERC — Ethereum Improvement Proposals," https://eips.ethereum.org/erc, 2018, [Accessed 14th November 2018].

[16] scikit-learn developers, "sklearn.cluster.kmeans," https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html, 2018, [Accessed 16th December 2018].