Mining DApp Repositories: Towards In-Depth Comprehension and Accurate Classification

Yeming Lin*†, Jianbo Gao‡, Tong Li*†, Jingguo Ge *†, Bingzhen Wu*†
*Institute of Information Engineering, Chinese Academy of Science, Beijing, 100093, China
†School of Cyber Security, University of Chinese Academy of Sciences, Beijing, 100049, China
‡School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China
*†{linyeming,litong,gejingguo,wubingzhen}@iie.ac.cn, ‡gaojianbo@pku.edu.cn

***Abstract*—Blockchain has recently attracted great interest from both academia and industry. Ethereum introduces programmability into blockchain through smart contracts and provides an open-source computing platform for blockchain-based decentralized applications (DApps). There are currently thousands of DApps pertaining to different application domains, including games, gambling and finance. In order to better comprehend blockchain application scenarios and help developers understand DApps better, clear DApps classification criteria are needed. However, many DApps that can be found through collection websites (commonly known as DApp Stores) are not classified properly, making these datasets imprecise. This issue has motivated the present empirical study of DApp categories, as a part of which over 2,500 DApps in three DApp Stores are investigated, allowing us to produce and publicly release a high-quality dataset in which misclassified DApps are relabeled manually, facilitating their more precise classification. We also propose DAppClassifier, a novel technique for classifying DApps based on their actual functionalities. When developing the new classifier, we extracted features from source code, bytecode and historical transactions, and trained neural networks to classify DApps. Our approach was evaluated on the released dataset and achieved good precision.**

***Index Terms*—Blockchain, Decentralized Application, Comprehension, Ethereum, Smart Contract**

I. Introduction

Owing to the growing interest in Bitcoin, considerable efforts have been invested into the development of blockchain techniques. Ethereum is a public open-source blockchain-based distributed computing platform, providing a Turing-complete virtual machine for executing smart contracts. Smart contract refers to open-source programs that can be automatically executed without any centralized control. Consequently, it is the most important innovation of Ethereum.

Blockchain-base decentralized applications (abbreviated as DApps), are the emerging trend in the blockchain development, as they rely on smart contracts instead of traditional centralized server as the back-end. Owing to its transparency, decentralization, and security, this infrastructure offers immense potential for use in a variety of fields, including finance, governance, supply chain management, etc. Thus, it is not surprising that the number of DApps has already surpassed 2,500 within 4 years, with the value of the DApp market estimated at billions of dollars [1].

DOI reference number: 10.18293/SEKE2020-088

DApp Store is a repository that facilitates DApp management. Akin to AppStore for the iOS system, it provides a convenient platform for users to browse DApps and select those that meet their needs. To expedite searching, DApp Stores classify DApps into a set of categories, such as Games, Gambling, Exchange, and High-risk.

Such categorization can be helpful for both users and developers, as users can browse the appropriate category to find relevant DApps, whereas developers can determine the most optimal category for their DApps prior to submission.

At present, DApp categories are manually selected by the developers at the time of submission. The maintainers of DApp Stores will subsequently manually check if the classification is correct, aiming to detect high-risk DApps (such as those hiding a Ponzi scheme). However, as such classification relies on human judgment, it can be error-prone. On one hand, the predefined categories may be ambiguous, and the developers may find it difficult to locate a proper category for their DApp. On the other hand, as manually verifying classification for each DApp is time-consuming, there is a risk of exposing DApp users to security issues.

To provide a comprehensive understanding of the DApp classification status, as a part of this investigation, an empirical study involving over 2500 DApps in three DApp Stores is conducted. Based on the findings obtained, a dataset is constructed and its accuracy is optimized by manually relabeling misclassified DApps.

However, to the best of our knowledge, no effort has been devoted to the DApp classification problem. Although several approaches have been proposed for labelling and identifying smart contracts (a component of DApp), they can only be utilized to cluster similar smart contracts [2] or identify special smart contract types, such as Ponzi scheme or honeypot [3] [4]. Thus, none of them can be directly applied to classify multifarious DApps. Even though, some approaches to mobile App classification can be potentially modified for use in DApp classification, as they do not take advantage of the unique characteristics of DApps, the classification performance would be compromised.

To overcome these shortcomings, in this paper, we propose DAppClassifier, a novel technique for classifying DApps based

on easily obtainable rich and comprehensive features that represent the actual DApp functionalities. DAppClassifier first extracts features from source code, bytecode and historical transactions, which is the communication between the user and the DApp. Next, hybrid neural networks are devised to classify DApps.

To evaluate DAppClassifier performance, it is applied to the optimized dataset (obtained in the first phase of this study as a part of which misclassified DApps were relabeled manually), achieving >84% average precision.

The main contributions of this work are as follows:

- **We conduct a systematic empirical study on DApp status,** extensively investigate DApp classification problem and obtain the different categories characteristics.
- **We construct a high-quality dataset** by relabeling misclassified DApps. The revised dataset is open sourced and can be accessed from https://bit.ly/2JFmtiS.
- To eliminate the need for manual verification and reclassification, **we propose DAppClassifier that can automatically classify DApps based on their rich and comprehensive features.** The effectiveness of DAppClassifier is subsequently confirmed by testing its performance against other available techniques.

II. Background

In this section, we provide the background information required for understanding our work.

A. Ethereum and Smart Contract

Ethereum is a public open-source blockchain-based distributed computing platform that provides a running environment for decentralized applications.

A smart contract is a computer program outlining the rules under which the participants agree to interact with each other. If the pre-defined rules are met, the agreement is automatically enforced.

B. Ethereum Virtual Machine (EVM)

Ethereum provides EVM to support the compilation and execution of smart contracts. Technically, it is the runtime environment for smart contracts in Ethereum. It is a stack-based, register-less virtual machine, where operators and operands are all pushed onto the stack indistinguishably, with the exception of the data that requires persistent storage space on Ethereum. Solc compiler will translate readable solidity code into bytecode, only EVM can understand.

C. Decentralized Applications

Centralized systems directly control the operation of the individual units and flow of information from a single center. Decentralized applications (DApps) are applications that run on a decentralized network rather than a single computer. Technically, a DApp is composed of front-end and back-end code, whereby the front-end is an Internet-based interface, typically a web page, and the back-end contains the key data and operations, typically based on one or more smart contracts in a blockchain. When users interact with blockchain-based DApps, a transaction is initiated and recorded on the blockchain permanently. So we can get all the interactions between DApps and users from the blockchain.

III. Empirical Study

In this section, current DApp classification status is briefly outlined.

For this analysis, three DApp Stores: State of the DApps [1], DappRadar [5], and Dapp.com [6] are chosen as the representatives of the current DApp market (based on the Google Search results).

A. Overview of DApp Categories

This section commences with an overview of DApp categories, to help readers better understand the current classification criteria. DApp Stores utilize predefined categories, allowing users to find the DApps they need. However, these are not standardized, making comparisons across different platforms difficult, while introducing the risk of misclassification.

TABLE I
DAPP CATEGORIES ON THE WEBSITES. FOR EACH CATEGORY, THE COLUMNS OF THE TABLE SHOW, FROM LEFT TO RIGHT: THE NUMBER OF DAPPS FROM *State of the DApps* (**#S**), *DappRadar* (**#R**), AND *Dapp.com* (**#D**).

Category	#S	#R	#D
Games	303	402	455
Gambling	211	375	266
High-risk	148	333	256
Exchanges	106	57	65
Finance	101	-	53
Social	86	-	45
Media	55	-	-
Development	49	-	-
Marketplaces	41	15	-
Property	29	-	-
Governance	27	-	-
Wallet	20	-	-
Security	20	-	-
Storage	17	-	-
Identity	15	-	-
Health	4	-	-
Insurance	4	-	-
Energy	3	-	-
Collectibles	-	52	-
Tools	-	-	57
Art	-	-	33
Others	-	260	84
Total	1239	1494	1314

B. Misclassification of DApps

In this section, we provide a descriptive analysis on the DApps in the three most widely used platforms: State of the DApps [1], DappRadar [5], and Dapp.com [6] to determine if these are properly classified by the developers. We crawled all DApps information from three platforms and matched them by front-end URL and contract address. If they have the same front-end URL or similar front-end URL with the same contract address, we consider them to be the same DApp, for some DApps have different URL parameters in order to identify the source of the request. Our investigation has uncovered

TABLE II
MISCLASSIFICATION ACROSS DIFFERENT DAPP STORES

Overlap Categories	#DApps
Games, High-risk, Gambling	8
Gambling, High-risk	77
Games, High-risk	71
Games, Gambling	37
Games, Collectibles	25
Games, Marketplaces	13
High-risk, Finance	12
Finance, Others	20
Social, Others	24
Tools, Others	12

two important problems with the current classification system, which are discussed below.

1) The classification criteria are ambiguous and differ across DApp Stores: We compare DApps and the categories utilized by the aforementioned three DApp Stores, and find that the classification criteria are non-uniform and ambiguous, for the following two reasons.

First, each DApp Store uses a different number of categories, at 18, 7, and 9, respectively (Table I). While Games, Gambling, High-risk and Exchanges are utilized in all three cases, *Health* and *Insurance*, for example, only exist in *State of the DApps*.

Second, even if DApp Stores utilize the same category, such as Games or Gambling, the classification criteria can be ambiguous. In an ideal scenario, the same DApp should be classified into the same category on all DApp Stores. However, according to our investigation, this is not always the case, as shown in Table II. For example, 37 DApps classified as Games on one DApp Store are classified as Gambling on the other.

Moreover, as no clear description of the categories is provided on the DApp Stores, developers would find it challenging to determine the most appropriate category for their DApps.

2) DApps are often misclassified by DApp Stores: As a part of our investigation, we examine the characteristics of DApps and check if these corresponded to their classification by DApp Stores.

For this purpose, we conducte a preliminary experiment on smart contracts of DApps, which is guided by three hypotheses: (1) Intuitively, DApps with the same smart contracts should be in the same category; (2) DApps with the same runtime code should be in the same category; and (3) DApps with the same opcode list should be in the same category, because opcode (rather than the operand) determines the program execution logic.

To test these hypotheses, we extract smart contracts from DApps. When comparing run-time code, we match the code textually, whereas for opcode list comparison, we split the runtime code into opcode list and operand list, and remove the latter.

Our findings revealed that, in the three DApp Stores included in our analysis, 41, 29, and 36 DApps, respectively, violate the three hypotheses given above, suggesting high degree of misclassification. We manually analyze these DApps and found that more than 40% of DApps changed from High-risk to other categories, in order to trick users into using. We manually corrected these classification errors in our dataset.

C. The Difficulties of DApp Classification

As mentioned earlier, due to the lack of clarity in the classification criteria used by DApp Stores, developers often struggle with interpreting the requirements for each category, which may result in misclassification of their DApps.

To mitigate these issues, an automatic classification approach is needed, as it would allow developers to identify the most suitable category for their DApps, while aiding the DApp Store maintainers in the verification process, which is currently conducted manually.

However, objective DApp classification is challenging for several reasons, as explained below.

1) Limited EVM features: EVM is a runtime environment for smart contracts based on a 256-bit register stack. Unlike in traditional operating systems, in EVM, program's operators and operands are all pushed onto the stack indistinguishably. As EVM relies on fewer operator types, it is not as complex as traditional operating systems. For example, Dalvik Instruction Set Architecture (ISA) defines 218 Android application instructions, and Java runtime machine (JVM) provides about 202 instructions, whereas only 137 are provided by the EVM.

Traditional program classification techniques are mostly based on static and dynamic features of the code. Due to the simplicity of smart contracts, it is much harder to generalize features to classify DApps.

2) Redundancies in libraries: To reduce human effort and avoid source code duplication, DApp developers tend to invoke public libraries developed by some third-party organizations (like openzeppelin[1] or oraclize[2]). Most developers and companies also store proprietary code in private libraries for subsequent reuse. In the context of this investigation, a library can be regarded as a special kind of smart contract. A typical smart contract is composed of multiple functions with different access scope, such as Internal, External, Public, Private, etc. However, in contrast to traditional programming languages, when compiling a smart contract that relies on libraries, all public and external functions will be compiled into application binary interfaces (abbreviated as ABIs), which serve as the inference for library invocations, regardless of whether the function is needed by the DApp.

We analyze library invocation by all DApps from the three DApp Stores, whereby Table III provides all libraries that are invoked by more than one category. As can be seen, most public or external functions are never used. Due to such redundancy, many conventional program classification techniques (which are often based on program similarity) cannot be applied effectively.

Moreover, if a library is invoked by multiple DApps from different categories, these DApps will have similar sets of

[1]https://openzeppelin.org/
[2]http://provable.xyz/

TABLE III
MOST COMMONLY INVOKED LIBRARIES.

Library Name	#DC	#SC	#Int	#Ext	#Used
Math	11	47	12	4	0
ECRecovery	8	20	2	1	0
MathLib	3	8	4	3	0
PaymentLib	2	6	6	21	0
CommUtils	2	6	19	8	0
DLL	2	6	0	8	0
Player	2	6	9	8	1
AttributeStore	2	6	0	2	0
StringLib	2	6	1	1	0
SortitionSumTreeFactory	2	4	1	8	0
Helper	2	4	4	17	0
LinkedListLib	2	2	11	6	0

For each library, the columns of the table show, (from left to right): the number of DApp categories (#DC) and smart contracts (#SC) invoking the library, internal and private functions (#Int), external and public functions (#Ext), and public and external functions (#Used) in the library that are actually used.

ABIs. As a result, ABI-based classification techniques which are commonly used in Mobile App classification cannot be directly applied to classify DApps.

As illustrated above, current DApp classification is time-consuming and error-prone, as it is challenging to utilize the limited information from the DApp itself for this purpose. To address this issue, we develop an automated classification method, DAppClassifier, as described in the subsequent section.

IV. DAPPCLASSIFIER

A. An Overview of DAppClassifier

In this section, we use machine learning to solve the DApp classification problem. The resulting DAppClassifier predicts DApp category based on easy-to-access features representing the actual DApp functionalities. The overall DAppClassifier structure is shown in Figure 1. Given the input of all information pertaining to a particular DApp—including DApp bytecode, transactions in history, and DApp source code (optional)—the classification is performed in two steps: the feature extraction process and the classification process. In the following sections, we provide detailed description of these two processes.

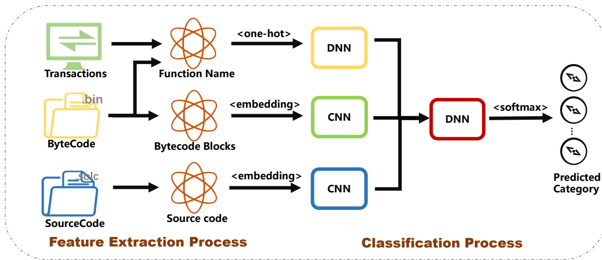

Fig. 1. An Overview of DAppClassifier

B. Feature Extraction Process

In this step, various features that are related to the DApp category are identified, ensuring that they are easy-to-access and can reflect the program functionalities.

The feature extraction process is guided by three hypotheses: (1) Function names reflect their actual functionalities; (2) Similar execution logic is more likely to imply similar program functionalities; and (3) The developer-defined names in source code are indicative of the actual functionalities.

1) Function name features: In practice, function names should reflect program functionalities. However, these can be difficult to obtain. Intuitively, function names can be deduced from function calls incorporated into the execution traces during transactions, but this is unreliable, as not all functions will be called during execution, especially for infamous DApps. As an alternative, features can be extracted from binary code and transactions. However, this approach leads to two difficulties discussed below.

First, only the 4-byte digital signatures (hash values of function names) can be extracted from binary code. These signatures cannot be directly used as features, because one of the classification criteria mandates that similar function names should imply similar functionalities. Thus, it is necessary to convert each signature into the initial word-form. Second, as explained in Section III, a significant number of unused library functions are compiled into the bytecode, which would compromise the effectiveness of the selected features.

Our feature extraction approach can mitigate these issues. We retrieve word-form function names by utilizing Ethereum Function Signature Database[3] —a large-scale open-source database that records common function names and corresponding signatures. This allows those 4-byte signatures to be mapped to their human-readable versions (in our experiment, about 60% function names can be matched). In addition, we perform an additional filtering process, whereby we identify all library functions before collecting all the function names invoked through transaction. Finally, we remove all the uninvoked public library function names to eliminate redundancies.

2) Bytecode block features: The bytecode reflects DApp behavior, and can thus be used in classification. However, the bytecode cannot be used directly due to several reasons.

First, bytecode is composed of three components: creation code, swarm code, and runtime code. As the first two elements are used for the creation and distributed storage, they have no contribution to the program runtime behavior. Second, runtime code consists of operators and operands (i.e., PUSH 0X80). If the operands are not filtered out, the extracted feature will be too sensitive to the operands. Third, runtime code comprises of multiple blocks, each of which indicates a logical unit. Thus, to maximize the feature utility, runtime code should be divided into basic blocks.

To alleviate these issues, DAppClassifier extracts refined and subtle features from bytecode in three steps: [2]

1) Redundant code elimination: In this step, all superfluous creation and swarm code parts, which easily identified, are removed.
2) Desensitization: All the operands (i.e., the immediate numbers after operators) are removed. Note that, as a special kind of operand, the signature of function name

[3]https://www.4byte.directory/

is also removed. Those have already been discussed in function name features.

3) Division into basic blocks: In runtime code, operators JUMP, JUMPI, REVERT, STOP, and RETURN are the indicators of an interruption in a logical relationship. DAppClassifier divides the runtime code according to the opcodes.

Following the above process, DAppClassifier can identify refined and subtle block-level sub-sequences of bytecode as a feature.

3) Source code features: Compared with bytecode, source code can better convey the intent of developers. Not only the function names and statements are in high-level language, which is similar to the human language, but code also contains abundant programming notes, which convey intention of developers.

For this reason, DAppClassifier applies tokenization and embeddings features from source code, which is a two-layer neural network that processes text, whereby its input is a text corpus and its output is a set of vectors.

C. Classification Process

In the classification model, (1) three units are used to handle the three kinds of features, which are subsequently combined by (2) applying another Deep Neural Network (DNN) model.

1) DNN for Feature Name features: DNN model is a learning method with multiple layers of neural networks. It is particularly suitable for classification prediction problems where inputs are assigned a class or label.

Specifically, our network is a fully connected DNN model with RELU activations, N layers, and M units per layer. We swept over N = [3, 4, 5, 6, 7] and M = [32, 64, 128, 256, 512]. Our best performing model has N = 3 layers and M = 64 units, yielding a learning rate of 3×10^{-5}.

2) CNN for Function Body features: Convolutional Neural Network model (abbreviated as CNN) is a neural network that uses convolution in place of general matrix multiplication.

Specifically, our network is a CNN model with an embedding layer (embedding dim = 128), a convolutional layer (256 filters with kernel_size = 5) and a MaxPool layer (each layer contains 64 neurons).

3) CNN for Source Code features: In line with the approach adopted for sentence classification in the *NLP* scenario, to handle features, CNN with the same specifications as given above is utilized to extract meaningful sub-structures from source code.

4) DNN to Conjunct the Units: The intermediate results of the above three models are concatenated by a model composed of three layers: an input layer combining the output of the three units, a 128-dimensional fully connected layer, and a *softmax* layer to output the final category.

V. Evaluation

In this section, we describe the experimental design adopted for evaluating DAppClassifier, followed by the experimental results demonstrating its correctness and effectiveness.

TABLE IV
EXPERIMENTAL RESULTS

Dataset	DAppClassifier	FuncName	ByteBlocks	Source
#S	84.6%	78.2%	77.2%	79.9%
#R	83.5%	78.9%	76.3%	79.3%
#D	84.0%	80.1%	79.6%	70.3%

A. Evaluation Design

We conduct large-scale experiments on DApp classification based on our self-constructed open-sourced dataset.

Dataset: Data on the three most commonly-used DApp Stores, namely State of the DApps, DappRadar, and Dapp.com are crawled, manually checked and all misclassified DApps are relabeled or removed(As explained in Section III-B),we matched them by front-end URL and contract address. If they have the same front-end URL or similar front-end URL with the same contract address, we consider them to be the same DApp, for some DApps have different URL parameters in order to identify the source of the request. **Yielding a large-scale dataset covering about 2,573 DApps including 11,230 smart contracts deployed on the Ethereum Environment.**

For each DApp, we download the bytecode and transactions, and retrieve its source code (if available) by Etherscan. To collect data on these smart contracts, we manually run an Ethereum node, starting from the Genesis block to the latest block to identify all the transactions and extract bytecode and runtime code. We focus on the DApp categories containing at least 20 DApps, since these categories are more representative of the general classification status. The revised dataset has been released (https://bit.ly/2JFmtiS), it contains the DApp name, contract address, category, DApp url, etc..

B. Experimental Results

In this section, we present the experimental results related to the two research questions.

RQ1: How many DApps can be accurately classified into the correct category? The experimental results of our classifier reported in Table IV indicate that its accuracy surpasses 84% when applied to each of the DApp Stores, confirming that our approach is technically promising. As precision, recall and F-score are consistent for multiclassification problem, only the precision of our approach is reported.

RQ2: How does each feature contribute to the DApp classification performance?

The columns 3–5 of Table IV show the performance (measured in terms of precision) of DAppClassifier based on a single feature (with a single classification unit).

Features extracted from source code should aid in accurate classification, as the programmer's intent is typically conveyed through function and routine names and developer comments. However, the results reported in the last column of Table IV counter this intuitive expectation, requiring further investigation.

Note that the experiments focusing on source code only are conducted on DApps for which source code is available. As nearly half of the DApps lacked source code, the dataset

used for training in this experiment is substantially reduced. In particular, in *Dapp.com* only 50.4% of DApps are released with source code, compared with 63.9% and 56.7% DApps on *State of the DApps* and *DappRadar* respectively. Thus, all three features should be utilized to improve the classification accuracy.

VI. Discussions

In this section, we will discuss the benefits and shortcomings of our approach.

Function name feature extraction benefits: In the proposed approach, prior to extracting function name features, based on the unique DApp characteristics, all the un-invoked public library function names are removed. To evaluate the effectiveness of this strategy, we conduct another experiment on the DApps with historical transactions, applying only the function name feature unit. Besides the feature extraction process described in Section IV, we conduct comparison experiments, each incorporating the following feature extraction processes:

- *Names from Invocation & Bytecode removed unused libraries*: The features we used in our approach.
- *All names from Bytecode*: All function names extracted from bytecode are utilized, i.e., the unused public library functions are not removed.
- *All names from Invocation*: All function names from invocations in the history of transactions are identified and utilized.
- *Names from Invocation & Bytecode*:The function names collected by adopting the strategy described in Section IV are combined with those from the invocations.

As can be seen in Figure. 2, the features collected by utilizing *All names from Invocation* process achieve the lowest accuracy. This finding supports our hypothesis that function names identified through invocations of historical transactions are not sufficient for meaningful classification, due to users inadequate. The features collected via the strategy denoted as *All names from Bytecode* yield a higher precision because all the functions in the application are utilized, not just those that have been called. However, lack of understanding of user behavior leads to insufficient accuracy.

We take the advantage of both and combine them as input. The features collected through *Names from Invocation & Bytecode* and *Names from Invocation & Bytecode removed unused libraries* can both reach the top accuracy given a long training time. However, by removing all the unused library methods, the learning convergence can be expedited.

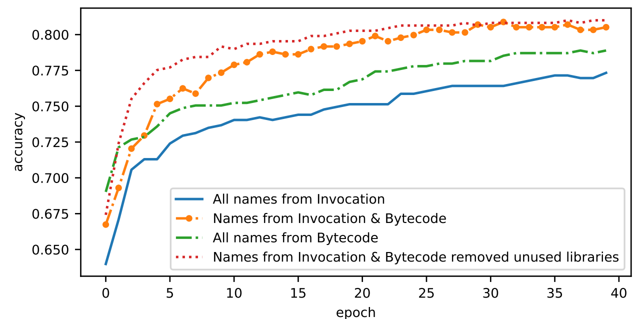

Fig. 2. Comparison within different features

VII. Related Work

Several studies have been conducted to evaluate the Ethereum ecosystem. For example, some authors characterized money transfer [7], contract invocation [8], code similarity [9] of Ethereum. Other researchers focused on financial activities on Ethereum, including Ponzi schemes [3], Honeypots [4] detect and ICO behavior finding [10], which might be a complement to our work. There is an empirical study on distributed applications [11]; however, our research is more systematic and is conducted on a larger scale. Moreover, machine learning has been used to label similar smart contracts with source code [2], even though its notion of a "cluster" cannot be precisely defined.

VIII. Conclusion and Future Work

Owing to the development of blockchain and mobile technology, the number of DApps has already surpassed 2,500, and the scale of DApp market is estimated at billions of dollars [1]. To provide a better comprehension of decentralized applications (DApps), we have conducted a systematic empirical study on DApp classification status, and have constructed a dataset by relabeling misclassified DApps. Based on our empirical findings, we have proposed DAppClassifier—a novel approach for classifying DApps based on their real functionalities. Extensive evaluations have demonstrated that DAppClassifier can achieve an average precision of greater than 84%.

References

[1] S. of the DApps, "Stateofthedapps," 2020. [Online]. Available: https://www.stateofthedapps.com
[2] R. Norvill, B. B. F. Pontiveros, R. State, I. Awan, and A. Cullen, "Automated labeling of unknown contracts in ethereum," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2017, pp. 1–6.
[3] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, and Y. Zhou, "Detecting ponzi schemes on ethereum: Towards healthier blockchain technology," in *Proceedings of the 2018 World Wide Web Conference*. International World Wide Web Conferences Steering Committee, 2018, pp. 1409–1418.
[4] C. F. Torres and M. Steichen, "The art of the scam: Demystifying honeypots in ethereum smart contracts," *arXiv preprint arXiv:1902.06976*, 2019.
[5] D. Radar, "Dappradar," 2020. [Online]. Available: https://dappradar.com
[6] DApp.com, "Dappcom," 2020. [Online]. Available: https://www.dapp.com
[7] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange, "Understanding ethereum via graph analysis," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1484–1492.
[8] L. Kiffer, D. Levin, and A. Mislove, "Analyzing ethereum's contract topology," in *Proceedings of the Internet Measurement Conference 2018*. ACM, 2018, pp. 494–499.
[9] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the ethereum smart contract ecosystem," *arXiv preprint arXiv:1905.00272*, 2019.
[10] G. Fenu, L. Marchesi, M. Marchesi, and R. Tonelli, "The ico phenomenon and its relationships with ethereum smart contract environment," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 26–32.
[11] K. Wu, "An empirical study of blockchain-based decentralized applications," *arXiv preprint arXiv:1902.04969*, 2019.