# SmartEmbed: A Tool for Clone and Bug Detection in Smart Contracts through Structural Code Embedding

Zhipeng Gao
*Monash University*
Melbourne, Australia
zhipeng.gao@monash.edu

Vinoj Jayasundara
*Singapore Management University*
Singapore
vinojmh@smu.edu.sg

Lingxiao Jiang
*Singapore Management University*
Singapore
lxjiang@smu.edu.sg

Xin Xia
*Monash University*
Melbourne, Australia
Xin.Xia@monash.edu

David Lo
*Singapore Management University*
Singapore
davidlo@smu.edu.sg

John Grundy
*Monash University*
Melbourne, Australia
John.Grundy@monash.edu

***Abstract*—Ethereum has become a widely used platform to enable secure, Blockchain-based financial and business transactions. However, a major concern in Ethereum is the security of its smart contracts. Many identified bugs and vulnerabilities in smart contracts not only present challenges to maintenance of blockchain, but also lead to serious financial loses. There is a significant need to better assist developers in checking smart contracts and ensuring their reliability. In this paper, we propose a web service tool, named SMARTEMBED, which can help Solidity developers to find repetitive contract code and clone-related bugs in smart contracts. Our tool is based on code embeddings and similarity checking techniques. By comparing the similarities among the code embedding vectors for existing solidity code in the Ethereum blockchain and known bugs, we are able to efficiently identify code clones and clone-related bugs for any solidity code given by users, which can help to improve the users' confidence in the reliability of their code. In addition to the uses by individual developers, SMARTEMBED can also be applied to studies of smart contracts in a large scale. When applied to more than 22K solidity contracts collected from the Ethereum blockchain, we found that the clone ratio of solidity code is close to 90%, much higher than traditional software, and 194 clone-related bugs can be identified efficiently and accurately based on our small bug database with a precision of 96%. SMARTEMBED can be accessed at http://www.smartembed.net. A demo video of SMARTEMBED is at https://youtu.be/o9ylyOpYFq8**

## I. INTRODUCTION

In recent years, with the adoption and development of cryptocurrencies on distributed ledgers (a.k.a., blockchains), Ethereum has attracted increasingly attention as a blockchain platform. At the heart of the Ethereum platform are *smart contracts*. A Smart contract is a computer program that can be triggered to execute any task when specifically predefined conditions are satisfied. A major concern in the Ethereum platform is the security of smart contracts. A smart contract in the blockchain often involves cryptocurrencies worthy of millions of dollars (e.g., DAO[1], Parity[2] and many more). Moreover, different from a traditional software program, the smart contract code is immutable after its deployment. They cannot be changed but may be killed when any security issue is identified within the smart contracts. This introduces challenges to blockchain maintenance and gives much incentive to hackers for discovering and exploiting potential problems in smart contracts, hence there is a very significant need to check and ensure the robustness of smart contracts before deployment.

Many prior works have investigated bug detection of smart contracts (e.g., [1]–[3]). A major disadvantage is that all these existing tools require certain bug patterns or specification rules defined by human experts. Considering the high stakes in smart contracts and race between attackers and defenders, it can be far too slow and costly to write new rules and construct new checkers in response to new bugs and exploits created by attackers. Recently, there are also studies on clones and clone detection for Ethereum smart contracts (e.g., [4], [5]). However, they use expensive symbolic transaction sketch or pair-wise comparisons which affect their efficiency and they are limited to clone detection. Machine learning and deep learning techniques have been used for clone detection and bug detection problems (e.g. [6], [7]) in traditional software programs too, but little has been applied for smart contracts.

In this paper, we present SMARTEMBED, a web service tool which can be accessible at http://www.smartembed.net. SMARTEMBED can efficiently and effectively check smart contracts for clones and bugs, and can evolve bug checking rules easily along with additions of new bugs. The main idea of SMARTEMBED is two folds. (1) Code Embedding: utilizing basic program analyses and the availability of many open-source smart contracts, we encode each code element and bug pattern automatically, including their lexical, syntactical, and even some semantic information, into numerical vectors via techniques adapted from word embeddings (e.g., [8]). (2) Similarity Checking: utilizing efficient similarity comparisons among the numerical vectors representing various kinds of code elements at various levels of granularity in smart contracts, we can detect clones similar to each other and bugs similar to known ones.

[1]https://en.wikipedia.org/wiki/TheDAO(organization)
[2]https://paritytech.io/security-alert-2/

2576-3148/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSME.2019.00067

SMARTEMBED is unique in that it utilizes deep learning and similarity checking techniques to unify clone detection and bug detection together efficiently and accurately for Ethereum smart contracts. When applied to more than 22K solidity contracts curated from the Ethereum blockchain, SMARTEMBED effectively tells us that the clone ratio of the Solidity code is at around 90% and 194 out of 202 reported clone-related bugs are true bugs.

The rest of the paper is organized as follows. Section II presents overall workflow of our approach and details of each step. Section III introduces the implementation details of our tool and its usages. Section IV shows the experimental results of our evaluation. Section V concludes our work.

## II. APPROACH

### A. Overview

Fig.1 illustrates the overall framework of SMARTEMBED. Based on code embeddings and similarity checking, SMARTEMBED targets two tasks in a unified approach: clone detection and bug detection. For clone detection, SMARTEMBED can identify similar smart contracts. For bug detection, based on our bug database, SMARTEMBED can detect bugs in the existing contracts in the Ethereum blockchain and/or in any smart contract given by solidity developers that are similar to any known bug in the database. Our approach contains two phases: a model training phase and a prediction phase.

There are mainly 4 steps during the model training phase. We built a custom Solidity parser for smart contract source code. The parser generates an abstract syntax tree (ASTs) for each smart contract in our collected dataset, and serializes the parse tree into a stream of tokens depending on the types of the tree nodes (step 1). After that, the normalizer reassembles the token streams to eliminate nonessential differences (e.g., the stop word, values of constants or literals) between smart contracts (step 2). The output token streams are then fed into our code embedding learning module, and each code fragment is embedded into a fixed-dimension numerical vector (step 3). After the code embedding learning step, all the source code is encoded into the source code embedding matrix; in the meanwhile, all the bug statements we collected are encoded into the bug embedding matrix (step 4).

In the prediction phase, any given new smart contract is turned into embedding vectors by going through the steps 1,2,3 and utilizing the learned embedding matrices. Similarity comparison is performed between the embeddings for the given contract and those in our collected database (step 5), and similarity thresholds are used to govern whether a code fragment in the given contract will be considered as code clones or clone-related bugs (step 5-6).

### B. Details

*1) Parsing:* SMARTEMBED employs ANTLR[3] and a custom Solidity grammar[4] to generate ASTs for each smart contract. Listing 1 shows a simple example of a smart contract defined in Solidity. Depending on the types of the tree nodes, the ASTs is serialized differently for contract-level and statement-level program elements to capture structural information in and around the focal elements.

```
1  pragma solidity ^0.4.15;
2
3  contract Overflow {
4      uint private r=0;
5
6      function addValue(uint value) returns (bool){
7          // possible overflow
8          r += value;
9      }
10 }
```

Listing 1. An Example Solidity Program

**Contract Level Parsing:** All terminal tokens are extracted from the ASTs by an in-order traversal. The contract level parsing result of the sample code is shown below.

```
1 1_10 : pragma solidity ^ versionliteral ; contract
    Overflow { uint private r = 0 ; function
    addValue ( uint value ) returns ( bool ) {  r
    += value ; } }
```

**Statement Level Parsing:** For statement parsing, more structural information (containment and neighbouring) as well as some semantic information (data-flow) is added to the sequences. The statement level parsing result of line 8 is given as follows.

```
1 8_8 : sourceUnit contractDefinition contractPart
    functionDefinition block statement
    simpleStatement r += value ; function addValue
    add value ( uint value ) returns ( bool )
    contract Overflow overflow { }
```

*2) Normalization:* SMARTEMBED normalizes the parsing sequence to remove some semantic-irrelevant information. All simple variables, non-essential punctuation marks and different type of constants are replaced or removed. The following code snippet exemplifies the operation of this step:

```
1     uint private r = 0 ;
2      ==>
3     uint private SimpeVar = decimalnumber
```

*3) Code Embedding Learning:* SMARTEMBED embeds code elements and bug patterns, including their lexical, syntactical, and some semantic information into numerical vectors via techniques adapted from word embeddings. We choose Fasttext [8] as the code embedding algorithm as it performed on par or better compared with traditional word2vec.

***Token Embedding:*** The normalized token streams with structural information generated by the normalizer for the solidity contracts are used as the training corpus. We adapted the Fasttext algorithm to train code embedding models. After the training, each token in the training corpus, including the tokens representing structural information, is mapped to fixed-dimension vector with real values.

***Higher Level Embedding:*** Based on the basic vector representation for each token, the code embeddings for higher-level

[3]https://www.antlr.org/
[4]https://github.com/solidityj/solidity-antlr4

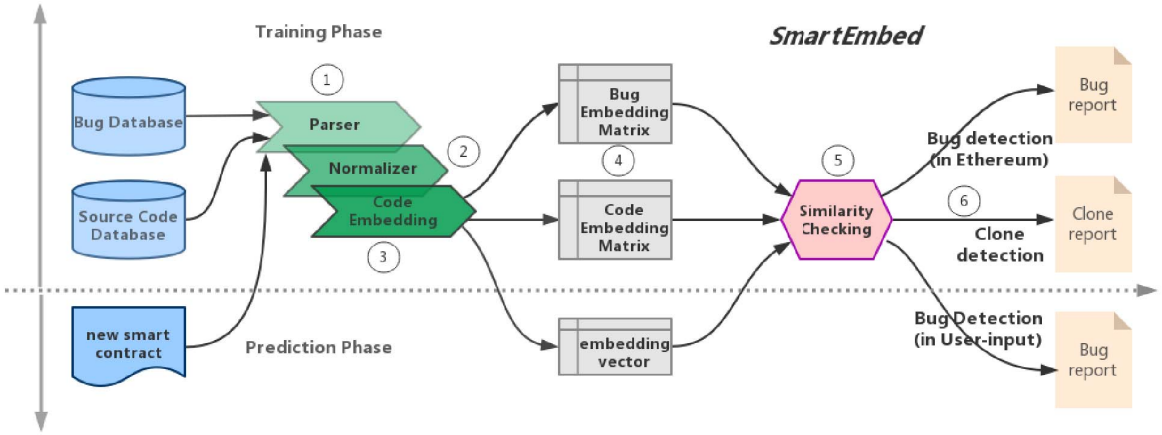


Fig. 1. Overview of Our Approach

code fragments (e.g., statements, functions, subcontracts, and contracts) are composed together. To be more specific, we define the code embeddings for a particular code fragment as the summation of the embeddings of all its constituent tokens.

*4) Embedding Matrices:* By stacking individual vectors together, we obtain a source code embedding matrix $\mathbf{C^{c\times d}}$ for clone detection and a bug statement embedding matrix $\mathbf{B^{b\times d}}$ for bug detection.

***Source Code Embedding Matrix*** $\mathbf{C^{c\times d}}$: The first dimension $c$ is the total number of contracts; the second dimension $d$ is the code embedding size we set previously. The $i^{th}$ element $\mathbf{C_i}$ $(i = 1, 2, ..., c)$ is the vector representation for the $i^{th}$ contract.
***Bug Statement Embedding Matrix*** $\mathbf{B^{b\times d}}$: The first dimension $b$ corresponds to the total number of bug statements in our bug database, and each row of the matrix, i.e., $\mathbf{B_i}$ $(i = 1, 2, ..., b)$ represents the code embedding for a specific bug statement.

*5) Similarity Checking:* We define a similarity metric, which is used in the downstream tasks of clone detection and bug detection.

***Definition***: Let $C_1$ and $C_2$ be two code fragments, and $e_1$ and $e_2$ be their corresponding code embeddings. We define the semantic distance as well as similarity between the two code snippets as follows:

$$Distance(C_1, C_2) = \frac{Euclidean(e_1, e_2)}{\|e_1\| + \|e_2\|} \quad (1)$$

$$Similarity(C_1, C_2) = 1 - Distance(C_1, C_2) \quad (2)$$

Given any two code fragments $C_i$ and $C_j$, if their similarity score is over a specific similarity threshold $\delta$, $C_i$ and $C_j$ are viewed as a clone pair.

*6) Clone Detection and Bug Detection:* Both clone detection and bug detection tasks can be viewed as variants of the problem of finding "similar" code, depending on the definition of similarity. For clone detection, we measure the similarity between pairs of smart contracts, and identify them as clones if the similarity score is over a predefined threshold for clones. For bug detection, we search for code fragments in given contracts that are more similar to the known bugs than a predefined threshold for bugs.

## III. Implementation Details & Tool Usage

We have implemented SmartEmbed as a standalone web service to facilitate Solidity developers in checking their smart contracts. The source code and data can be found in our Github repository[5].

***Data Collection***. We collected 22,275 verified Solidity smart contracts using EtherScan[6], which is a block explorer and analytics platform for Ethereum. The contracts contain 135,239 subcontracts, 631,261 functions, around 2 million statements, and more than 7 million lines of code. In the meanwhile, we collected 22 well-known vulnerable smart contracts and pinpointed 37 buggy statements in the contracts, which served as the bug database for SmartEmbed.

***Backend Model***. The collected source code of contracts are inputted into the workflow of our approach described in Section II, and the output is the code embeddings which are used as the backend model for similarity checking.

***Frontend User Interface***. On the user interface, SmartEmbed provides an input box for Solidity developers to submit their source code (cf. Fig. 2 and Fig. 3). After a Solidity developer submits his/her source code to the server, the source code is parsed and normalized, then the contract and each statement is converted into a vector by our backend model for similarity checking. The outputs are organised into two separate result tabs. For the clone detection result tab, SmartEmbed returns top-5 most similar clone contracts in our code base together with the similarity scores and links to their locations in EtherScan (cf. Fig. 2). For the bug detection result tab, SmartEmbed highlights the buggy lines in the submitted source code and reports the bug types back to the developer (cf. Fig. 3).

## IV. Evaluation

We compared SmartEmbed with two well-known tools that are specific for clone detection (DECKARD [9] extended for Solidity) and bug detection (SmartCheck [3]) respectively.

[5]https://github.com/beyondacm/SmartEmbed
[6]https://etherscan.io/


Fig. 2. Sample Results of Clone Detection

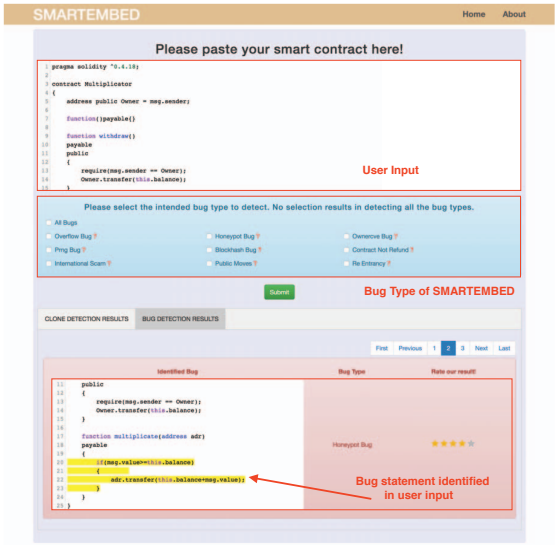


Fig. 3. Sample Results of Bug Detection

For clone detection, we run DECKARD and SMARTEMBED against 22,725 smart contracts source code, the experiment results show that both tools identified around 6.6 million lines of code as code clones, while the total lines are just 7.3 million, which means the clone ratio of solidity code is at around 90%, much higher than traditional software. One main reason for introducing clones is the irreversibility of smart contracts stored in Ethereum blockchain. By manually checking some clones detecked by our approach but not by DECKARD, we found code clones such as type-III or even type-IV semantic code clones can also be detected, which means SMARTEMBED is highly effective to identify the code clones in smart contract. For bug detection, SMARTEMBED can identify clone related bugs in Ethereum blockchain more efficiently and accurately. When the similarity threshold is set to 0.95, our tool reports 202 clone related bugs, we manually validate these candidate bugs and 194 of which are labelled as true bugs, while SmartCheck can only detect 117 of these verified bugs by using the same bug pattern type within our bug database.

## V. SUMMARY AND FUTURE WORK

This paper presented SMARTEMBED, a web service tool for detecting code clones and bugs in smart contracts accurately and efficiently. It develops a code embedding technique for tokens and syntactical structures in Solidity code and utilizes similarity checking to search for "similar" code satisfying certain thresholds. The approach is automated on the contract and bug data collected from the Ethereum blockchain. It helps developers to find repetitive contract code and clone-related bugs in existing contracts. It also helps to efficiently validate given smart contracts against known set of bugs without the need of manually defined bug patterns. Its backend model can be easily updated to recognize new contract clones and new kinds of bugs when the contract code and bugs evolve. In the future, we plan to enrich the contract and bug databases so that SMARTEMBED can detect more clones and bugs.

## ACKNOWLEDGMENT

This research is supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant from SIS at SMU. We also thank the anonymous reviewers for their insightful comments and suggestions.

## REFERENCES

[1] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[2] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *25th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[3] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, WETSEB '18, pages 9–16, New York, NY, USA, 2018. ACM.

[4] Ningyu He, Lei Wu, Haoyu Wang, Yao Guo, and Xuxian Jiang. Characterizing code clones in the ethereum smart contract ecosystem. *arXiv:1905.00272*, 2019.

[5] Han Liu, Zhiqiang Yang, Chao Liu, Yu Jiang, Wenqi Zhao, and Jiaguang Sun. Eclone: Detect semantic clones in ethereum via symbolic transaction sketch. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 900–903, New York, NY, USA, 2018. ACM.

[6] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260, Sep. 2017.

[7] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 17–26, 2015.

[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[9] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 96–105, 2007.