Towards Interpreting Smart Contract against Contract Fraud: A Practical and Automatic Realization

Ming Li, Anjia Yang, *Member, IEEE*, Xinkai Chen

Abstract—Contract fraud is a big nuisance in our society. People are scammed largely because of vague language used in contracts, which can cause misunderstandings. Therefore, people will seek professional help to review over ambiguous terms, especially, when signing a big contract, for example, leasing or buying property. With the advent of Ethereum blockchain, a new type of contract, named smart contract, is emerging nowadays, enabling people to describe a complicated logic as an automatically executable computer program. However, due to the lack of the computer background and software development experience, many people have difficulty in understanding blockchain-based smart contracts, which is adverse to the popularization of Ethereum. It has resulted in a new wave of contract fraud caused by smart contracts, which are self-executing and self-enforcing but also hard to understand by people. To fill this huge gap, we propose an approach to enable people without computer background to understand and operate Ethereum smart contracts. In doing so, smart contract fraud can be deterred if people have a better understanding of contract terms. Particularly, we investigate the general rules of the smart contract code, and build a novel tool named SMTranslator to automatically generate readable document. SMTranslator first translates smart contracts into standard structured files and identifies the core statement of each function in smart contracts. By exploiting the custom natural language generation, we generate the documents for smart contracts that can provide correct and understandable descriptions. We collect numerous contracts in Ethereum and select a number of typical contracts to conduct the experiments. Extensive experimental results demonstrate the feasibility and effectiveness of our approach.

Index Terms—Ethereum, smart contract, contract fraud, fraud deterrence, natural language generation.

1 Introduction

Over the past ten years, blockchain technology has gained considerable attention and adoption in both industrial and academic area since it was coined in 2008 [1]. Bitcoin and Ethereum, as two of the biggest blockchain platforms, have achieved valuations of more than 745 billion US dollars in December 2018. The two blockchain platforms provide a promising way to build a blockchain-based decentralized application (DAPP) that mutually distrust parties can reach an agreement in a secure way without reliance on a third-party. Nowadays, a great deal of DAPPs have been established in the real world, such as Micropayment scheme [2], naming system [3] and crowdsourcing system [4]. We could expect that these decentralized applications will contribute significantly to constitute a more fair society in the current digital age.

One main reason that boosts the prosperous of blockchain is the efficiently supporting for *smart contract*. Smart contract is an automatic self-execution computer program that runs without reliance on a central party. It is published as a transaction in the blockchain network. Particularly, the underlying cryptography techniques and consensus protocol constitute a secure environment for running smart contract. Under the majority honest security, a contract is *tamper-resistant* and *trackable* once being confirmed by blockchain nodes (e.g., *miners*) for several blocks. For this reason, data recordings in smart contract can be presented as a valid proof for judgement in legal disputes [5], which provides a promising way for digital forensics to gather effective evidences against practical issues, e.g., financial criminal.

Generally, the program languages for developing smart contracts are different among the existing blockchain platforms. For example, Bitcoin can only run non Turing-complete scripting language. The complicated program logics, such as *functions* and *exceptions*, are not supported. This limitation makes Bitcoin have the natural disadvantages when being used in other areas. To support Turing-complete smart contracts, Ethereum is designed and employed prevalently as the world's second biggest cryptocurrency. According to the statistics of [6], there are total 14,205 smart contracts have been deployed in Ethereum in the last twelve months in 2018. Our work focuses on Ethereum which involves millions of US dollars in smart contract.

Currently, Ethereum supports three types of program language to develop smart contract: *Solidity, Serpent* and *LLL*. Solidity is the first choice and flagship programming language for developing smart contract in Ethereum. It is a high-level program language that supports arbitrary program logic. In the process of code execution, Solidity is compiled into *bytecode* and executed in Ethereum Virtual Machine. Specially, DAPPs are different from other GUI-based (Graphic User Interface) applications that people do not need to care about the underlying source code. DAPPs require people to provide valid function inputs by using Ethereum Wallet or Remix-IDE. Thus, the primary condition for people to operate a contract is the ability of reading and understanding the Solidity contract. However, people who work in some fields do not have the programming skills but also show great interests in blockchain, e.g., the financial worker. How to help these people to

• *M. Li, A. yang, and X. Chen are with the College of Information Science and Technology and the College of Cyber Security, Jinan University, Guangzhou 510632, China. E-mail: limjnu@gmail.com.*

easily participate in the DAPP ecosystem is still an open challenge.

Apparently, it is not easy to enable an individual without any programming skill to understand the contract code. One practical way is to parse the source code into intelligible description sentences. A series of attempts have been made to generate the source code document for C/C++/Java [7], [8], [9], [10], [11]. However, we can not adopt their schemes directly in Solidity document generation for three reasons: 1) most of them are designed for software maintenance persons or develpers who have a certain programming skill, not for individuals without the computer background [9]. 2) Some of their schemes summarize the meaning of a function according to the function name [10], [11]. However, not each function name in smart contract can present the accurate meaning. According to our observation, the action performed in a certain function may not correspond with the *verb* described in its name. 3) The existing researches on documentation for program language focus on explaining the meaning of the source code. However, in terms of smart contract documentation generation, the primary focus is to discover and present the vulnerabilities in the generated document other than the original meaning. In summary, solidity is a newly appeared program language that there does not exist an effective tool to automatically generate understandable document at present.

Motivated by the aforementioned issues, we propose **SM**art contract **Translator** (SMTranslator), an automatic document generation scheme for the Solidity smart contract. We implement a system prototype to verify our scheme. Particularly, we define a standard template to describe a function which contains *Summary Description, Short Description, Return Description, Input Description, Core Statement Description* and *Call Description*. Summary Description is to help people to have a general view of the contract. The rest parts are to generate comments for a special function. First, SMTranslator translates the Solidity contract into structured XML representation, which can facilitate us to obtain each part of the source code. Then, we make the core statement analysis to identify the most important action in a function, and parse the identified core statements based on part-of-speech analysis. Lastly, we build a custom natural language processing system to reorganize the key words and generate the final document. Evaluation results indicate that the generated document can help people without programming background to understand Solidity contract and provide correct inputs when publishing a transaction. In a nutshell, our specific **contributions** can be recognized as follows:

- An approach for automatically generating document for Solidity smart contracts is proposed in this work. To the best of our knowledge, this is the first work that generates readable document of smart contract for people who do not have the programming background.
- Our approach is different from the previous document generation approaches for C/C++/Java. A novel approach is designed to summarize the meaning of the contract and present a readable description for each function. We convert the Solidity contracts into XML format files. Then, the core statement analysis is designed to find the core action of the method. We also identify the special expressions and features of Solidity and represent them with suitable words. The custom natural language processing is developed to parse the key works and generate the readable English sentences.
- A system prototype is developed for Solidity contract and we have released our tool to the public as an helpful tool to understand smart contract for people. We collect a large number of Solidity contracts from Ethereum and conduct extensive experiments to verify the usability and feasibility of our approach.

The remainder of the paper is organized as follows. In Section 2, we introduce the background and formulate the motivation of this work. In Section 3, we present our concrete approach to generate Solidity document. In Section 4, we present the experiments and evaluation results. The related works are give in Section 5. Finally, the conclusion and future works are given in Section 6.

2 Background and Motivation

2.1 Background and Preliminaries

Blockchain and Smart Contract. Most recently, blockchain has gained significant attention and has been deployed in different scenarios [2], [12], [13]. It is essentially a distributed ledger which are maintained by a number of network nodes [14]. These mutual distrust nodes can reach an agreement by the consensus protocol, e.g., proof of work and proof of stake. More in detail, blockchain is compose of a series of *blocks*. Each block, organized together as an ordered hash chain, contains lots of *transactions*. Particularly, the review of main features on blockchain are listed as follows: 1) *Complete Decentralization*: Blockchain is a global computer that is maintained by distributed P2P network. Many mutual distrust nodes are able realize fairly data communication without relying on a central third party. 2) *Correctness* and *Traceability*: Blockchain is transparent data structure that each node can trace and verify the correctness of the data. The validation of data is ensured with the underlying cryptographic tools (e.g., digital signature, hash function). 3) *Immutability* and *Irreversibility*: Transactions are tamper-resistant since they are organized as Merkle has tree. Blocks are also connected together as hash chain which ensures the immutability and irreversibility. 4) *Cryptography*: The security of blockchain is compose of the underlying cryptography techniques which ensure the transfer of the digital currency or status among different parties in a secure way.

And also, smart contract was first proposed by Szabo in 1997 [15] before the invention of blockchain. It is a main component of blockchain technology that provides Turing-complete programming language (i.e., arbitrary computer codes execution), which allows blockchain to be applied in many applications [4], [16], [17]. Ethereum is the first blockchain platform that supports Turing-complete smart contracts which are executed in the form of transactions. Specifically, smart contract in Ethereum is converted into bytecodes that are run in Ethereum Virtual Machine (EVM). It mainly contains two parts: 1) version declaration, and 2) contract body. The version declaration is to ensure that the contract can not be compiled with a breaking compiler version. The contract body contains the variable declaration and function definition. Each function is identified by a unique name and type parameters which are regarded as the signature statement. In particular, there exist comments which can be identified by the symbol /***/ in the contract body to provide the overview of function. A declaration of a function is called as a signature (or statement) [1].

1. We recommend the readers to refer to [18] for more information.

```
1  mapping(address => uint) private userBalances;
2  contract TokenSample {
3      function transfer(address to, uint amount) {
4          if (userBalances[msg.sender] >= amount) {
5              userBalance[to] += amount;
6              userBalance[msg.sender] -= amount;
7          }
8      }
9
10     function withdrawBalance() public {
11         uint amountToWithdraw = userBalances[msg.sender];
12         require(msg.sender.call.value(amountToWithdraw)());
13         userBalances[msg.sender] = 0;
14     }
15 }
```

Fig. 1: An example of smart contract vulnerability.

Control Flow Graph. Control flow graph (CFG) is a type of graphical representation that a several nodes and edges are utilized to represent the control flow of the execution of programs and applications [19]. Each CFG node is a basic block which denotes a straight-line piece of code in program. Direct edges are used to denote the jumps among different blocks. Note that there are two type of designated blocks, i.e., entry block and exit block. The first one is the enters of the flow graph and the second one is exit. Specifically, CFG has been used in many program execution scenarios to show the visualization of the program. Our scheme employs CFG to present the control flow of the smart contract, which enables people to understand the process of the concrete execution.

PageRank. PageRank is an algorithm that is proposed by Google Search to rank the web pages in their search engine. It can be used to approximate the importance of the nodes in a graph [10]. Generally, PageRank computes the importance of a given node based on the number of edges which point to the node and the importance of the nodes from which those edges originate. It has also been used to highlight the importance of functions or methods in a software project [10], [20]. A method that is called by many times by other methods is regraded as more important that other method. We utilize PageRank to rank the importance of smart contract codes in a specific function in this paper.

Natural Language Processing. Due to the complex and diverse of the natural languages, it is not easy to make computers truly understand what the meaning of a speech or text in the real-world. Natural language processing (NLP), a subfield of artificial intelligence, is used to mitigate this challenge that enables computers to understand and process human natural languages [21]. It can parse and produce human natural language under the spoken or written form. NLP mainly involves three parts: speech recognition, natural language understanding and natural language generation. *Stanford CoreNLP* is a widely used toolkit which has realized core steps for NLP [22]. We use this toolkit to analyze the statement of the contract code and generate readable English sentences.

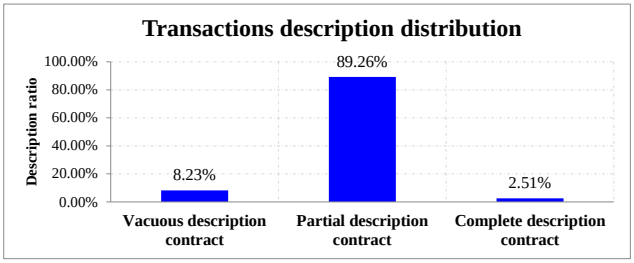

Fig. 2: Ethereum transaction volume statistics.

2.2 Problem Statement and Motivation

The main motivation for this work is described as follows: currently, contract fraud in blockchain has caused tremendous economic losses. There does not existing an effective approach to guide people without technique background to understand the correct meaning of smart contracts and run the functions, which is apparently adverse to the popularization of blockchain. More preciously, people who intend to take participate in a DAPP should first understand its purpose, then they can call the function of a smart contract using the secret key. They can only resort to the skilled developers or contract comments if they are short of the special technique background. However, both methods have some limitations. On the one hand, part of contract developers do not provide contract comments, or the code comments are not completed to illustrate the meaning of the contracts according to our investigation. On the other hand, solidity is a newly emerging program language that the corresponding developers are few at present.

Particularly, there exist many security vulnerabilities which are hard to be distinguished even for the developers. To illustrate, we take an example as shown in Fig. 1. This contract allows people to earn the token by transfer ETH to this contract. It is not easy to identify the vulnerability of this contract for majority of people. So they may choose to participate in this DAPP without doubt. However, an attacker can call transfer() when his code is executed on the external call in withdrawBalance(). Thus, before his balance not being set to 0, the attacker can still transfer the tokens even though he has received the withdrawal. This vulnerability is called cross-function reentrancy which is also exploited in "DAO attack" [17].

In addition, as for the smart contract comments, we classify the smart contracts into three types based on the completeness of comments [23]: *Vacuous description contract* refers to the contracts that contain hardly any comments. *Partial description contract* refers to the contracts that only part of some special functions are commented. *Complete description contract* refers to the contracts that all of their functions have been commented, including the signature, and input/output parameters. Specifically, we download 6,862 smart contracts which have launched more than 100 transactions after being deployed from the beginning of Ethereum[2]. The transactions distribution of these contracts are shown as in Fig. 2. According to our observations, only 41.81% of the total functions (74,615 out of 178,445) have been commented in the total 6,862 contracts. Furthermore, 8.23% (i.e., 565 out of 6,862) of the contracts belong to the *vacuous description contract*, 89.26% (i.e., 6,125 out of 6,862) of the contracts belongs to *partial*

2. Etherscan just shows only the latest 500 verified contracts source code at the time of writing.

description contract, and only 2.51% (i.e., 172 out of 6,862) of the contracts belong to *complete description contract*. Namely, more than 97% of smart contracts do not provide complete comments. Specially, we find that some comments are even inaccurate. In "CAIDCrowdsale.sol", the developers claim the copyright notice in the comments but not present the correct meaning. In addition, we find some of the comments are written by non-English languages, which precludes some people from understanding the contract. Besides, "SiaCashCoin.sol" which has the maximum number of transactions (837,794 transactions) does not provide any descriptions or comments in the contract.

The above issues motivate us to design an effective approach to normalize Solidity contract comments and guide people to participate in the DAPP without being defrauded. Based on our approach, people without technique skills can understand how to utilize the smart contract to participate in the Ethereum ecosystem. Furthermore, we mark the vulnerabilities of a smart contract, which can prevent people from suffering from the economic losses.

3 Approach

In this section, we will first present the overview of our approach SMTranslator, and then describe the details.

3.1 Overview of SMTranslator

Our approach is based on the analysis on the deployed Solidity contracts in Ethereum. The methodology can be summarized as four phases: Data Collection, Smart Contract Structure Processing, Smart Contract Analysis and Natural Language Generation. As for the first phase, we collect Solidity contracts from the open public Ethereum website Etherscan[3]. We identify the differences between Solidity and other program languages, and figure out the main features of the published contracts. During the second phase, SMTranslator converts Solidity contract code into regular data structure, which can effectively improve the interpretation with the code details. After the two preparation phases, we analyze the key words and core statements performed in functions and aim to obtain the core meaning of each method in the third phase, and draw the visual CFG to depict the process of contract functions. Specially, the vulnerabilities within a function will be marked in the CFG. The last phase is to generate the final readable contract documentation based on NLG.

The design of SMTranslator is illustrated in Fig. 3. During the contract structure processing phase, we generate the customized XML file for Solidity based on SmartCheck [24], which can promote the efficiency of the latter parts (e.g., contract visualization and contract analysis). After that, we analyze smart contract from two aspects: contract visualization and contract analysis. We utilize CFG to describe the execution process of a function with a graph, and enumerate the importance of statements by PageRank. The special statements are identified in Solidity, such as ether transfer, event and modifier claim. These statements are useful to enable people to understand the main meaning of functions.

3.2 Data Collection

We collect smart contracts which are published in succession from the launch of July 30th, 2015. Due to the restrictions of the Etherscan, it is not allowed to crawl all of smart contract source codes. We extract two principles to collect some representative smart contracts: first, the developers deploy their smart contracts in Ethereum to support practical applications, so the number of confirmed transactions on a smart contract can represent the activity and utility of the smart contract to a certain content. However, 66.82% of smart contracts do not have more than 10 transactions after being deployed. We collect the highly visited smart contracts that have more than 100 confirmed transactions. Second, we choose smart contracts that their size is more than 1KB, i.e., about more than 100 lines code. Based to the two rules, we collect 964 Solidity contracts from Etherscan. The majority of smart contracts have more than 150 lines of code.

3. https://etherscan.io/contractsVerified

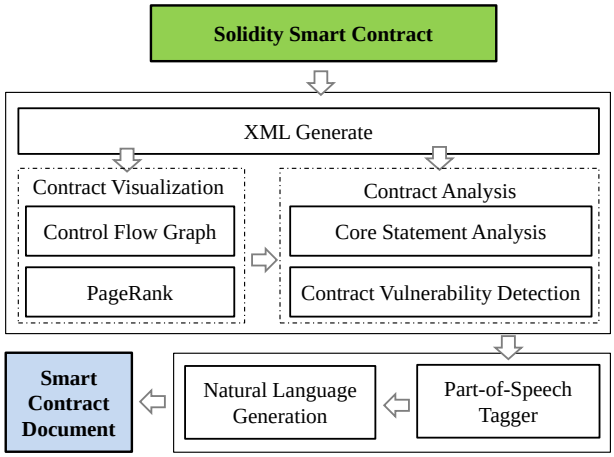

Fig. 3: The architecture of SMTranslator.

3.3 Smart Contract Structure Processing

The design goal of XML generation in SMTranslator is to simplify Solidity contract into general structured document format. SMTranslator adopts the standard Extensible Markup Language (XML) to represent the original contract code. XML is a makeup language that is easy to be understood both for human and computer. It does not lose any information from Solidity contract into XML representation, which is vital to maintain the original meaning of the contract. We generate the XML document by referring to SmartCheck [24], a static analysis tool which aims to automatic vulnerability detection. SMTranslator translates source code into XML, and also includes all the original information in contract (e.g., comments and description), which are helpful to generate the contract document.

As shown in Fig.**??**, it is the generated XML file for "SimpleStorage.xml" by SMTranslator. The declaration of contract is in the element of "$<contract><\backslash contract>$". The identifier of the contract is the short description of a contract. We can interpret the identifier by our custom NLP. The element of "$<comment><\backslash comment>$" refers to the method comments. Each method is identified in the element of "$<function><\backslash function>$". According to the explicit XML elements, SMTranslator can locate and obtain a specific code easily through XML Application Programming Interfaces (APIs).

3.4 Smart Contract Analysis

The reminder of this section describes the analysis of the contract core statement, stereotype identification, and vulnerability

detection.

3.4.1 Core Statement Analysis

The expected results of Solidity document generation is to reveal the function and utility of a contract. To achieve this, we need to analyze the core actions or important statements performed in a given function and present the call dependency. In general, consider the distinctiveness of public blockchain platform, it is inadvisable to develop an application only by using Solidity. Developers are inclined to develop a blockchain-based application with multiple program languages. For example, they may develop the core process (coin transfer or status transition) in Solidity, and develop the intermediate logic and user interfaces (UI) in Java and Javascript, respectively. It is usually considered that the codes in Solidity contract are important statements in most cases. Namely, it is more easily to identify the core statement in Solidity contract compared with other program languages. We aim to use a few descriptive sentences to enable readers to know the meaning of a method. Based on the previous experience of Hill, *et al.* [9] and our extensive observation, we summarize the following principles to obtain the core statement of a function.[4]

Ending Statement. It refers to the statements that lie in the end of a method. It is usually used to change the state/value of a variable, or execute method call. The core statement for a function with a return value corresponds to the assignment statement for the end return. It is easy to identify Ending in the *void* return type function that he last lines are usually the core statements. For example, in the below fucntion *vote(uint8 toProposal)*, we can not summarize the meaning of this method according to its name (*vote*). In view of the *void* return type, we locate the end of the three lines from 4 to 6 as the core statements and use them to improve the summary. Specifically, "." in Solidity refers to an attribution of a variable. Thus, "sender.voted" is described as "sender's voted".

EtherUpdate Statement. In Ethereum, the primary concern for participants is the security of their accounts. So any update on their account balance may be corresponding to the core statement. EtherUpdate is the statements that are related with account update. Particularly, there are three ways to perform Ether (the digital asset in Ethereum) transfer in Solidity: 1) *address.transfer()*, 2) *address.send()*, 3) *address.call.value().gas()*. We observed that most of the contracts use the first method *address.transfer()* to transfer currency. The main reason is that *address.transfer()* can throw exception if there exist an error, which may be more secure for individuals.

In the above function, the name *distr* is not a normal word that can be understood easily. SMTranslator generates the description by combining the main action performed in the method. The first two lines are to set the status of the variables. The third line is the main purpose of this method. It sends *amount* Ether from *address(0)* to *to*. The action "Transfer" reminds people that their account will update after the execution of this method. In addition, according to our observing, many contracts adopt the name "from" and "to" to present the currency from sender's address to receiver's address. SMTranslator interprets this type of Ether transfer action as: "*amount* is sent from address *from* to address *to*".

EventClaim Statement. It refers to the statements that indicate important events in a function. Solidity contract contains a special declaration called "event". It reminds people that an important action will be executed in this function. Generally, people can use the interface of Ethereum client to listen to an "event". Once it is called, the arguments will be recorded in the transaction log. We find some general rules in terms of "event" method that some particular functions appear with high frequency, such as *Deposit(·), Transfer(·)*. These events are related with the update of the account balance. On the other hand, there exist some "event" methods which are used to change the status of a variable or execute specific action, such as *Approval(·), OwnerChanged(·) and Pause(·)*. SMTranslator identifies the declaration of EventClaim statement by the declared *verb* "emit", and interprets the meaning with the "event" name and its input parameters. Apart from the "event" statement to record important actions, some methods may also contain other core statement declarations. Thus, in terms of the "event" method interpretation, we usually combine EventClaim statement with other statements together.

SameAction Statement. It refers to the statements that there exists a method call *Func* which has the same action with the function. In general, *Func* contains the same *verb* word with the method name. For example, line 1 is the method signature and the method name is "issueMaxSynths". On line 5, the method call "issueSynths()" can be analyzed by the camel-case that it has the same action with the method signature. They have the same *verb* "issue" and *noun* "Synths". If there exist only one line code in a method, it is obviously the core statement and the method call usually has the same action with the method.

Conditional Statement. It refers to the executions which are based on special conditions. Detailedly, it ensures that the specific codes can be executed only if some special conditions are satisfied. SMTranslator identifies this type of statement with the key words *if, while, for* or *switch*. There may have multilayer nestings using *if* statement. In this situation, SMTranslator reveals the main execution code in the statement and illustrates the meaning of the judgements together. Particularly, Solidity contract contains some special conditional judgement statements which are used to handle error, e.g., *require* and *assert*. They can throw an exception and return immediately in case of the conditions are not met. We can see that the conditional statements contribute a lot to help users to understand when or how a method can be executed. So SMTranslator illustrates the conditional statements in the document.

Modifier Statement. It is the declaration that verifies some special conditions before a method execution. It is declared in the signature of a method. Modifier statement usually defines some conditional judgements with *require* statements and can change the performed action of the method. SMTranslator identifies Modifier statements with the declaration "modifier". A method can have multiple 'modifier" statements. SMTranslator interprets the condition judgement of "modifier" one by one and generates fixed description format for any method.

3.5 Natural Language Generation

Natural language processing (NLP) is described as the process of producing meaningful phrases and sentences in the form of natural language to do useful things [25]. It is used to analyze human language by combining machine learning and deep learning algorithms, and aims to make computer understand the meaning of various differences of language without being explicitly told. In our work, we utilize NLP technology to analyze the core statements and input/output parameters of Solidity contract. For

4. The contract codes we analyzed below take from Ethereum.

example, a specific word with camel-casing naming is parsed as the gerund form and recognized as a readable sentence with proper preposition.

SMTranslator adopts the Stanford CoreNLP NLP Toolkit which is a Java annotation pipeline framework [22] to provide core natural language analysis. It contains many NLP tools, including Part-of-Speech (POS) tagger, Named Entity Recognizer (NER), Parser, Coreference Resolution System and so on. Most of the Solidity contracts follow the naming rules of variable and method. So SMTranslator can interpret different part of words accurately in the statement and declaration based on POS tagger. Take a method *ManagedAccount(address _owner, bool _payOwnerOnly)* in "DAO.sol" for example. The method name "ManagedAccount" is parsed into two words. The first word "Managed" is the past tense verb which is marked as {*"pos": "VBD"*} and shows the base form as {*"lema": "manage"*}. "Account" is marked as {*"pos": "NN"*} and the base form is {*"lema": "account"*}. The input parameter "_payOwnerOnly" can be parsed as the same way, "Only" is marked as {*"pos": "RB"*}. "pos" refers to element of the part of speech. POS tagger has defined more than 40 types of "pos". "VBD", "NN" and "RB" represent the verb with base form, noun with singular or mass and adverb, respectively.

After the analysis of POS tagger, the parameters and core statement are turn into structured data, which let us obtain the separate key words. Based on this, the final phase of SMTranslator is to use Natural Language Generation to organize these words and generate readable English sentences.

Compared with POS tagger that deals with "Reading" task in Solidity contract, Natural Language Generation (NLG) can be deemed as executing "Writing" task. It aims to turn structured data into human readable sentences. Particularly, SMTranslator follows the typical architecture of NLG described in [26]. It mainly contains three components: Document Planner, Microplanner and Surface Realizer. Document planner is the component that interprets the performed fact/action in each method and organizes them as a sequence which can be easily understood. Microplanner is the component that determines which suitable words or phrases can be used to describe the sequence. To interpret different parts of a method, the microplanner adds some specific words into a phrase (e.g., adjectives or adverbs), which can smooth it more readable. The last component surface realizer is to organize these phrases as natural language sentences. As described above (Subsection 3.3, 3.4 and 3.5), we first convert Solidity contract into the structured XML format and analyze the core statements and important declarations as the inputs of document planner. Then, we generate the fixed structure for each specific statement in surface realizer.

It is worth noting that we identify an important point which can help us to generate the document in NLG: parsing global variables and functions under the explanation of Solidity documentation [5]. We aware that many global variables and functions have deterministic meaning in Solidity contract. Interpreting these variables and functions in advance can effectively help people to understand the function more clearly. For example, *msg* is the initiator of a contract, e.g., *msg.sender* refers to the initiator's address, *msg.value* refers to the number of currency that the initiator transfers to the contract. The method *sha256(var m)* is to compute a hash value of *m* which is the digest of this message. It is a cryptography algorithm that people can not understand the meaning only by its method name. In addition, there is another special method called *selfdestruct(address recipient)*. It destroys the current contract and sends its remaining currency to the specific address *recipient*, which is a very special method that does not exist in other languages. SMTranslator interprets these special variables and functions with pre-described sentences in the generated document.

5. https://Solidity.readthedocs.io/en/latest/Solidity-in-depth.html

TABLE 1: The explanation of description type.

Description Type	Explanation
Short Description	The summary about this method.
Return Description	Return type and value explanation.
Modifier Description	Special conditions that the method should satisfy.
Input Description	Input parameter explanation.
Core Description	The core statement about this method.
Call Description	Method calls about this method.

SMTranslator organizes the document structure by creating 6 types of description for each method in document planner [10]: *Short Description, Return Description, Modifier Description, Input Description, Core Statement Description* and *Call Description*. The order of these descriptions is determined by the sentences logic and semantic analysis. *Short description* reveals the most important information of the method which is put in the first place, which requires people to pay more attention. It emphasizes some specific actions or facts like *Ether transfer* or *status change*, and represents a brief, highlevel action summarizing a whole method. *Return description* describes the return value. It clarifies the type of output parameters. It is usually put together with *Short description*. *Modifier description* is to declare special conditions (i.e., method with "modifier") that should be satisfied before the execution. *Input Description* clarifies all of the input parameters. *Core Statement Description* serves to indicate the main function of this method. *Call Description* is to illustrate which methods depend on this method. It is used to evaluate the importance of a method. These descriptions are briefly shown in Table 1.

We describe the above 6 type of descriptions with different phases. *Short Description* uses the subject "This method", the *verb* phase "can be used to", the *verb* identified from its name to represent main function, and combines with a *noun* to represent the direct-object. *Return Description* is added in the end of *Short Description* with the *conjunction* "and returns". For example, in the method *function receiveEther() returns(bool)*, it is interpreted as: "*This method receiveEther() can be used to receive Ether and returns bool value*". *Modifier Description* is to describe the special conditions. SMTranslator analyzes the conditions to be met in the declaration of the modifier and interprets it as: "This method can only be called if". In *Input Description*, SMTranslator presents the number of inputs unless there exists only one input. It uses the *verb* "is" to illustrate each parameter one by one. For the convenient operation of people, SMTranslator interprets the type meaning of each input and presents the base form of this type. For example, the parameter "*address receiver*", SMTranslator interprets the variable *receiver* as: "*The variable receiver is the address type that holds a 20 byte value, e.g., 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2*". *Call Description* clarifies which functions have called this method. SMTranslator parses it as "This method is called by:". These phases and sentences are organized together to generate the complete document in the surface realizer.

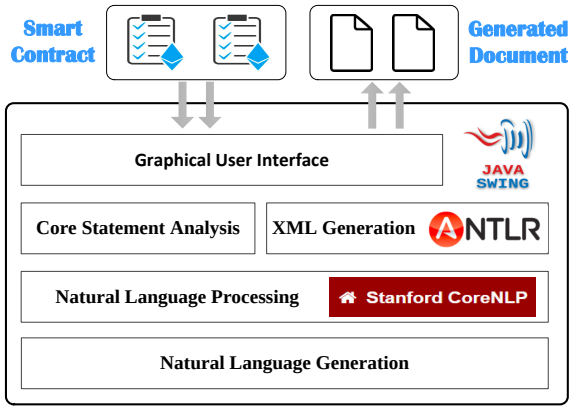

Fig. 4: The system design of SMTranslator.

4 Evaluation Results and Analysis

In this section, we introduce the developed SMTranslator tool and present the evaluation results. We discuss the efficiency and feasibility of SMTranslator by referring to the principles formulated in [10], [11], [27], and mainly focus on three aspects: 1) to access whether our tool can help people without programming background to understand the functionality of a method in smart contract, 2) to access whether our tool can help them to understand how to use a method, 3) to access whether the generated documentation can be more instructive or accurate than the existed comments. For the last point, we consider that whether the generated document can contribute more information about a contract, not only repeat the information that already exists in the comments.

The reminder of this section first presents the prototype implementation of SMTranslator, then introduces the preparation and metrics of the evaluation, and analyzes the experiment results lastly.

4.1 Prototype Implementation

We implemented SMTranslator in Java with roughly 4000 lines of code. Our source code is available at *https://github.com/lim60/SMTranslator*. We develop SMTranslator as a graphical user interface (GUI) tool based on Java Swing library in Eclipse platform, which could make it more easily be operated by people. SMTranslator takes Solidity contract as input and support to bulk import a number of smart contracts. In particular, we have already downloaded numerous smart contracts locally. People can check the existence of a particular contract using its contract name. After finishing the interpretation, people could get a generated document for contract *name* which is named as "{*name*}_document.txt".

There are four main modules in SMTranslator. In terms of XML generation, we adopt ANTLR v4, a parser generator for reading, processing and translating structured text or binary files. It converts Solidity contract format into structured XML data format. SMTranslator integrates Stanford CoreNLP toolkit to provide part of speech analysis. It is worth noting that CoreNLP provides numerous APIs, which allows us more easily to develop the application. Fig. 5 is the system design of SMTranslator. We run our experiments on a PC with a 3.5-GHz CPU, and 16-GB memory.

TABLE 2: The questions we ask in the questionnaire. The optional answers are "Strongly Agree", "Agree", "Disagree" and "Strongly Disagree".

Type	Question
$Q_1 - Usability$	I feel this tool is easy to use and operate.
$Q_2 - Accuracy$	The explanations and summaries for a method is accurate.
$Q_3 - Readability$	The summaries generated by this tool are easy to read and I can totally understand the meaning of each generated sentences.
$Q_4 - Conciseness$	The summaries generated by this tool do not contain unnecessary information.
$Q_5 - Instructiveness$	I can easily use a specific method by the Ethereum wallet under the direction of the explanations.
$Q_6 - CoreAnalysis$	I feel the tool for core statement analysis of a method is accurate and does not miss some important information.

4.2 Participants

To verify the readability and intelligibility of document generated by SMTranslator, we invited 10 student volunteers from Jinan University in China. They all have some basic knowledge of blockchain technology and Ethereum platform. 4 of them are graduate students who come from Computer-Science. They have the experiences of software development. 3 of them are graduate students who come from Marketing-Management. The rest of them come from Economics. In particular, the volunteers are required to finish the questionnaires based on the generated documents and the contract codes.

4.3 Questions and Metrics

To access whether SMTranslator performs well for the above mentioned principles, we list several questions by the form of a questionnaire. As shown in Table 2, there are seven questions about the usability of the tool, the accuracy, readability, conciseness, intelligibility and instructiveness of the generated document. We assign an extra question for the volunteers who are the Computer-Science background. Only they can verify the accuracy of the core statement analysis by checking the Solidity contract code. The optional answers for each question can be "Strongly Agree", "Agree", "Disagree", and "Strongly Disagree". We also assigned a value for each answer which are 4, 3, 2, 1, respectively [10].

4.4 Selected Smart Contract

As shown in table 3, we used 10 typical Solidity contracts to evaluate SMTranslator and conduct the investigation. First, to compare our generated summaries with the comments written by contract authors, we select 4 contracts that belong to *complete description* type. We intend to find whether the existing comment represents the core action of the method. Participants could first check whether the existing comments can help them to use the methods in these 4 contracts. Then they refer to the generated summaries by our tool. In addition, we choose another 6 contracts belong to *vacuous description* and *partial description* type. Most of the methods in the 10 contracts are public and can be used, which let people to test the instructiveness of the document. Take the "SiaCashCoin.sol" for example, it creates a cryptocurrency

TABLE 3: The selected smart contracts in the experiments.

Type	Contract	Total Functions	Commented Functions	Size (KB)
Particial Description	MossCoin	13	5	7.8
	CCEToken	18	12	7.9
	DeusETH	26	3	6.8
	TutorialToken	19	12	7.7
	VocToken	26	9	7.3
	AMNToken	19	13	7.7
	ZmineToken	20	9	7.8
Vacuous Description	XBORNID	30	0	7.9
	XBR	32	0	7.8
	SiaCashCoin	29	0	7.1

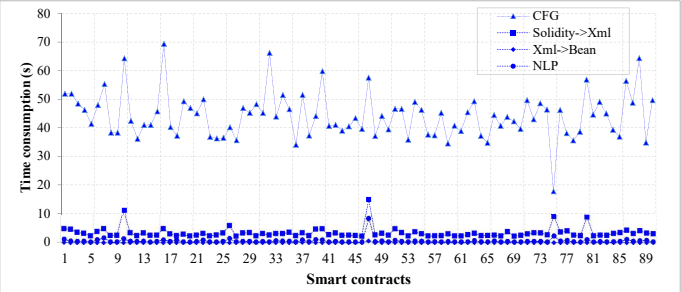

Fig. 5: Time performance on the smart contract analysis.

that aims to improve the payment of data storage based on smart contract. It contains 29 methods and none of them have comments. Using the Ethereum wallet, people could use the method and pay for the service for saving their data. Generally, it can not be finished by people without software background. In our experiments, we will verify whether the 6 persons who come from non-Computer Science can understand and use the method. The next section will analyze the results of the questionnaires based on the generated 10 summary documents.

4.5 Results Analysis

Usability. In the usability judgement over the 10 participants, there were 8 participants who rated "Agree" (5) and "Strongly Agree" (3). The majority of the volunteers agreed that the tool was easy to generate a summary document for a Solidity contract. There were still 2 participants rated "Disagree". The main concern of them was the size of the tool. They hoped that the size of SMTranslator could become small. We found that the reason for the big size lied in the library of Standford CoreNLP (*stanford-corenlp-3.9.2-models.jar*) which was about 345M. To address this issue, we will consider to provide the Solidity contract interpretation service based on browser/server architecture which is more convenient to use.

Readability* and *Conciseness. The readability and conciseness mainly focus on the correctness and intelligibility of the generated sentences. In most of the cases, the sentences are short and have fixed form, which make volunteers be easy to understand. We added some *verb* to describe a method when there does not exist *verb* in the signature, e.g., "handle", "process" and "create". 6 of 10 participants responses shown that the generated summary was readable and concise. However, there are some issues when the declaration of the method signature is irregular. For example, in the method *_memcpy(uint _dest, uint _src, uint _len)* in "EtherDogCore.sol", CoreNLP identifies "cpy" as the *verb* and "mem" as the *noun*. The short description is described as "The function is used to cpy mem". Apparently, the *verb* "cpy" is not a correct word and can not reveal any meaning. We will tackle the irregular words interpretation in our future work.

Instructiveness. It is a very important measurable indicator of SMTranslator on whether SMTranslator can guide people to use the method. We require each volunteer obtains her/his key pair in the test Blockchain network, and conducts the practical operation for some method. We found that 9 of 10 participants rated "Strongly Agree" and "Agree" for the document. 1 participants felt the introduction of some input parameters were hard to understand, e.g., *struct*. We also got some feedbacks on the introduction of the input parameters that it is better to illustrate what a suitable value should be given for an input.

Accuracy. In terms of accuracy, only the Computer-Science background volunteers are required to conduct the investigation. It aims to identify that the main function of a method are corresponding with the generated summary in the document. We found that 3 of the 4 participants rated "Agree" for the generated summary documents. 1 volunteer rated as "Disagree".

Core Analysis. In the investigation of core statement analysis, we just let the 4 participants with Computer-Science background to participate in and randomly select 50 methods interpretation from the generated document. Each volunteer is required to to check whether the identified core statement is accurate. We marked out the core statement for each method and provided some instructions when they read the Solidity contract code. They examined the generated summary and rated them with the four answers. In addition, they have the opportunity to provide some suggestions for improvement on SMTranslator.

According to the analysis of the results, we found that about 65.3% of the methods are rated with "Agree" and 23.6% were rated with "Disagree". There has 13.6% of the methods were rated with "Strongly Disagree". When the participants read the summaries for a special method, they found some important information is missed. SMTranslator just gave part of the core statements. In Solidity contract, there exist lots of methods that belongs to *bool return* type. In these method, the last lines set the value of status for a variable. Thus, it is necessary to parse all the lines to summarize the meaning. In addition, we found that when a method has many lines, we missed to parse some core statements in the middle position. We realize that most of source code in Solidity contract have some important revealing and need to interpret the whole method by analyze all of the lines. We will introduce the action dependency analysis into SMTranslator.

5 Related Work

To the best of our knowledge, SMTranslator is the first work that generates readable English sentences for Solidity smart contract. We also identify that there are some related research works. A briefly discussion is given in this section.

5.1 Documentation Generation for Code

Documentation generation techniques in program language attempt to generate readable natural language sentences for developers, which can significantly improve their work efficiency. Developers can be relieved from tedious writing source code documentation and help the successor to understand the code quickly. As smart contract is a newly emerging program language, the previous works mainly focused on $Java/C++/C/C\#$ language. Emily *et al.* presented a technique to automatically generate descriptive summary comments for Java methods [9], [28]. They designed the Software Word Usage Model (SWUM) to capture the action, theme and arguments for a given method. Due to the limitation that the generated documents can not interpret the context of the source code accurately in some situation, Paul *et al.* proposed a automatically documentation generation technique which can analyze how a specific method was invoked [10]. They utilized static call graph and PageRank algorithm to analyze the relationship and importance of the code methods. Recently, Benwen *et al.* proposed an approach to generate descriptive name for unit tests [11]. Their goal was to let the developers to understand the purpose of a test. This approach built the action dependency graph to identify the test scenario.

Our approach is different from these approaches in that we aim to create the readable English sentences that people without any programming skill can understand the contract code. Thus, we combine the different parts of a method (e.g., method name, modifier, input/output parameters and core statement) with Natural Language Processing to illustrate how the method works and its main function.

5.2 Smart Contract Analysis

There exist some research works on smart contract analysis which are mainly related with Bitcoin and Ethereum. Most of the them focused on the security and privacy issues of the contract code [24], [29], [30], [31]. Sergei *et al.* proposed SmartCheck which aimed to detect code issues in Solidity contract [24]. It translated Solidity contract into XML-based intermediate format, which can be utilized to analyze the source code by SMTranslator. Loi *et al.* designed a symbolic execution tool called Oyente to detect the potential security bugs in Solidity [17]. But due to the different design goals, SmartCheck and Oyente can not be directly adopted to generate descriptive sentences to make people understand smart contract.

6 Conclusion and Future Work

We find a practical problem that people in different areas show great interests in blockchain-based applications while lack of programming skill in understanding the contract code. To fill this gap, we propose an approach for automatic document generation for Solidity smart contract and design a system prototype named SMTranslator. By analyzing the particularities and features of the Solidity contract, we covert the code into the structured XML formation and use core statement analysis to obtain the important function of a method. Natural language processing method is used to interpret the identified statements and generate the understandable English sentences. Finally, the implementation and evaluation are conducted for our tool with a large number of smart contracts in the real world.

In addition to extend and improve this tool, we identify several directions for the future work. First, the goal of people to understand the smart contract is to use the blockchain-based application. However, it will cause severe loss if there exist potential security issues in the contract code. Thus, it is critical for SMTranslator not only interpret the source code correctly, but also can recognize the security issues and reveal the issues to people. Second, supporting static action dependency analysis for a particular method is necessary to reveal the whole meaning of a method.

Acknowledgments

This work was supported by National Key R&D Plan of China (Grant No. 2017YFB0802203, 2018YFB1003701), National Natural Science Foundation of China (Grant Nos. 61825203, U1736203, 61702222, 61472165, 61732021, 61877029, 61872153, 61802145, U1636209), National Joint Engineering Research Center of Network Security Detection and Protection Technology, Guangdong Provincial Special Funds for Applied Technology Research and Development and Transformation of Important Scientific and Technological Achieve (Grant Nos. 2016B010124009 and 2017B010124002), Guangdong Key Laboratory of Data Security and Privacy Preserving (Grant No. 2017B030301004), Guangzhou Key Laboratory of Data Security and Privacy Preserving (Grant No. 201705030004), Fundamental Research Funds for the Central Universities under Grant 21618329.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
[2] "Wikipedia. list of cryptocurrencies," "https://en.wikipedia.org/wiki/List_of_cryptocurrencies", [Online].
[3] H. A. Kalodner, M. Carlsten, P. Ellenbogen, J. Bonneau, and A. Narayanan, "An empirical study of namecoin and lessons for decentralized namespace design." in *WEIS*. Citeseer, 2015.
[4] M. Li, J. Weng, A. Yang, W. Lu, Y. Zhang, L. Hou, J.-N. Liu, Y. Xiang, and R. Deng, "Crowdbc: A blockchain-based decentralized framework for crowdsourcing," *IEEE Transactions on Parallel and Distributed Systems*, 2018.
[5] "Blockchain can legally authenticate evidence, chinese judge rules," "https://www.coindesk.com/blockchain-can-legally-authenticate-evidence-chinese-judge-rules", 2018, [Online].
[6] "Etherscan," "https://etherscan.io/contractsVerified", 2018, [Online].
[7] C. Riva and Y. Yang, "Generation of architectural documentation using xml," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002, pp. 161–169.
[8] D. R. Day and O. O. Fox, "Object oriented programming system with displayable natural language documentation through dual translation of program source code," Sep. 14 1999, uS Patent 5,953,526.
[9] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 43–52.
[10] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2016.
[11] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 625–636.
[12] R. S. M. J. F. Muneeb Ali, Jude Nelson, "Blockstack: A global naming and storage system secured by blockchains," in *USENIX Annual Technical Conference, USENIX ATC 2016*, Denver, CO, 2016, pp. 181–194.
[13] T. V. Asaph Azaria, Ariel Ekblaw, "Medrec: Using blockchain for medical data access and permission management," in *2nd International Conference on Open and Big Data, OBD 2016*, Vienna, Austria, Aug. 2016, pp. 25–30.
[14] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009.
[15] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[16] W. Gavin, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.
[17] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269.
[18] "Solidity documentation," "https://solidity.readthedocs.io/en/latest/solidity-in-depth.html", 2018, [Online].
[19] H. Theiling, "Extracting safe and precise control flow from binaries," in *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*. IEEE, 2000, pp. 23–30.
[20] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Component rank: relative significance rank for software component search," in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 14–24.
[21] C. D. Manning, C. D. Manning, and H. Schütze, *Foundations of statistical natural language processing*. MIT press, 1999.
[22] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.
[23] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sept 2016, pp. 625–636.
[24] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," 2018.
[25] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research*, vol. 12, no. Aug, pp. 2493–2537, 2011.
[26] E. Reiter and R. Dale, *Building natural language generation systems*. Cambridge university press, 2000.
[27] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 83–92.
[28] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 232–242.
[29] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.
[30] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust*. Springer, 2017, pp. 164–186.
[31] Y. Zhang, X. Lin, and C. Xu, "Blockchain-based secure data provenance for cloud storage," in *International Conference on Information and Communications Security*. Springer, 2018, pp. 3–19.