Smart Contracts Vulnerability Auditing With Multi-Semantics

Zhen Yang
Department of Computer Science
City University of Hong Kong
Hong Kong, China
zhyang8-c@my.cityu.edu.hk

Jacky Keung
Department of Computer Science
City University of Hong Kong
Hong Kong, China
Jacky.Keung@cityu.edu.hk

Miao Zhang
Department of Computer Science
City University of Hong Kong
Hong Kong, China
miazhang9-c@my.cityu.edu.hk

Yan Xiao
School of Computing
National University of Singapore
Singapore, Singapore
dcsxan@nus.edu.sg

Yangyang Huang
Department of Computer Science
City University of Hong Kong
Hong Kong, China
yhuan5@cityu.edu.hk

Tik Hui
Department of Computer Science
City University of Hong Kong
Hong Kong, China
tikhui@cityu.edu.hk

***Abstract*—Smart contracts vulnerability auditing is vitally critical to ensure transaction execution in normal on blockchain. The current data-driven approaches normally tokenize smart contracts into a series of sequences according to only one tokenization standard for vulnerability detection purpose, resulting some of the semantic contexts could not be reflected within restricted sequence length.**

To address this limitation, we generate sequences from smart contracts in three tokenization standards for which we utilize n-gram language model to capture semantic contexts respectively, and finally exploiting our effective combination strategy of Intersection or Union to integrate the audited results from multiple semantic contexts. In order to evaluate the proposed approach, we applied it on over 7200 Ethereum smart contract samples. Experimental result shows our proposed method is capable of detecting vulnerabilities and competitive with the baseline in test sets, with improved precision of over 44% when Intersection is applied in their results, as well as improved Recall measure up by over 300% and F-measure up by 220% when Union is applied. Our proposed method for smart contract vulnerability detection, an important tool for developing quality decentralized software applications, is able to analyze multiple semantic contexts and successfully detects more true vulnerabilities with high precision, outperforming that of the baseline approaches.

Keywords—Software Engineering, Smart Contract, Ethereum, N-gram Language Model, Vulnerability Auditing

I. INTRODUCTION

Smart contracts, as the backend logics of decentralized applications (DApp), are a series of automatically executed programs written by a Turing-complete programming language such as solidity, first applied by Ethereum in around 2013 [1]. They operate on top of the consensus protocol of the blockchain network for the purpose of ensuring the trading to complete between different parties without trust, by which people could remove the traditional central entities those managing the transaction network and operate their businesses directly peer-to-peer, thereby to simplify the transaction procedure. As one of the most popular blockchain platform, Ethereum has been growing to a huge volume: The total Ether supply and market capitalization on Ethereum have been up to 100 Million Ether and 18 Billion USD respectively. Nevertheless, due to the unfamiliarity with the smart contract development and its peculiar operation mechanism, a great amount of potential risks appeared in those DApps operating on the Ethereum such as the infamous DAO attack in 2016 leading 60 million USD loss [2-5]. As such, vulnerability auditing on smart contracts has been becoming a vitally critical research area. Current data-driven approaches such as the latest literature of S-gram [6], utilize N-gram Language Model (NLM) to analyze token sequences generated based on smart contract Abstract Syntax Tree (AST). Nevertheless, like most literatures using NLM in code defect prediction, the S-gram only adopted one kind of tokenization standard in sequences generation, leading to the ignorance of some semantic contexts within a restricted sequence length.

In this paper, we propose an improved model, namely Multi-Semantic gram (MSgram), based on the S-gram to further boost its performance by analyzing sequences of multiple semantics. Basically, for each smart contract, we generate sequences in three ways for the multi-semantic purpose: 1) Text sequence covering only text information, i.e. only focus on leaf nodes in AST when tokenization 2) Structure sequences covering only structure information, i.e. only focus on non-leaf nodes in AST when tokenization 3) Combined sequences covering both structure and text information of a smart contract, i.e. a completely AST representation. For the part of constructing the sequences of structure, text and combined version, we modified the SBT Traversal proposed by Hu et al. [7] to make it more general in sequence extraction and fused the characteristic of type-based tokenization of S-gram, thereby to make the *tokenizer* focus mainly on Ethereum smart contracts.

Since each kind of sequence represent a kind of semantic, for their different tokens combinations and permutations, thus the analysis on different kind of sequence will capture different information. As such, we feed each kind of sequence into their corresponding trained NLMs respectively to capture their individual semantic and thereby to obtain separate audited results. For each smart contract, we can get three audited results

978-1-7281-7303-0/20/$31.00 ©2020 IEEE
DOI 10.1109/COMPSAC48688.2020.0-153

from above analysis of three kinds of sequences. After that, we adopt a combination strategy of Intersection or Union on these three separate audited results to integrate a multi-semantic analyzed audited result, i.e. the final audited result for assisting avoiding vulnerabilities during the development of the smart contracts.

We evaluate our proposed approach on 7200 Ethereum smart contracts gathered from Etherscan.io [8] with division into train set and test set. Furthermore, the whole evaluation process is based on two phases of model construction and security auditing. In model construction phase, we optimize separate NLMs and assess combination strategies to obtain the best setting of MSgram based on train set. Then in the auditing security phase, we build the MSgram with the pre-configured setting in the construction phase to make a comparison with the baseline based on the test set.

The result shows final audited results with high precision up to 51.79% (averagely 48.02%) in test set when we made an Intersection combination. While with a Union combination adopted to capture more true vulnerabilities, we obtain another final audited result with recall up to 66.02% (averagely 58.57%) and F-measure up to 0.4211 (averagely 0.3406) in test set. Compared with the performance of the baseline i.e. original S-gram model in test set, the final audited result with Intersection strategy can improve the precision by up to 106.59% (averagely 44.24%). For the auditing result with Union strategy, it can improve the recall by 360.55% at most (averagely 306.50%) and the F-measure by 271.02% at most (averagely 224.22%).

In summary, the contributions of this paper include the following three-fold:

(1) We propose a novel concept of integrating multiple semantics in smart contract vulnerability auditing with N-gram Language Model.

(2) We put forward three tokenization standards by modifying the SBT Traversal and generate three kinds of sequences based on structure only, text only and combined version in order to capture multi-semantics.

(3) We explore the combination strategies (Intersection and Union strategies) of multi-semantics and empirically study the performance and their pros and cons.

II. BACKGROUND

A. Ethereum Smart Contracts and Vulnerabilities

Ethereum Smart contracts consist of a series of state variables and functions which can be invoked by Ethereum externally own accounts for transaction operations. Most of those state variables store digital assets or currency possessed by users and those functions manage the trading processes or regulations predefined by the contract deployer. Due to the unfamiliarity with the smart contract development and its peculiar operation mechanism, a lot of security risks in capital management and trading continuously come out.

For instance, the reentrancy risk leading the DAO attack in 2016 is caused by a malicious attacker try a recursive manner to invoke a profitable function of the targeting contract repeatedly via his attack contract. Other instances like timestamp dependency risk: Some of the critical operations depending on timestamp could have risk of getting attacked by malicious attackers who change the block's timestamp illegally to make profits. Vulnerabilities in smart contract as mentioned above won't crash the code execution but can destroy the normal transaction logics predefined by contract deployers, thereby causes huge losses [3, 9, 10].

B. N-gram Language Model

In this paper, we adopt the N-gram Language Model (NLM) to analyze those token sequences generated from smart contracts. The core methodology of the NLM is derived from Markov chain which is essentially a Bayesian conditional probability model. It uses the continuous former n words to estimate the probability of the current word t_i, i.e. $P(t_i|t_{i-(n-1)},\dots t_{i-1})$, as shown in Equation (1). Thus, given a sentence, e.g. $S=t_1t_2t_3t_4\dots t_k$, its probability is shown in Equation (2).

$$P\left(t_i\middle|t_{i-(n-1)},\dots t_{i-1}\right)=\frac{count\left(t_{i-(n-1)},\dots t_{i-1},t_i\right)}{count\left(t_{i-(n-1)},\dots t_{i-1}\right)} \quad (1)$$

$$P(S)=\prod_{i=1}^{k}P\left(t_i\middle|t_{i-(n-1)},\dots t_{i-1}\right) \quad (2)$$

After we train the model (i.e. we calculate each word's probability of n-gram in the whole corpus), similarly with S-gram, we also adopt the log-transformed perplexity metric to measure whether a token sequence is a potential vulnerability or not, because the form of addition can speed up the calculation while avoiding the floating-point number overflows downward caused by the too small probability after product. *In the field of natural language process, a sentence with a higher perplexity score means its word sequence is scarcer, in other words, it tends to be an abnormal or incorrect sentence.* Since vulnerable sequences occupy only a little part in the whole dataset, if a token sequence is scarce, it has a great chance to be a vulnerable sequence. Therefore, for those sequences with high perplexity scores, we put them into candidate vulnerability list. The perplexity log-transformed score for a whole sentence S with the length of K is represented by $H(S)$ like below:

$$H(S)=-\frac{1}{K}\sum_{i=1}^{K}log\ P(t_i|t_{i-(n-1)}\dots t_{i-1}) \quad (3)$$

III. METHODOLOGY

A. Model Framework Overview

The framework of the proposed MSgram includes two phases, i.e. the model construction phase and security auditing phase, respectively. Figure 1 shows the overview of MSgram framework.

In the model construction phase, **the first step is Tokenization**. We propose three tokenization standards and adopt our *tokenizer* to extract each smart contract source code in the train set into three sequence collections of Combined Sequence, Text Sequence and Structure Sequence respectively. **The next step is model training and optimization.** We feed each collection into the NLM tool kenlm [17] to train their models and optimize them by different parameters and thresholds. Then we select those optimized NLMs based on the precision of their *separate audited result*, we named the optimized NLM as *Detector*. **The final step is designing and assessing combination strategies.** We provide two kinds of

combination strategies, i.e. Intersection and Union to improve the performance of those single *Detectors* and thereby generate a final audited result. The verified best parameter and threshold combination for each single *Detector*, as well as the effective combination strategy will be applied in the security auditing phase.

For the phase of security auditing, when an auditing target smart contract comes, MSgram firstly enable the tokenizer to extract it into three kinds of sequence collections then analyze token sequences by their corresponding *Detectors* respectively to output the separate audited results and finally automatically follow the combination strategy predefined in the construction stage to get the final audited results.

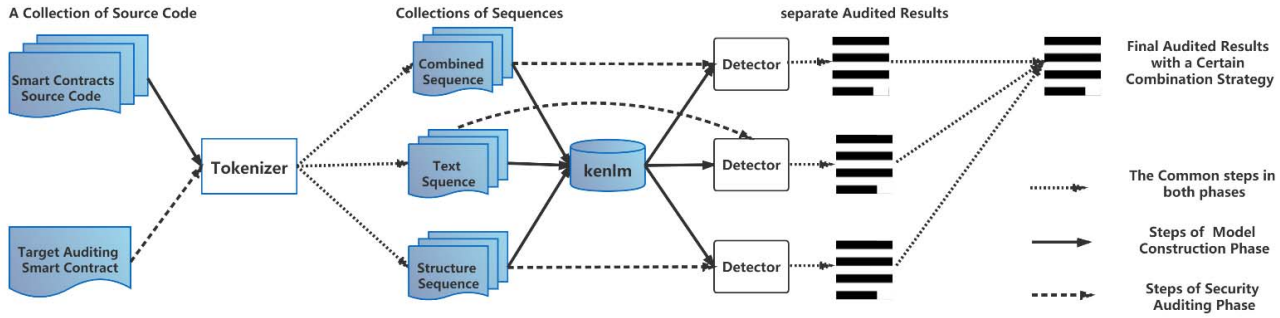

Fig. 1. The Framework Overview of MSgram

B. Tokenization

In this paper, we modify SBT Traversal to be more generalized in sequence extraction. We also add characteristics of the tokenization method in S-gram to create our *tokenizer*, thereby to generate three kinds of sequences, which is the first step of MSgram. SBT Traversal mechanism essentially uses a series of brackets to reserve code's structure information to make its sequences recoverable to AST and transforms each node of AST into tokens for constructing sequences. Set the following contract function `sample` as a simple example and the Figure 2 shows the AST of this function which is used for transferring `a` Ehters to the account address `recv`:

```
function sample (uint a, address recv) public {
                recv.transfer(a);
                }
```

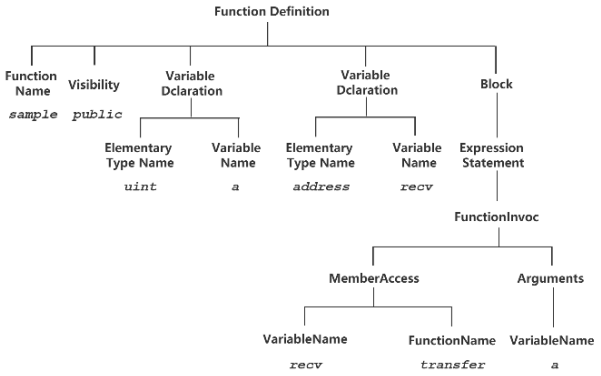

Fig. 2. The AST of function sample

More specifically, since leaf nodes in AST are composed by *type* and *value*, while non-leaf nodes are composed by *type* only. The original SBT Traversal uses four tokens: "(", "*type* / *type-value*", ")" and " *type* / *type-value*" to represent a non-leaf / leaf node in AST, as well as arrange tokens in a nested manner to represent code structure [7], while actually, in NLMs, we cannot distinguish different brackets of different nodes by this way for every bracket with the same probability in NLM, causing us lose the structure range information of different nodes. On the second hand, too many insignificant tokens increase the perplexity of normal sequences with different degrees, which increases the model's false positive rate. Besides, former literature has studied adequately and testified that researchers should tokenize the source code in a relatively high level to detect vulnerability on semantic, otherwise the NLM could only detect syntactic errors, which is the work of compilers [11]. Therefore, we modify the SBT Traversal basically in the following way: 1) For non-leaf nodes denoting structures such as FunctionDef, ExpressionStatement and etc. We express them by two tokens: "(*type*" and " *type*)" so that the language model can differentiate structures' range from themselves with their corresponding probabilities concisely. 2) For those leaf nodes without range information needs to be kept such as ElementaryTypeName-uint, VariableName-a, etc. We express them by one token: "*type-value*", thereby to drop insignificant tokens but guarantees their integrity of necessary information. 3) We also add characteristics of the tokenization standard S-gram used, such as add a "call_min_gas" token when using `transfer` function, to make our *tokenizer* focus mainly on Ethereum smart contracts. Then after processed by tokenization, the abovementioned code snippet can be extracted into a combined sequence like below, we adopt different indentations to show its structure intuitively:

Combined Sequence:

(*FunctionDef*
FunctionName-sample
Visibility-public
(*VariableDeclaration*
ElementaryTypeName-uint VariableName-a
VariableDeclaration)
(*VariableDeclaration*

ElementaryTypeName-uint VariableName-a
VariableDeclaration)
(*Block*
(*ExpressionStatement*
(*FunctionInvoc*
(*MemberAccess*
VariableName-recv FunctionName-transfer
call_min_gas
MemberAccess)
(*Arguments VariableName-a Arguments*)
FunctionInvoc)
ExpressionStatement)
Block)
FunctionDef)

From above sequence, we can recover it back to AST, since the structure information is reserved, at the same time, each leaf-node hold their *type* and their *value* to reserve their text information. For structure sequence, we delete all the leaf nodes, which means we extract it in a much higher semantic level and only keep the code structure information. For text sequence, we keep those leaf nodes as well as specifier tokens to denote a certain intention's start, such as FunctionSpecifier-function, which means a start of a function. By the basic rules above, we create a *tokenizer* by the tool [12] solidity-parser-antlr to generate three kinds of sequences for the whole data set. For the code snippet above, its structure and text sequence are shown below in a similar way respectively:

Structure Sequence:

(*FunctionDef*
(*VariableDeclaration VariableDeclaration*)
(*VariableDeclaration VariableDeclaration*)
(*Block*
(*ExpressionStatement*
(*FunctionInvoc*
(*MemberAccess MemberAccess*)
(*Arguments Arguments*)
FunctionInvoc)
ExpressionStatement)
Block)
FunctionDef)

Text Sequence:

FunctionSpecifier-function FunctionName-sample Visibility-public ElementaryTypeName-uint VariableName-a ElementaryTypeName-address VariableName-recv VariableName-recv FunctionName-transfer call_min_gas VariableName-a

C. Model Parameters and Thresholds

This section introduces the parameters (i.e. gram number, sequence length and minimum token occurrence) and thresholds (i.e. minimum perplexity and top flagged vulnerability amount) used in model training and optimization. Different from other literature adopting NLM [6, 11, 13, 14], we add another new threshold which is the minimum perplexity score to label the potential vulnerabilities together with the top flagged vulnerability amount, thereby to make the candidate vulnerability list in audited result dynamic with different NLMs.

- Gram Number: An n-gram language model with different *n* will mark different perplexity score for a same sequence. Due to the former study [6] has verified that when the n larger than 5 the performance of NLM decreased in smart contracts vulnerability auditing, thus we evaluate *n* with 2-6 to find the most appropriate one in our experiment.
- Sequence Length: Sequence length represents the number of tokens in a sequence. If we only detect the whole sequence of a contract, we cannot accurately localize the vulnerable row or position. Thus researchers study vulnerability auditing using NLM normally break the whole sequence into snippets. Nevertheless, since different snippets may reflect different scenarios either vulnerable or not, different length of sequences also affect the performance in vulnerability auditing. In our experiment, we set the sequence length as 10,20,30,40,50 respectively to observe their impacts. Since the experiment empirically testify that the precision of the separate audited result increases steadily with the increment of sequence length, where we cannot optimize the NLM by this parameter, instead we evaluate the MSgram under each fixed sequence length.
- Minimum Perplexity: The minimum perplexity is one of the thresholds used in constructing candidate vulnerability list and those sequences with higher perplexity score than predefined minimum perplexity score would be labeled as potential vulnerable sequence. Since same contracts with different NLMs have different perplexity range and distribution for their sequences snippets (e.g. some NLMs may mark the perplexity score of a collection of sequences in a range of 1-20, while others may mark them in a range of 100-10000), setting a same minimum perplexity for them when implementing the vulnerability auditing would lead to an unfair comparison among NLMs. Besides even if we normalize them into a fixed range such as 0-1, their distribution is still different and without any change. In this case, after each collection of sequences is fed into their corresponding trained NLMs and obtain their collections of perplexity scores, we all evenly set three minimum perplexities within the range of the minimum and the median of their entire perplexity scores respectively to ensure we can capture more than half of those true vulnerabilities every time under this threshold and try to keep fair comparison in different NLMs.
- Top Flagged Vulnerability Amount: The top flagged vulnerability amount is another threshold used in constructing candidate vulnerability list. After we select a series of sequences with higher perplexity than the predefined minimum perplexity score, we rank them in descending order and only capture those sequences within the range of the top flagged vulnerability amount to further filter the candidate vulnerability list. For each collection of perplexity score, we select six numeric values below or around the mean of this collection of perplexity score to ensure we have the chance of capturing all vulnerable sequences of each smart contract under this threshold and try to keep a fair comparison as well.

- Minimum Token Occurrence: Setting the minimum token occurrence when building NLM can generalize the model to avoid incorrectly auditing for some non-vulnerable sequence. In our experiment, we convert all the tokens those only appear three times to <UNK> as the another parameter of NLM, which is a testified practice in NLP study [15].

D. Combination Strategies

After recording the parameter and threshold combinations of optimized NLMs of each sequence type under each sequence length, we start to combine their separate audited results, which is the last step of MSgram. In this section, we try two general kinds of strategies for combination. Each strategy includes two steps for exploration, **the first step is combining by sequence type, i.e. integrating audited results based on different semantics, while the second step is a further pruning by sequence length to try to further prune false positives**. The following part we firstly introduce the step 1 of each strategy respectively, then introduce the step 2 of them in the last paragraph of this section.

The first strategy is Intersection. An obvious principle is if a sequence or its sub-sequence can be detected as vulnerable by more than one *Detector* of different sequence types, it has a much more opportunity to be a true vulnerability. Since the sequence length represents the number of tokens in a sequence, thus even with same sequence length, the range of sequences in a contract of different sequence types are various for different semantic levels. In this case, we cannot simply combine the separate audited results. Instead, we adopt those sub-sequences (i.e. sequences within the range of its long sequences in smart contracts) to prune their long sequences in corresponding audited results. Furthermore, the range of structure sequences is longer than that of text sequences and combined sequences under same sequence length in a same contract, which means text sequences and combined sequences are always sub-sequences of structure sequences. Thus, as the 1st step of Intersection, we adopted that if a sequence is confirmed as vulnerable in the Structure Sequence[1] and its sub-sequence in Combined Sequence or Text Sequence got vulnerable confirmed as well, then this sequence is labeled as vulnerable. More formally, we get a new candidate vulnerable set of audited result for the smart contract i by the following formula $S_z(i)$:

$$S_z(i) = \bigcup_{\forall z,j,k \in i} \begin{matrix} AR_{Struct}(z,\theta_{Struct}) = TRUE \ \wedge \\ ((AR_{Text}(j,\theta_{Text}) = TRUE \ \wedge \ Range(j) \subseteq Range(z)) \\ \vee (AR_{Comb}(k,\theta_{Comb}) = TRUE \ \wedge \ Range(k) \subseteq Range(z))) \end{matrix} \quad (4)$$

The $AR_{Struct}(z,\theta_{Struct})$ represent the audited result for structure sequence z in contract i by the optimized NLMs with best parameter and threshold combination θ_{Struct}, so on so forth for $AR_{Text}(j,\theta_{Text})$ and $AR_{Comb}(k,\theta_{Comb})$. $Range\ (z)$ represents the range of sequence z, so on so forth for $Range(j)$ and $Range(k)$. $\cup$ represents the union of each sequence z satisfied the formula 4.

The second strategy is Union. Since those sequences are generated from different tokenization standards, leading them to capture different semantics by different token arrangements and semantic levels. Therefore, if we take a union way to combine their audited results, it should capture more true vulnerabilities than models that only apply one tokenization. In this case, we use the audited results of the other two kinds of sequences to trim the false negatives for the audited results of the structure sequence in a union way, which is the 1st step of Union strategy. Specifically, in addition to keeping sequences labeled as vulnerable in audited results for Structure Sequence, once a sub-sequence of a structure sequence is identified as vulnerable in Text Sequence or Combined Sequence, we label this structure sequence as a vulnerable. More formally, we obtain another new candidate vulnerable set of audited result for the smart contract i by the following formula $S_z(i)$:

$$S_z(i) = \bigcup_{\forall z,j,k \in i} \begin{matrix} AR_{Struct}(z,\theta_{Struct}) = TRUE \ \vee \\ (AR_{Text}(j,\theta_{Text}) = TRUE \ \wedge \ Range(j) \subseteq Range(z)) \\ \vee (AR_{Comb}(k,\theta_{Comb}) = TRUE \ \wedge \ Range(k) \subseteq Range(z)) \end{matrix} \quad (5)$$

Here we introduce the 2nd step of each strategy. When comes to this step, all audited results have combined based on the structure type. Thus, in this step, we further prune false positives by different sequence length, which means the 2nd step is the same in both strategies. Basically, we use the audited result of previous smaller sequence length to prune the false positives in that of current sequence length, e.g. using audited result of sequence length 10 to prune the false positives in that of sequence length 20. The rationale of 2nd step is if a vulnerability gets confirmed in multiple sequences of different lengths, it has a great chance to be a true vulnerability. For instance, in audited results of same sequence type but different sequence length, if a sequence *BC* is labeled with vulnerable in an audited result of sequence length 2 and in another audited result of sequence length 3, a sequence *ABC* is also labeled with vulnerable. Since *BC* is a sub-sequence of *ABC*, there's a great chance that a vulnerability exists in *ABC*, thus we labeled *ABC* with vulnerable, and otherwise it should be labeled with non-vulnerable to reduce false positives. More formally, we obtain the final candidate vulnerable set of the audited result for the smart contract i by the following formula (6). $AR_{SL+10}(z)$ represents the audited result of structure sequence z in contract i under sequence length $sl+10$ (for our interval is 10 for each sequence length) processed by 2nd step, so on so forth for $AR_{SL}(j)$. But the performance of the above strategies' application and combination need further empirical verification, which will be studied in Section IV.

$$S_z(i) = \bigcup_{\forall z,j \in i} \begin{matrix} AR_{SL+10}(z) = TRUE \ \wedge \\ (AR_{SL}(j) = TRUE \ \wedge Range(j) \subseteq Range(z)) \end{matrix} \quad (6)$$

IV. Experiments

A. Experimental Design and Evaluation Metrics

The whole experiments mainly follow the steps discussed in Section III, including the data collection, NLM optimization, combination strategy exploration and the comparison with baseline. The baseline paper S-gram utilize accuracy as the evaluation metrics, which is actually unfair and meaningless. Because vulnerable sequences only occupy an extremely little part in the whole sequence collection, which means even the vulnerability detector can only detect very few true vulnerabilities, its performance in accuracy still can be very high. Therefore, the accuracy metric cannot reliably reflect the true performance of the model. **As such, in our experiment, we adopt the precision, recall and F-measure to assess the model's performance on the whole train set and test set.**

1. We adopt the "Combined Sequence" with initial capital letters to represent the collection of combined sequences, which is also applicable to "Text Sequence" and "Structure Sequence".

Precision measures a model's performance of detecting true vulnerabilities among labeled vulnerabilities while recall measures a model's performance of detecting true vulnerabilities among the whole true vulnerabilities. And F-measure assesses the model's performance based on both metrics above. Only when the model's performance in precision and recall are both high, its F-measure will be high.

B. Dataset Collection and Construction

To make a fair comparison with the original Sgram, we retrieve all smart contracts used in the original S-gram from Etherscan.io [8], and adopt the Oyente [16] to confirm vulnerabilities for those contracts and generate test reports, which follows the same way applied in the literature of S-gram. However, we find there're a lot of same smart contracts but with different contract addresses, which is caused by deploying same contracts more than one time on Etherscan.io. This circumstance might decrease the perplexity of both normal and defective sequences of repeatedly deployed contracts, causing the experimental results imprecise. Therefore, we delete all the duplicated contracts and finally we obtain totally over 7200 smart contracts as our whole dataset. We divide the whole dataset into two parts of trainset (90%, around 6.5 thousand smart contracts) and test set (10%, around 700 smart contracts). The whole data set has been published in github[2], including source code and test reports that record location and type of vulnerabilities.

C. Experiment Devices

Our experiments are conducted on a Windows10 laptop with 6 cores 12 threads of Intel Core 2.20GHz i7-8750H CPU, 16GB RAM and 512GB SSD.

D. Optimization for the NLM

In this part, we adjust those parameter and threshold combinations discussed in the section III and investigated their influence on each single NLM. In previous studies [3, 9 ,13], researchers tend to fix the sequence length and top flagged vulnerability firstly to find a gram number with highest precision or maximum average true vulnerabilities. And by this way of adjusting one parameter while fixing others to further adjust the rest of parameters. However, this kind of greedy optimization method cannot find the best parameter and threshold combination, since it has no chance of testifying every combination of those parameters and thresholds. Thus, in our experiments, we try to exploit all the parameters and thresholds together to form as many as different combinations as we can for each single NLM's construction to find the best combination.

During the whole experiments, we try different combinations on those parameters and thresholds and constructed 5*5*3*6*3=1350 models in total for comparison on the precision of the whole train set. Then the model with the highest precision under each sequence length in each sequence type is selected as the optimized NLM. Table I presents parameter and threshold combination of each optimized NLM for three kinds of sequence types under different sequence length. Table II presents their performances in precision, recall and F-measure. Obviously, regardless of sequence types, the highest value of each metric almost appears when auditing longest sequences. On the other hand, by horizontally comparison, the performance of NLMs for Text Sequence are always the best under each sequence length, which means this kind of tokenization standard captures more useful information in vulnerability detection.

We also present Figure 3 to show their performance variations among five sequence lengths clearly. According to Figure 3, we find that, for each sequence type, with the increment of sequence length, each of the evaluation metric is generally in ascend order, especially the precision, which is also demonstrated in the literature of S-gram [6]. It is because the increment of sequence length leads to the overlap of vulnerable snippets, which means a sequence may consist of more vulnerabilities, causing those vulnerabilities easier to be detected to some extent. Nevertheless, a very long sequence is useless in localizing vulnerability in a small range, thus developers should refer to the actual situation to select a proper sequence length and we cannot optimize NLM on it. As such, in later comparison with baseline, we compare the performance under each sequence length, which follows the same way used in S-gram paper.

TABLE I. THE PARAMETER AND THRESHOLD COMBINATION OF EACH OPTIMIZED NLM FOR THREE KINDS OF SEQUENCE TYPES UNDER DIFFERENT SEQUENCE LENGTH

	Optimized NLM For Combined Sequence			**Optimized NLM For Text Sequence**			**Optimized NLM For Structure Sequence**		
Sequence Length	***Gram Number***	***Minimum Perplexity***	***Top Flagged Vulnerability Amount***	***Gram Number***	***Minimum Perplexity***	***Top Flagged Vulnerability Amount***	***Gram Number***	***Minimum Perplexity***	***Top Flagged Vulnerability Amount***
10	3	100	30	2	70	40	6	30	20
20	2	20	60	2	30	60	3	5	20
30	2	7	130	3	7	80	3	3	40
40	5	7	180	2	30	130	6	4	50
50	5	5	200	3	7	150	4	4	150

TABLE II. THE PERFORMANCE OF EACH OPTIMIZED NLM FOR THREE KINDS OF SEQUENCE TYPES UNDER DIFFERENT SEQUENCE LENGTH

	Optimized NLM For Combined Sequence			**Optimized NLM For Text Sequence**			**Optimized NLM For Structure Sequence**		
Sequence Length	***Precision***	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***
10	0.1050	0.1172	0.0580	0.1743	0.1850	0.1240	0.1784	0.1443	0.0976
20	0.1532	0.1614	0.1038	0.2853	0.2473	0.1958	0.2724	0.1379	0.1040
30	0.2112	**0.2881**	0.1923	0.3534	0.3009	0.2417	0.3201	0.2018	0.1683
40	0.2796	0.2389	0.1776	0.3954	**0.3206**	0.2655	0.3829	0.2022	0.1673
50	**0.3105**	0.2816	**0.2172**	**0.4318**	0.3121	**0.2680**	**0.4198**	**0.3571**	**0.3119**

2. https://github.com/yz1019117968/MSgramDataset.git

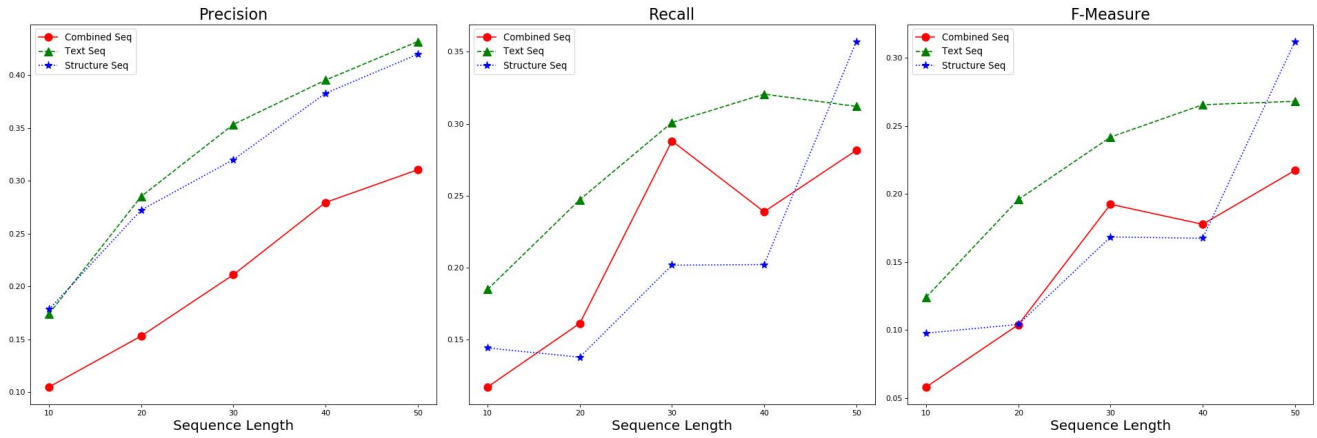

Fig. 3. The Performance of Each Optimized NLM for Three Kinds of Sequence Types Under Different Sequence Lengths

E. Experiments on Combination Strategies

In this section, we conduct a series of experiments on the two combination strategies proposed in section III to explore the way of exploiting the audited results from multi-semantics.

For the Intersection strategy, we focus on its improvement of precision. After implemented the combination by sequence type and pruning by sequence length, we present the Table III to demonstrate the precision of audited results under each sequence length before and after applying Intersection strategy. Since the strategy is applied based on audited result of Structure Sequence, we set its precision as an original reference. Figure 4 shows the improvements of two steps under Intersection strategy towards the original precision and the precision after 1st step respectively, as well as this strategy's final improvement towards the original precision. Each box represents the improvements on precision under five sequence lengths respectively, the green dotted line represents the mean improvement among five sequence lengths. It's obvious that the improvement by the 1st step is small compared with that of the 2nd step. Based on the Table IV which shows the Figure 4's information by accurate numbers, we find that the improvement by the 1st step up to 15.53%, averagely 8.23% but the improvement by the 2nd step can up to 51.60%, averagely 20.50%. The final precision of audited results under each sequence length achieves improvement by up to 69.08%, averagely 26.08% with respect to the original precision. Nevertheless, their recall and F-measure decline, which is caused by a small part of the incorrect removal of true positives during pruning. Due to this strategy is used for boosting the detecting precision, the little decline in recall and F-measure can be tolerated. *Therefore, we keep this strategy and adopt it in security auditing stage.*

TABLE III. THE PRECISION BEFORE AND AFTER THE INTERSECTION STRATEGY UNDER EACH SEQUENCE LENGTH

Sequence Length	**Original**	**After the Intersection Strategy**	
		1st Step: Combined by Sequence Type	***2nd Step: Pruning by Sequence Length***
10	0.1784	0.2061	0.2061
20	0.2724	0.3038	0.4606
30	0.3201	0.3416	0.3731
40	0.3829	0.3948	0.4638
50	0.4198	0.4376	0.4538
Average	0.3147	0.3368	0.3915

TABLE IV. THE IMPROVEMENT ON PRECISION AFTER THE INTERSECTION STRATEGY UNDER EACH SEQUENCE LENGTH

Sequence Length	**Improvement of Intersection Strategy**		**Final Improvement**
	1st Step: Combined by Sequence Type	***2nd Step: Pruning by Sequence Length***	
10	**15.53%**	/	15.53%
20	11.53%	**51.60%**	**69.08%**
30	6.74%	9.20%	16.56%
40	3.11%	17.48%	21.14%
50	4.23%	3.71%	8.10%
Average	8.23%	20.50%	26.08%

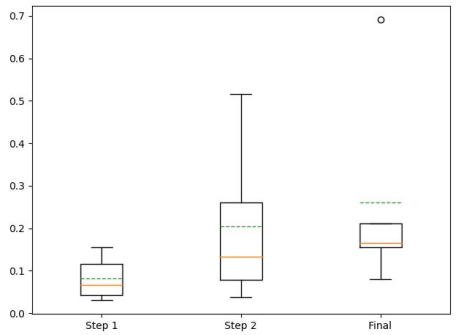

Fig. 4. The Improvements on Precision of the Step 1, Step 2 Under Intersection Strategy and this Strategy's Final Improvement

For the Union strategy, we mainly focus on the final result of whether it can find more vulnerabilities. Similarly, after implemented the combination by sequence type and pruning by sequence length, we present the Table V to demonstrate the performance on precision, recall and F-measure before and after applying Union strategy for audited results under each sequence length. Similarly, we adopt the performance of audited results for Structure Sequence as the original reference to show the improvements. Figure 5 adopts three sub-figures to demonstrate, for each metric, the improvements in step 1, step 2 under the Union strategy towards the original precision and the precision after 1st step respectively, as well as this strategy's final improvement towards the original precision. For every sub-figure, each box represents the five improvements on its metric under five sequence lengths respectively, the green dotted line

represents the mean improvement on this metric among five sequence lengths. It's obvious that the 1st step's improvements on recall and F-measure are great but 2nd step's are weak and even negative. However, their improvements on precision are both weak. More specifically, Table VI shows the Figure 5's information by accurate numbers, from which we find that, after 1st step, the recall achieves improvements by up to 279.38%, averagely 210.08% and the F-measure by up to 182.97%, averagely 119.48% after the combination by sequence type with comparison to the original performance, although the precision declines a little of 5.63% on average. Nevertheless, after the 2nd step, the precision averagely improves only by 1.79% while the recall and F-measure both decline averagely by 14.73% and 8.63%, which isn't worth for pruning. *As such, in the later security auditing stage, for the strategy of Union, we only adopt the 1st step of combination by sequence type and drop the 2nd step.*

F. Comparison with the Baseline

After the models are trained and the combination strategies are configured, we start to adopt them on test set to get an unbiased estimation for the performance of the MSgram and make a comparison with the baseline. Since the paper of S-gram did not publish its source code, we tried our best to re-implement one for the comparison. The following Table VII shows the performance of MSgram with both combination strategies and of the baseline we re-implemented. Table VIII presents the improvement of MSgram with respect to the baseline. Figure 6 reflect the content of Table VII and Table VIII intuitively. It's obvious that, in test set, the MSgram's precision in vulnerability detecting can be up to 51.16% in sequence length 40 under the Intersection strategy while, under the Union strategy, the precision, recall and F-measure can be up to 38.97% in sequence length 50, 66.02% in sequence length 30 and 42.11% in sequence length 50 respectively. Then compared with S-gram, the Intersection audited result of MSgram can improve the precision by up to 106.59%, averagely 44.24%. Under sequence length 30 and 50, the improvement appears in all three metrics. While the Union audited result of MSgram, although averagely 2.26% decline in precision, can improve the recall by 360.55% at most, averagely 306.50% and the F-measure by 271.02% at most, averagely 224.22%. More specifically, under sequence length 20, 30, 40, MSgram outperforms the baseline in all three measures. Thus, both the Intersection and the Union strategy can obtain greatly improvements on their focusing measures, as well as keep stability or even excel on other measures. As such, the MSgram outperformed the baseline on a great extent.

TABLE V. THE PRECISION, RECALL AND F-MEASURE BEFORE AND AFTER THE UNION STRATEGY UNDER DIFFERENT SEQUENCE LENGTH

	Original			**Performance After the Union Strategy**					
				1st Step: Combined by Sequence Type			***2nd Step: Pruning by Sequence Length***		
Sequence Length	***Precision***	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***
10	0.1784	0.1443	0.0976	0.1499	0.5124	0.2019	0.1499	0.512	0.2019
20	0.2724	0.1379	0.1040	0.2550	0.5232	0.2943	0.2547	0.496	0.2862
30	0.3201	0.2018	0.1683	0.3186	0.6498	0.3719	0.3297	0.596	0.3679
40	0.3829	0.2022	0.1673	0.3730	0.6120	0.4030	0.3775	0.554	0.3824
50	0.4198	0.3571	0.3119	0.4082	0.6838	0.4548	0.4187	0.438	0.3384
Average	0.3147	0.2087	0.1698	0.3009	0.5962	0.3452	0.3061	0.519	0.3154

TABLE VI. THE IMPROVEMENT ON PRECISION, RECALL AND F-MEASURE AFTER THE UNION STRATEGY UNDER DIFFERENT SEQUENCE LENGTH

	Improvement on Precision, Recall and F-Measure After the Union Strategy						**Final Improvement**		
	1st Step: Combined by Sequence Type			***2nd Step: Pruning by Sequence Length***					
Sequence Length	***Precision***	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***
10	-16.00%	254.97%	106.78%	/	/	/	-16.00%	254.97%	106.78%
20	-6.40%	**279.38%**	**182.97%**	-0.10%	-5.10%	-2.80%	-6.40%	**279.38%**	**182.97%**
30	-0.50%	221.94%	120.92%	3.49%	-8.30%	-1.10%	-0.50%	221.94%	120.92%
40	-2.60%	202.64%	140.93%	1.20%	-9.60%	-5.10%	-2.60%	202.64%	140.93%
50	-2.80%	91.46%	45.81%	2.56%	-36.00%	-25.60%	-2.80%	91.46%	45.81%
Average	-5.60%	210.08%	119.48%	1.79%	-14.70%	-8.60%	-5.60%	210.08%	119.48%

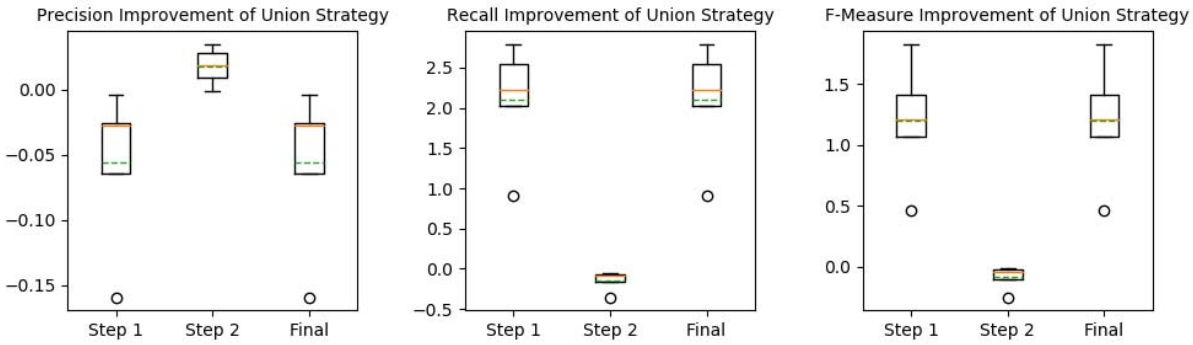

Fig. 5. The Improvement on Precision,Recall, F-Measure of the Step 1, Step 2 Under Union Strategy and this Strategy's Final Improvement

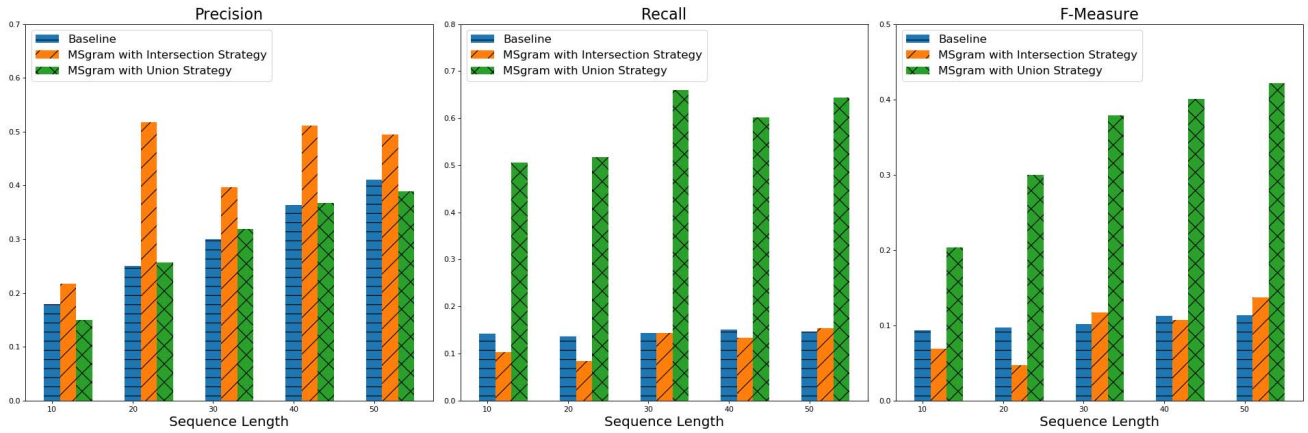

Fig. 6. The Performance of Baseline and MSgram

TABLE VII. THE PERFORMANCE OF BASELINE AND MSGRAM UNDER DIFFERENT SEQUENCE LENGTH

	Baseline					MSgram in Intersection Strategy					MSgram in Union Strategy				
Sequence Length	***10***	***20***	***30***	***40***	***50***	***10***	***20***	***30***	***40***	***50***	***10***	***20***	***30***	***40***	***50***
Precision	0.1791	0.2507	0.3006	0.3633	0.4112	0.2176	**0.5179**	0.3972	0.5116	0.4942	0.1501	0.2571	0.3196	0.3679	**0.3897**
Recall	0.1423	0.1366	0.1433	0.1510	0.1465	0.1036	0.0844	0.1441	0.1336	**0.1540**	0.5051	0.5175	**0.6602**	0.6020	0.6439
F-Measure	0.0937	0.0972	0.1020	0.1129	0.1138	0.0691	0.0469	0.1173	0.1075	**0.1374**	0.2030	0.3001	0.3785	0.4004	**0.4211**

TABLE VIII. THE IMPROVEMENT OF THE MSGRAM UNDER DIFFERENT SEQUENCE LENGTH

	Improved by MSgram in Intersection Strategy						Improved by MSgram in Union Strategy					
Sequence Length	***10***	***20***	***30***	***40***	***50***	***Average***	***10***	***20***	***30***	***40***	***50***	***Average***
Precision	21.5%	**106.59%**	32.11%	40.81%	20.19%	44.24%	-16.20%	2.56%	**6.31%**	1.27%	-5.23%	-2.26%
Recall	-27.2%	-38.21%	0.53%	-11.52%	**5.09%**	-14.26%	255.05%	278.75%	**360.55%**	298.68%	339.48%	306.50%
F-Measure	-26.3%	-51.78%	14.94%	-4.80%	**20.74%**	-9.43%	116.54%	208.71%	**271.02%**	254.71%	270.14%	224.22%

V. THREATS TO VALIDITY

Our paper includes the following threats to validity:

The Re-implementation for the baseline. Since the source code of S-gram was not published, we tried our best to follow its every experimental detail and re-implemented one for comparison with our improved MSgram model. The re-implemented S-gram model presents a higher precision than the one in its paper but with a lower accuracy. One possible reason is the smart contracts the S-gram paper used includes a lot of duplicates (we pointed out in Section IV-B), which improves the probability of both vulnerable and invulnerable sequences. At the same time, due to the invulnerable sequences among those should occupy a great part, causing the relatively high accuracy of the S-gram, even if its precision is low.

Vulnerabilities labeled by a symbolic execution tool. We followed the work of S-gram to use a symbolic execution tool named Oyente to confirm vulnerabilities. According to the paper of Oyente, it has a low false positive rate of only 6.3%, which means the dataset still have a small part of sequences mislabeled. Even if it is labeled by humans manually, mislabeled data cannot be avoided either. But actually, on the other hand, using the same data improves comparability with baseline.

VI. RELATED WORK

Smart contract vulnerability auditing. Since the vulnerable nature of smart contracts, a lot of researchers have been devoting themselves into this fields and proposed various attempts [18]. Wu H et al. [19] adopted the mutation testing in smart contract testing. Luu et al. [16] proposed a symbolic execution technique named Oyente to detect contracts' vulnerabilities based on several predefined patterns. Jiang B et al. [10] put forward a tool, namely ContractFuzzer, with Application Binary Interface (ABI) specification based fuzzing input relying on predefined test oracles. Similarly, Liu et al. [20] also adopted fuzzing techniques and introduced ReGuard but only targeted reentrancy risk of smart contracts. The method based on predefined detecting patterns can indeed audit vulnerabilities in a high precision but is also restricted by those patterns. Therefore, other researchers raised data-driven approaches that mainly applied various language models, such as Liu H et al. [6] applied n-gram language model to analyze smart contracts' vulnerabilities in the code sequences level.

Language model in software engineering. The language model has been widely applied in software engineering field, since the essence of programming code is very similar to human natural language but with stricter structure. For example, Sureka A et al. [21] applied the language model to detect duplicate bug reports. Han S et al. [14] proposed a code completion method from abbreviated input using Markov Chain which is the base of language model. In the field of code vulnerability auditing, traditional projects such as C and Java projects have tried language models [23]. Ray B et al. [13] studied the buggy code by n-gram models and testified that buggy code lines are much more unnatural than non-buggy lines which provided the basis of detecting code defects using language models. Wang S et al. [11] applied n-gram language model to analyze code sequences extracted from some Java projects, detecting code defects by capturing semantic information. Nevertheless, those studies including the S-gram tokenized code only by one tokenization standard, where some of the semantic information cannot be fully reflected within restricted sequence length. Therefore, our MSgram adopts multiple tokenization standards to extract code semantic information, which further boosts the performance of vulnerability detection for smart contracts.

VII. CONCLUSION

In this paper, we implemented MSgram and provided two optional combination strategies of Intersection and Union to exploit multi-semantics. Experimental results show that if developers tend to boost their auditing precision, adopting the Intersection strategy could be better. However, if they focus on detecting more true vulnerabilities, the Union strategy is a good choice. Afterwards, we made a comprehensive comparison with the baseline, which demonstrated that the MSgram with multi-semantics outweighed the baseline in the performance of vulnerability detection. The MSgram with either the Intersection or the Union strategy not only obtains greatly improvements on their focusing measures but also keep stability or even excel on other measures. In this case, we came to the conclusion that the novel concept of analyzing smart contracts based on multiple semantics indeed performed much better than that based on single semantic in vulnerability auditing.

ACKNOWLEDGEMENT

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (No.11208017) and the research funds of City University of Hong Kong (7005028, 7005217), and the Research Support Fund by Intel (9220097), and funding supports from other industry partners (9678149, 9440227, 9440180 and 9220103).

REFERENCES

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum project yellow paper, vol. 151, no. 2014, pp. 1-32, 2014.

[2] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," Future Generation Computer Systems, 2017.

[3] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in International Conference on Principles of Security and Trust, 2017: Springer, pp. 164-186.

[4] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks and defenses," arXiv preprint arXiv:1908.04507, 2019.

[5] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on Ethereum smart contract vulnerabilities: a survey," arXiv preprint arXiv:1908.08605, 2019.

[6] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: towards semantic-aware security auditing for ethereum smart contracts," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018: ACM, pp. 814-819.

[7] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in Proceedings of the 26th Conference on Program Comprehension, 2018: ACM, pp. 200-210.

[8] "Etherscan." https://etherscan.io/ (accessed 2019).

[9] C. F. Torres and J. Schütte, "Osiris: Hunting for integer bugs in ethereum smart contracts," in Proceedings of the 34th Annual Computer Security Applications Conference, 2018: ACM, pp. 664-676.

[10] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018: ACM, pp. 259-269.

[11] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016: ACM, pp. 708-719.

[12] F. Bond. "solidity-parser-antlr." https://github.com/federicobond/solidity-parser-antlr (accessed 2019).

[13] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016: IEEE, pp. 428-439.

[14] S. Han, D. R. Wallace, and R. C. Miller, "Code completion from abbreviated input," in 2009 IEEE/ACM International Conference on Automated Software Engineering, 2009: IEEE, pp. 332-343.

[15] C. D. Manning, C. D. Manning, and H. Schütze, Foundations of statistical natural language processing. MIT press, 1999.

[16] 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, 2016: ACM, pp. 254-269.

[17] K. Heafield, "KenLM: Faster and smaller language model queries," in Proceedings of the sixth workshop on statistical machine translation, 2011: Association for Computational Linguistics, pp. 187-197.

[18] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, 2018: IBM Corp., pp. 103-113.

[19] H. Wu, X. Wang, J. Xu, W. Zou, L. Zhang, and Z. Chen, "Mutation testing for Ethereum smart contract," arXiv preprint arXiv:1908.03707, 2019.

[20] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, 2018: ACM, pp. 65-68.

[21] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," in 2010 Asia Pacific Software Engineering Conference, 2010: IEEE, pp. 366-374.

[22] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in 2012 34th International Conference on Software Engineering (ICSE), 2012: IEEE, pp. 837-847.

[23] N. E. Fenton and M. Neil, "A critique of software defect prediction models," IEEE Transactions on software engineering, vol. 25, no. 5, pp. 675-689, 1999.