information

*Article*

# The Impact of Input Types on Smart Contract Vulnerability Detection Performance Based on Deep Learning: A Preliminary Study

**Izdehar M. Aldyaflah [1], Wenbing Zhao [1,*], Shunkun Yang [2] and Xiong Luo [3]**

[1] Department of Electrical and Computer Engineering, Cleveland State University, Cleveland, OH 44115, USA
[2] School of Reliability and Systems Engineering, Beihang University, Beijing 100191, China; ysk@buaa.edu.cn
[3] School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China; xluo@ustb.edu.cn
* Correspondence: wenbing@ieee.org

**Abstract:** Stemming vulnerabilities out of a smart contract prior to its deployment is essential to ensure the security of decentralized applications. As such, numerous tools and machine-learning-based methods have been proposed to help detect vulnerabilities in smart contracts. Furthermore, various ways of encoding the smart contracts for analysis have also been proposed. However, the impact of these input methods has not been systematically studied, which is the primary goal of this paper. In this preliminary study, we experimented with four common types of input, including Word2Vec, FastText, Bag-of-Words (BoW), and Term Frequency–Inverse Document Frequency (TF-IDF). To focus on the comparison of these input types, we used the same deep-learning model, i.e., convolutional neural networks, in all experiments. Using a public dataset, we compared the vulnerability detection performance of the four input types both in the binary classification scenarios and the multiclass classification scenario. Our findings show that TF-IDF is the best overall input type among the four. TF-IDF has excellent detection performance in all scenarios: (1) it has the best F1 score and accuracy in binary classifications for all vulnerability types except for the delegate vulnerability where TF-IDF comes in a close second, and (2) it comes in a very close second behind BoW (within 0.8%) in the multiclass classification.

**Keywords:** blockchain; smart contract; vulnerability detection; Word2Vec; FastText; Bag-of-Words; Term Frequency–Inverse Document Frequency

check for updates

**Citation:** Aldyaflah, I.M.; Zhao, W.; Yang, S.; Luo, X. The Impact of Input Types on Smart Contract Vulnerability Detection Performance Based on Deep Learning: A Preliminary Study. *Information* **2024**, *15*, 302. https://doi.org/10.3390/info15060302

Academic Editors: Georgios Siolas, Georgios Alexandridis and Paraskevi Tzouveli

Received: 15 April 2024
Revised: 17 May 2024
Accepted: 22 May 2024
Published: 24 May 2024

## 1. Introduction

Smart contracts constitute the cornerstone of all decentralized applications [1]. As such, any vulnerability in a smart contract could be detrimental to the security of the application. The most well-known incident related to smart contract vulnerability is the DAO hack that took place in 2016 [2]. Here, DAO is short for decentralized autonomous organization. In the DAO hack, a hacker stole about ETH 3.54 million (worth about USD 150 million at the time) by exploiting the re-entrancy vulnerability in the smart contract that powers the DAO. Because of the severe impact of the incident, the Ethereum foundation was forced to perform a highly controversial hard fork to effectively restore the stolen funds back to the hands of the original investors [3]. The positive impact of the incident is the sharply increased awareness of the smart contract security issue. Various tools have been developed to detect vulnerabilities in smart contracts. In recent years, machine learning and deep learning have also been used to identify vulnerabilities in smart contracts because they could be more robust in detecting vulnerabilities if trained properly.

In many ways, smart contract source code resembles natural languages. Hence, the schemes developed for natural language processing have been used to transform the smart contract source code (typically written in Solidity) into a form of input that is conducive for analysis, either by rule-based vulnerability detection tools or by machine-learning models. A large number of schemes have been proposed, some of which would transform the bytecode of the smart contract instead of the smart contract code into feature vectors. Each scheme focuses on capturing some specific characteristics of the smart contracts. Hence, it would be interesting to study the impact of different types of input towards the detection of smart contract vulnerabilities. Although some studies have incorporated more than one type of input, we have yet to see a systematic study that examines the impact of different types of input in the context of smart contract vulnerability detection.

The goal of this study is to examine the impact of four types of input, namely, Word2Vec, FastText, Bag-of-Words (BoW), and Term Frequency–Inverse Document Frequency (TF-IDF), towards the detection of six common types of smart contract vulnerability, namely re-entrancy, integer overflow, integer underflow, timestamp dependence, delegate call, and call stack depth attack. We choose to use a deep-learning model, the convolutional neural network (CNN), as the classifier. We report the performance of vulnerability detection in term of binary classification (i.e., a particular type of vulnerability vs. normal case) and multiclass classification (i.e., all types of vulnerability and the normal case). The choice of the four types of input and six types of vulnerability is driven by two considerations: (1) these types of input and types of vulnerability are the most heavily reported in the literature of smart contract vulnerability detection studies; (2) the six types of vulnerability have publicly available datasets, and the python code for converting a smart contract to three out of the four types of input is available in GitHub (we developed the python code for TF-IDF).

To our knowledge, this is the first study that systematically examines the impact of the input types for smart contract vulnerability detection, which constitutes the main research contribution of this paper. Our original hypothesis is that different input types would be complementary to each other (at least some of them are) in that one input type could exhibit superior detection performance for some types of vulnerability while another input type would show excellent detection performance for some other types of vulnerability. If proven true, then, we could develop an ensemble model that would select the best input type for each type of vulnerabilities. Unfortunately, our experimental results show that this is not the case. Instead of them being complementary to each other, TF-IDF clearly outperforms all other input types. Nevertheless, we think the findings still carry research merit.

Furthermore, we note that it is not our goal to propose a methodology that outperforms existing approaches. The current study is limited to the study of the impact of the input types on the detection performance of smart contract vulnerabilities. As such, we choose to use CNN as the classifier for experiments because it offers reasonably good performance and it does not require the availability of huge amount of training data. For the same reason, we intentionally do not use any attention mechanisms to improve the classification performance, and we do not use more advanced models such as Bidirectional Encoder Representations from Transformers (BERT) [4].

The remainder of the paper is organized as follows. Section 2 provides the necessary background information for the current study. Section 3 discusses related work. Section 4 describes the methodology of the current study, including the dataset used, input preparation, and the classifier description. Section 5 presents the experimental results and analysis for our study. Section 6 reflects our findings and points out limitations of the current study. Section 7 concludes this paper.

## 2. Research Background

In this section, we briefly introduce the background of this study, including the definition of the six types of smart contract vulnerability and the four types of input methods.

### *2.1. Smart Contract Vulnerabilities*

A large array of vulnerability types have been identified and there is no universally accepted taxonomy for the vulnerabilities. In this paper, we adopt the taxonomy proposed in [5]. Here, we provide a brief introduction of the limited set of vulnerabilities that we aim to detect in the current study.

The *integer overflow and underflow* vulnerabilities occur when integer arithmetic operations produce numbers greater than the maximum or smaller than the minimum representable values, which would result in unexpected behaviors. Attackers could use this vulnerability to modify calculations or overflow/underflow checks, potentially resulting in unauthorized access or financial loss. To avoid this vulnerability before performing arithmetic operations, the operators and operands should be compared to ensure accuracy. Also, one could utilize assert(), require(), and the 'SafeMath.sol' package in the smart contracts [6].

A *re-entrancy* attack occurs when a malicious contract exploits flaws in the target contract's logic and frequently calls back into the target contract before the first call is completed. This can result in unanticipated actions such as illegal financial transfers or contract state manipulation. To eliminate the re-entrancy vulnerability, the functions should be structured to ensure that all internal state changes take place before calling another contract [7].

The *timestamp dependency* vulnerability refers to the use of the block timestamp value to execute an operation in a smart contract. The block timestamp is generated by the node that executes the smart contract. The issue of this vulnerability is that it makes the contract vulnerable to attacks and prone to manipulation. For example, the creating node of the block could manipulate the blockchain timestamp value to maximum monetary profit. Using the block height is preferable to using the block timestamp [8].

In the *callstack depth attack vulnerability* (CDAV), the attacker takes advantage of the Ethereum Virtual Machine (EVM)'s low callstack depth to produce a denial-of-service scenario or interrupt the blockchain network's normal operation. An attacker can launch a callstack depth attack by building a recursive or deeply nested call chain that exceeds the EVM's maximum callstack depth limit [9]. The Ethereum Improvement Proposal (EIP) 150 has placed a gas-based restriction on the call stake, which essentially would eliminate this attack.

To invoke another contract's function, the target function's application binary interface (ABI) must be available. A delegate call is used in cases when the target function's ABI is not available. One intention of the delegate call feature is to enable the call of the functions defined in another smart contract as if it is their own. This feature might also enable upgradability of the smart contract. However, the context-preserving aspect of the feature may lead to security issues, such as access violation. That is why *delegate* is regarded as a vulnerability and it has been exploited in attacks. To avoid potential problems, it is recommended that the delegated contract be stateless [10].

### *2.2. Types of Input*

We limit the discussion of the types of input to four specific types of input that are used in the current study, although a large number of input types have been proposed. These types of input are selected because of their popularity and the availability of open-source code.

*Word2Vec* was created by a group of Google researchers to represent word distribution in a vector space [11]. Word2Vec was designed to identify semantic similarities between words where words with similar meanings would have less distance [11]. To apply Word2Vec, the smart contract source code would be segmented into words (also referred to as word embedding), which would then be vectorized into numerical values.

More specifically, each smart contract fragment is expressed as a set of tokens (i.e., words). First, a model referred to as continuous bag-of-words (CBOW) is trained to predict the center token in the fragment. For an instance, with context tokens like "if", "sender", and "balance", the CBOW model may predict the center token to be "require". CBOW might be beneficial for finding similar patterns or tokens related to vulnerabilities in smart contracts.

Second, a model referred to as skip-gram is trained to predict the surrounding tokens given a center token. For instance, if "transfer" is the center token, the skip-gram model may predict "send", "recipient", or "amount" as surrounding tokens. By training the skip-gram model on a large dataset of smart contracts, it is able to understand the distributional features of code tokens and record their semantic links. Once given a new smart contract fragment, the trained skip-gram model could recognize tokens that commonly occur together with recognized vulnerable tokens, which may indicate the existence of vulnerabilities.

*FastText* extends the Word2Vec scheme by treating each word as an n-gram, which aims to provide a more nuanced representation of words. FastText could potentially generate better word embeddings for rare words and out-of-vocabulary words based on the shared structures with other words [12].

Similar to Word2Vec, FastText is also trained on a set of smart contract fragments. The training process is best understood using an example. For instance, when FastText comes across the token "untransferable", it divides the token into sub-words as n-grams, including "un", "transfer", "transf", "rans", "ansf", "nstr", "strans", "stransf", "tr", "ran", "ans", "ns", "sf", "f", and "untransferable". This step is to facilitate the model to recognize that "untransferable" is related to other tokens such as "transfer", "untransferred", "transferFrom", "untransferrable", because they contain the same sub-word "transfer". This is how FastText learns semantically and morphologically related tokens.

*BoW* represents a text as a bag (i.e., multiset) of its words, keeping track of the multiplicity of each word, but ignoring grammar and word order [13]. BoW focuses on capturing the frequency of words in a predefined set of words. More specifically, BoW would create a vocabulary by extracting all unique tokens from a set of smart contract fragments. Each token is assigned as a feature. The frequency of each token in the vocabulary is then counted within each fragment. BoW would convert each fragment into a vector, with each value indicating the frequency (i.e., number of instances) of a token in the vocabulary. The vector representation preserves the syntax and structure of the code fragment.

*Term Frequency–Inverse Document Frequency* (TF-IDF) uses two components to capture the importance of each word in a set of smart contract fragments [14]: (1) Term Frequency (TF): This measures the frequency of a token in a smart contract fragment, e.g., in terms of the raw count of the token as the fraction of the total token count in the fragment. The intuition is that the word would be more important if it appears a greater number of times in the same document. (2) Inverse Document Frequency (IDF): This measures the importance of a token in the entire collection of fragments. For a particular token, IDF is represented as the logarithm of the total number of fragments in the collection divided by the number of fragments containing the token. The intuition is that, if a token is present in many fragments, then it is less important as a feature for pattern recognition.

## 3. Related Work

A large body of work has been published on machine-learning-based detection of smart contract vulnerabilities. In this paper, we focus on studies that have employed deep-learning models for detection. Given sufficient training data, deep-learning models typically attain better performance than traditional machine-learning models, as we have demonstrated previously [15]. We further limit the related works to those that have adopted the same taxonomy on smart contract vulnerabilities [5] (Table 1).

**Table 1.** Summary of related work.

| Study | Input Type(s) | Types of Vulnerability Detected |
|---|---|---|
| DeeSCVHunter [16] | FastText (Word2Vec + Glove) | Re-entrancy and timestamp dependency |
| CBGRU [17] | Word2Vec+FastText | Re-entrancy, timestamp dependency, integer overflow/underflow, CDAV, and infinite loop |
| Peculiar [18] | Graph | Re-entrancy |
| BLSTM-ATT [19] | Sequential | Re-entrancy |
| TMP [20] | Graph | Re-entrancy, timestamp dependency, and infinite loop |
| AME [21] | Graph | Re-entrancy, timestamp dependency, and infinite loop |
| DA-GCN [22] | Graph | Re-entrancy and timestamp dependency |
| HAM [23] | Word2Vec | Re-entrancy, timestamp dependency, arithmetic vulnerability, unchecked return value, and Tx.origin |
| SPCBIG-EC [24] | Word2Vec | Re-entrancy, timestamp dependency, and infinite loop |

In [16], the re-entrancy and timestamp dependency vulnerabilities were detected (separately) using eight deep-learning models. The primary innovation was the introduction of an additional step prior to performing word embedding, which is referred to as the vulnerability candidate slice (VCS). The VCS was inspired by a common practice of extracting regions of interest in an image for recognition. Hence, the method was termed as DeeSCVHunter in the paper. The paper stated that three different word embeddings were employed, including Word2Vec, FastText, and Glove, and FastText was used as the default embedding method. However, the paper did not report the detection performance for each of the embedding methods. Presumably, the best performance out of the three embedding methods was reported.

In [17], Word2Vec and FastText were used for word embedding. Furthermore, CNN was used to perform further feature extraction based on the output of the Word2Vec embedding, and the bidirectional gated recurrent unit (BiGRU) was used to perform further feature extraction based on the output of the FastText embedding. Then, the features extracted by CNN and BiGRU were combined by concatenation. Then, a fusion neural network layer and a softmax neural network layer were used to perform classification based on the fused input. The dataset contains six different types of vulnerability, including integer overflow, integer underflow, re-entrancy, timestamp dependency, CDAV, and the infinite loop. It appears that binary classification was performed for each type of vulnerability.

In [18], a single type of vulnerability, i.e., re-entrancy, was detected using a transformer neural architecture called GraphCodeBERT [25] and an improved version of data flow graph (referred to as a crucial data flow graph) as the way to encode the smart contracts for classification. The paper reported the classification performance using two datasets.

In [19], a new model that converts the smart contract to a vector format was proposed (referred to as a sequential model). The study used a deep-learning model called the bidirectional long short-term memory with attention mechanism (BLSTM-ATT) to perform the detection of the re-entrancy vulnerability in smart contracts.

In [20], the smart contracts were converted into a graph format (combining control flow, data flow, and fallback information). Two deep-learning models were proposed to detect vulnerabilities based on normalized graph input. One model is referred to as a degree-free graph convolutional neural network (DR-GCN), and the other is a novel temporal message propagation network (TMP). The performance of the proposed input and two deep-learning models was reported for the detection of each of three types of vulnerability, namely, re-entrancy, timestamp dependency, and infinite loop. TMP was shown to have better performance than DR-GCN.

In [21], the same graph format that combines the control flow, data flow, and fallback information in the smart contracts was used as the input. Differently from that of [20], four levels of input (three local patterns, and one global graph-based input similar to that of [20]) were experimented on to see their impact on the detection performance. An attentive multi-encoder network was used to detect each of three types of vulnerability: re-entrancy, timestamp dependency, and infinite loop. The different levels of input were used to illustrate the interpretability of the detection process.

In [23], Word2Vec was used for word embedding, and a hybrid attention mechanism with deep learning was used for the detection of vulnerabilities. The detection performance was presented for each of the five types of vulnerability, namely re-entrancy, timestamp dependency, arithmetic vulnerability, unchecked return value, and Tx.origin vulnerability.

In [22], a control flow graph was used to represent the smart contract fragments as the input to the deep-learning model, which is referred to as a dual attention graph convolutional network (DA-GCN). Two types of smart contract vulnerability, namely, re-entrancy and timestamp dependency, were detected separately.

In [24], Word2Vec was used to encode the smart contract fragments as the input to a sophisticated deep-learning model referred to as the Serial–Parallel Convolutional Bidirectional Gated Recurrent Network Model incorporating Ensemble Classifiers (SPCBIG-EC). Two types of smart contract vulnerability, re-entrancy and timestamp dependency, were detected separately.

As can be seen, most of related studies have chosen to use a single input type. Although three types of input were mentioned in [16], FastText was used as the default input type, and the study did not disclose any impact of the input types on the vulnerability detection performance. In [17], Word2Vec and FastTest were fused together as the input. We are not aware of any study that systematically compared the impact of different input types on the detection performance.

Again, we note that it is not our goal is to propose a method that outperforms other approaches. Nevertheless, for completeness, we show the vulnerability detection performance of the related studies compared with that of ours in Table 6 in Section 5.3. The purpose of the comparison is to summarize what has been studied and the reported results instead of drawing any definitive conclusion on which approach is superior because these studies often used different datasets, in addition to the use of different input types and different classifiers.

## 4. Methodology: Dataset, Input Preparation, and Classifier Description

The primary objective of the current study is to investigate the impact of the four input types on the smart contract vulnerability detection performance using the same deep-learning model. The reasons for using CNNs for vulnerability classification will be elaborated in Section 4.2. As we have outlined in Section 2.2, the four types of input we plan to study have different approaches to extracting the features of the smart contracts. Word2Vec and FastText consider the similarity between different words. BoW focuses on the frequency of the words. TF-IDF also focuses on the word frequency, but considers not only the presence in a single smart contract fragment but also the presence of words in the entire corpora. The flow of the detection process is shown in Figure 1.

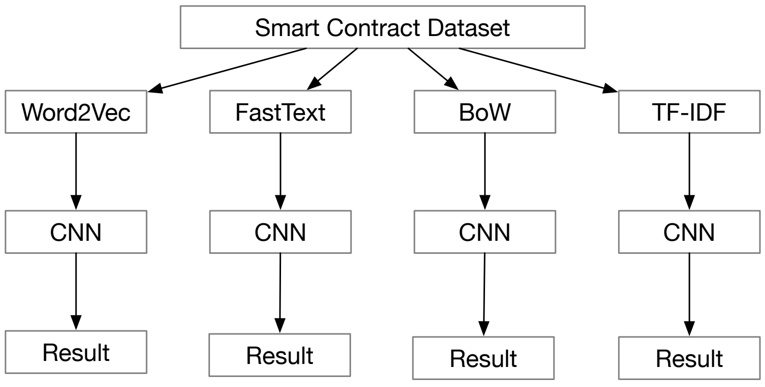


**Figure 1.** The overview of methodology in detection of smart contract vulnerabilities.

### *4.1. Data Preprocessing*

The data preprocessing step is to convert the original smart contract into a matrix that conforms to the input requirement of CNNs using the four methods we outlined previously. More specifically, the preprocessing of each smart contract is outlined as follows:

- All Ethereum solidity keywords are gathered, including "bool", "break", "return", "assert", "event", etc.
- Eliminate all components of the smart contract that are not related to the vulnerability (such as "pragma solidity 0.5.8"), and eliminates blank lines, comments, and non-ASCII values from the contract.
- Represent variable names as VAR with numbers (such as VAR1, VAR2, ...), and function names as FUN with numbers (such as FUN1, FUN2, ...).
- Tokenize each smart contract fragment line by line.
- Gather these tokens to create a matrix using the input method.

### *4.2. Feature Extraction and Classification with CNN*

CNNs have been predominantly used for image processing with excellent spatial characterization. A CNN would take fixed input sizes because of the use of fixed-size filters. Because smart contract is executed sequentially, one might expect that the input would be formulated as a time series. In fact, this is not the case for the four input types in our current study. For these types of input, a smart contract fragment is transformed into a fixed-dimension vector. In fact, during the revision stage of this paper we became aware of a paper on smart contract vulnerability detection based on CNNs [26] where each smart contract fragment was transformed into an image format and excellent results were achieved. Due to the above evidence, we argue that the use of CNN for classification is justified.

Once the data preprocessing is completed, the vectors are fed into the CNN for feature extraction and classification. The CNN architecture is shown in Figure 2. The CNN layers are described as following:

- Convolution Layer 1: We choose to use a one-dimensional convolution layer (Convolution Layer 1). The sizes for Word2Vec, FastText, BoW, and TF-IDF are (300, 100), (300, 100), 37, and 300, respectively.
- MaxPooling 1: Max pooling reduces the spatial dimension of the input data by only preserving the maximum value within each pooling window; hence, it helps in lowering computational complexity and managing overfitting. By offering translation in variance, max pooling lowers the number of parameters in the model and increases its resilience to small changes in the input data.
- Convolution Layer 2: This layer further extracts higher-level features.
- MaxPooling 2: This layer further decreases the dimensions of the feature maps.
- Dropout: This layer facilitates the learning of more robust features during training by arbitrarily setting a portion of the input units to zero. This helps prevent overfit-

ting [27]. By decreasing neuronal co-adaptation and enhancing generalization ability, dropout regularizes the model.
- Flatten: This layer flattens the input into a one-dimensional vector required by dense layers.
- Dense Layer 1: This fully connected layer performs a linear transformation with a rectified linear activation function. From the features that are extracted, it learns complex patterns and representations.
- Dense Layer 2: This fully connected layer performs further recognition of complex patterns in the data.
- Dense Layer 3: This final dense layer computes the probability distributions over the classes by applying the softmax activation function and translates the learned representations to the output classes. The softmax is an activation function typically used for classification. The function turns raw output results into probabilities that reflect the possibility of each class.

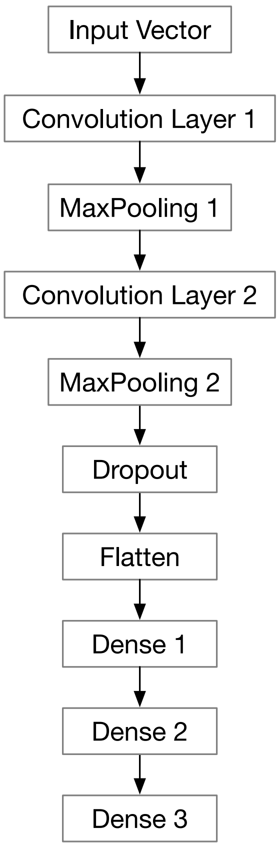


**Figure 2.** CNN layers.

### *4.3. Deep-Learning Library and Parameters*

Python is used as the programming language with the TensorFlow [28] and Keras deep-learning libraries. The Adam optimizer [29] is used with a learning rate at 0.001. The dropout of the Dropout layer is set at 0.5. Furthermore, an epoch of 50 and a batch size of 128 are used in the experiments. Five-fold cross-validation is used for the training and testing of deep learning. All experiments are conducted on an iMac-27 with a core i5 CPU and 64 GB of RAM.

The reason for using exactly the same set of hyper-parameters is to maintain consistency when comparing the impact of the input types for the performance of detection of smart contract vulnerabilities. We reiterate here that it is not our goal to propose a method that outperforms existing approaches for smart contract vulnerability detection. Hence, we intentionally do not tune the parameters for each type of input and for each type of vulnerability, which could potentially lead to improved performance. We choose to use Adam optimizer because it is a widely used optimization algorithm in deep learning for computer vision and natural language processing tasks. Furthermore, we set the dropout rate to 0.5 in our model because it is a commonly used regularization technique to prevent overfitting in neural networks by randomly dropping units during training. Similarly, we set the number of epochs to 50 and the batch size to 128 because these values are commonly used in deep-learning experiments for efficient training and convergence [17].

### *4.4. Smart Contract Dataset*

The experiments are based on the SmartBugs Dataset—Wild [30], which is a large-scale dataset of smart contract vulnerabilities. This dataset has been used in several recent studies, such as [17]. This dataset includes 47,587 genuine and distinct Solidity files in this collection and roughly 203,716 smart contract fragments in total. The dataset was labeled in accordance with [31] in [17]. The Solidity files in the dataset were separated into two categories: smart contracts with vulnerabilities and those without vulnerabilities. Vulnerable smart contracts include seven types of vulnerability, re-entrancy, timestamp dependency, integer overflow, integer underflow, callstack depth attack (CDAV), delegate, and integer big vulnerabilities. There are 12,247 smart contract fragments that are vulnerability-free and 35,151 smart contract fragments that have vulnerabilities. We intentionally removed the fragments for the integer big because this type strongly correlate with integer underflow and integer overflow. Furthermore, the dataset contains preprocessed smart contract fragments rather than the raw smart contracts. The number of smart contract fragments for each type of vulnerability in the dataset is provided as follows:

- Re-entrancy: 1224 fragments.
- Timestamp Dependency: 2908 fragments.
- Integer Overflow: 550 fragments.
- Integer Underflow: 4000 fragments.
- CDAV: 2800 fragments.
- Delegate: 980 fragments.

## 5. Vulnerability Detection Results and Analysis

We first present the results for binary classification scenarios. Then, we present the results for the multiclass classification scenario. Finally, we present a comparison with related work.

### *5.1. Binary Classification*

In binary classification, we perform detection of a single type of vulnerability at time. The dataset for each type of vulnerability would consists of a number of fragments with the same type of vulnerability and a number of fragments without any vulnerability. The performance of the binary classification is evaluated using four metrics: accuracy, precision, recall, and F1 score. All four metrics have a range of 0 to 1, with a greater number indicating better performance.

Accuracy is defined as the fraction of correctly classified entries among all instances in the dataset, as shown in Equation (1):

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{1}$$

To define precision, recall, and F1 score, it is necessary to first define four additional metrics: true positive (TP), true negative (TN), false positive (FP), and false negative (FN):

- True Positive (TP): The number of predictions accurately identified as belonging to the positive class.
- True Negative (TN): The number of predictions correctly identified as belonging to the negative class.

- False Positive (FP): The number of predictions wrongly classified as positive.
- False Negative (FN): The number of predictions mistakenly classified as negative.

Recall is also referred to as sensitivity or true positive rate. It is defined to be the fraction of true positive predictions among all real positive instances in a dataset as shown in Equation (2). Recall demonstrates the classifier's ability to identify all of the positive samples.

$$\text{Recall} = \frac{\text{TP}}{\text{TP + FN}} \tag{2}$$

Precision is defined to be the fraction of true positive predictions out of all positive predictions generated by the classifier, as shown in Equation (3). Precision demonstrates the classifier's ability to minimize false positives (i.e., classifying negative samples as positive).

$$\text{Precision} = \frac{\text{TP}}{\text{TP + FP}} \tag{3}$$

F1 score is defined to be the harmonic mean of precision and recall, as shown in Equation (4). The F1 score aims to strike a balance between precision and recall. The F1 score is especially useful when the class distribution has an imbalance or when false positives and false negatives have distinct consequences.

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

#### 5.1.1. Word2Vec Performance

The vulnerability detection performance using the Word2Vec input is shown in Figure 3 and Table 2. Among the six types of vulnerability, delegate and integer overflow have the highest detection performance. For the delegate vulnerability, both the accuracy and F1 score are higher than 96%. For the integer overflow vulnerability, both the accuracy and F1 score are higher than 94%. Furthermore, for both types of vulnerability, the recall is perfect, meaning that all vulnerabilities in the testing dataset have been identified. Hence, this input type made some false negative predictions. The detection performance for the CDAV, integer underflow, and re-entrancy lies in the second tier at 84–87% accuracy and F1 score range. The timestamp dependency is the most difficult to detect, with accuracy, precision, and recall all below 80% (only the recall is higher than 80%). It is interesting to note that, for all types of vulnerability, recall is higher than precision, meaning that there are more false positives than false negatives (if any).

#### 5.1.2. FastText Performance

The vulnerability detection performance is summarized in Figure 4 and Table 3. As can be seen, all four metrics (accuracy, precision, recall, and F1 score) are perfect at 100% for detection of the delegate vulnerability. The detection performance for integer overflow comes next with a perfect recall and accuracy of 91.67%. The presence of false positives reduces the precision, similar to the situation for Word2Vec. Again, the detection performance for CDAV, integer underflow, and re-entrancy is similar with accuracy in the range of 85–87%, and F1 score in the range of 84–87%. The detection performance for timestamp dependency comes last, with accuracy and F1 score below 80%. The overall trend for FastText is rather similar to that of Word2Vec.

#### 5.1.3. BoW Performance

The vulnerability detection performance with BoW is summarized in Figure 5 and Table 4. Again, delegate is proven to be the easiest to detect with accuracy at 99.19% and F1 score at 99.20% (the recall is perfect and the precision is at 98.41%). The detection performance for integer overflow is still in the second place with accuracy at 86.11% and F1 score at 87.50% (the recall at 97.22% is significantly higher than precision at 79.55%). The remaining types of vulnerability stand in the third tier with accuracy and F1 score both in the range of 74–79%. Unlike Word2Vec and FastText, recall is not always higher than precision. For re-entrancy and integer underflow, the precision is higher than recall, which means there are more false negatives than false positives.

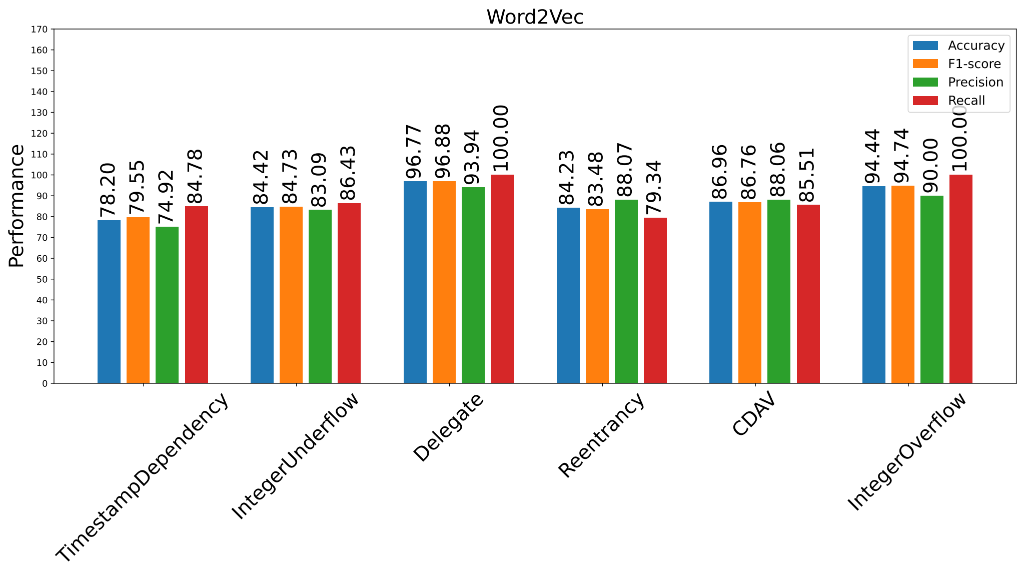


**Figure 3.** Vulnerability detection performance with Word2Vec embedding.

**Table 2.** Vulnerability detection performance with Word2Vec embedding.

| Dataset | Accuracy | Recall | Precision | F1 Score |
|---|---|---|---|---|
| Delegate | 96.77 | 100.00 | 93.94 | 96.88 |
| Integer Overflow | 94.44 | 100.00 | 90.00 | 94.74 |
| CDAV | 86.96 | 85.51 | 88.06 | 86.76 |
| Integer Underflow | 84.42 | 86.43 | 83.09 | 84.73 |
| Re-entrancy | 84.23 | 79.34 | 88.07 | 83.48 |
| Timestamp Dependency | 78.20 | 84.78 | 74.92 | 79.55 |

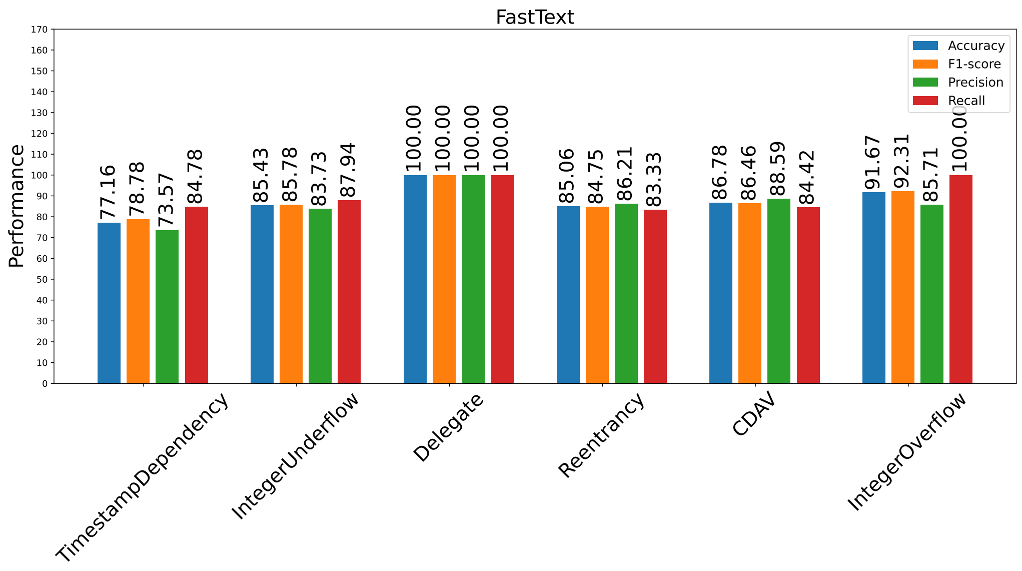


**Figure 4.** Vulnerability detection performance with FastText embedding.

**Table 3.** Vulnerability detection performance with FastText embedding.

| Dataset | Accuracy | Recall | Precision | F1 Score |
|---|---|---|---|---|
| Delegate | 100.00 | 100.00 | 100.00 | 100.00 |
| Integer Overflow | 91.67 | 100.00 | 85.71 | 92.31 |
| CDAV | 86.78 | 84.42 | 88.59 | 86.46 |
| Integer Underflow | 85.43 | 87.94 | 83.73 | 85.78 |
| Re-entrancy | 85.06 | 83.33 | 86.21 | 84.75 |
| Timestamp Dependency | 77.16 | 84.78 | 73.57 | 78.78 |

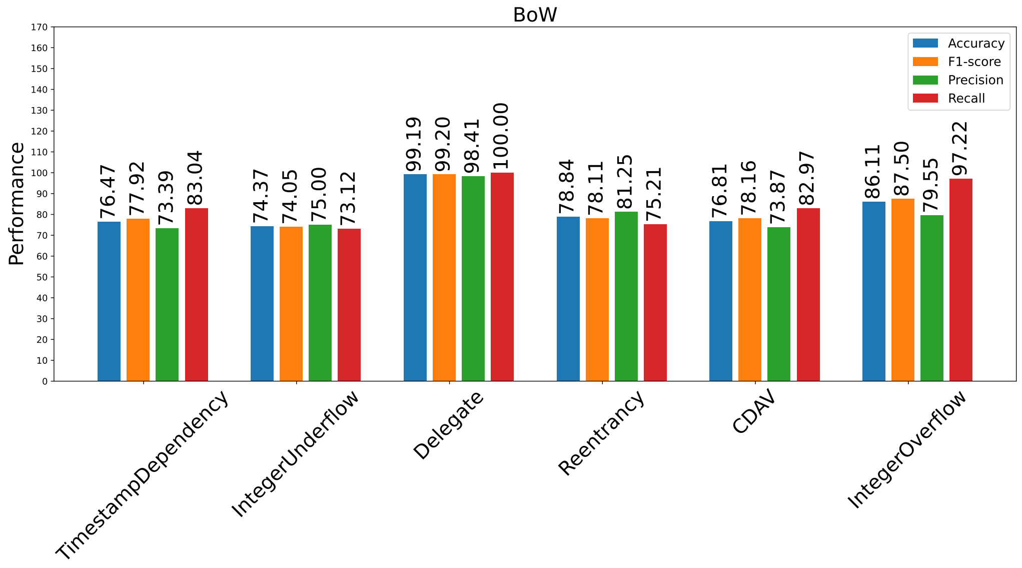


**Figure 5.** Vulnerability detection performance with BoW.

**Table 4.** Vulnerability detection performance with BoW.

| Dataset | Accuracy | Recall | Precision | F1 Score |
|---|---|---|---|---|
| Delegate | 99.19 | 100.00 | 98.41 | 99.20 |
| Integer Overflow | 86.11 | 97.22 | 79.55 | 87.50 |
| Re-entrancy | 78.84 | 75.21 | 81.25 | 78.11 |
| CDAV | 76.81 | 82.97 | 73.87 | 78.16 |
| Timestamp Dependency | 76.47 | 83.04 | 73.39 | 77.92 |
| Integer Underflow | 74.37 | 73.12 | 75.00 | 74.05 |

#### 5.1.4. TF-IDF Performance

The vulnerability detection performance for TF-IDF is summarized in Figure 6 and Table 5. Delegate is still proven to be the easiest to detect with the highest accuracy (99.19%) and F1 score (99.20%). Unlike other types of input, four types of vulnerability come next with detection accuracy and F1 score both in the range of 94–96%. Although the detection performance for the timestamp dependency still comes last, the accuracy (85.29%) and F1 score (85.37%) are both significantly better compared with other types of input. Furthermore, recall is slightly higher than precision consistently across all types of vulnerability.

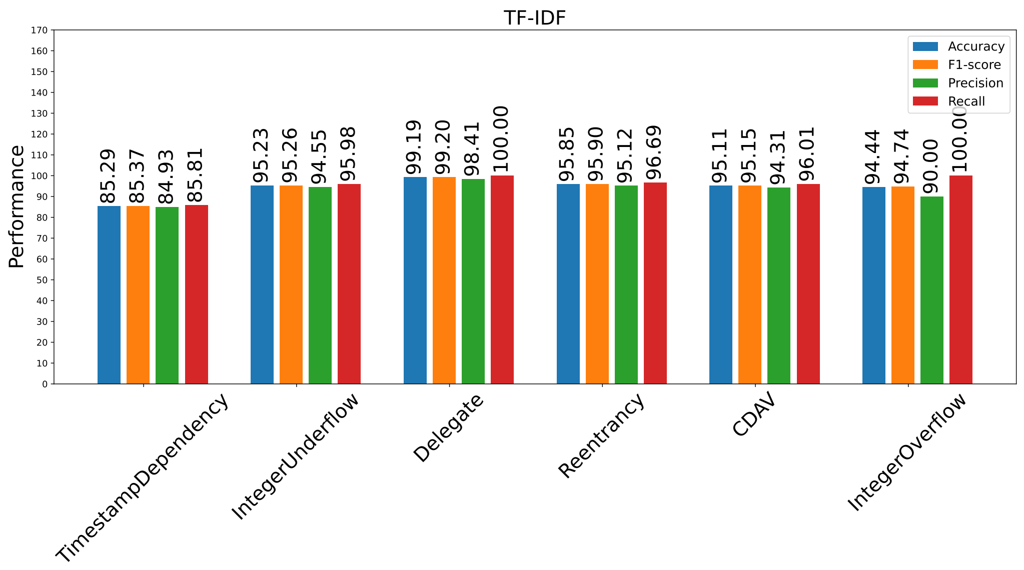


**Figure 6.** Vulnerability detection performance with TF-IDF.

**Table 5.** Vulnerability detection performance with TF-IDF.

| Dataset | Accuracy | Recall | Precision | F1 Score |
|---|---|---|---|---|
| Delegate | 99.19 | 100.00 | 98.41 | 99.20 |
| Re-entrancy | 95.85 | 96.69 | 95.12 | 95.90 |
| Integer Underflow | 95.23 | 95.98 | 94.55 | 95.26 |
| CDAV | 95.11 | 96.01 | 94.31 | 95.15 |
| Integer Overflow | 94.44 | 100.00 | 90.00 | 94.74 |
| Timestamp Dependency | 85.29 | 85.81 | 84.93 | 85.37 |

#### 5.1.5. Comparing the Impact of the Four Types of Input

To facilitate examining the impact of the four types of input on the vulnerability detection performance, we redraw the results in terms of accuracy, precision, recall, and F1 score in Figure 7, Figure 8, Figure 9, and Figure 10, respectively. As can be seen, for accuracy and F1 score, TF-IDF performs consistently better than the other three types of input for five types of smart contract vulnerability. For the integer overflow vulnerability, TF-IDF ties Word2Vec as the best performer (94.44% for accuracy and 94.74% for F1 score). For recall, TF-IDF is either the best or tied best performance among the four types of input for all six types of vulnerability (for delegate, all four types of input have perfect recall, and for integer overflow, Word2Vec, FastText, and TF-IDF all have perfect recall). However, for precision, TF-IDF is tied as the second best performance for the delegate vulnerability, losing to FastText by a very small margin (98.41% vs. 100%). Nevertheless, overall TF-IDF is the most preferable type of input among the four for the detection of the six smart contract vulnerabilities.

The second overall best performer is FastText. Word2Vec comes in a close third, and BoW has the worst performance. Considering that FastText is based on Word2Vec and is an improvement over Word2Vec, it is expected that FastText performs better than Word2Vec. More specifically, FastText could handle morphologically rich languages by adding sub-word information, which enables FastText to capture variations in word structure and semantics, which improves its performance in specific contexts.

In addition to considering the overall detection performance, it is also important to consider the complexity of different types of input. In our study, we choose to have a shape of (100, 300) for Word2Vec and FastText, meaning that each input matrix contains 100 words (i.e., features) with 300 dimensions for each word. TF-IDF encodes the data as a 300-dimension vector (i.e., the vocabulary has a size of 300). BoW uses a much smaller vector of 37 vocabulary words. Despite the very small vector used by BoW, its detection performance is on par with the best types of input for the detection of the delegate vulnerability. This analysis further demonstrates that TF-IDF stands out as the best choice of input type because it achieves the best performance with a significantly smaller dimension compared with Word2Vec and FastText.

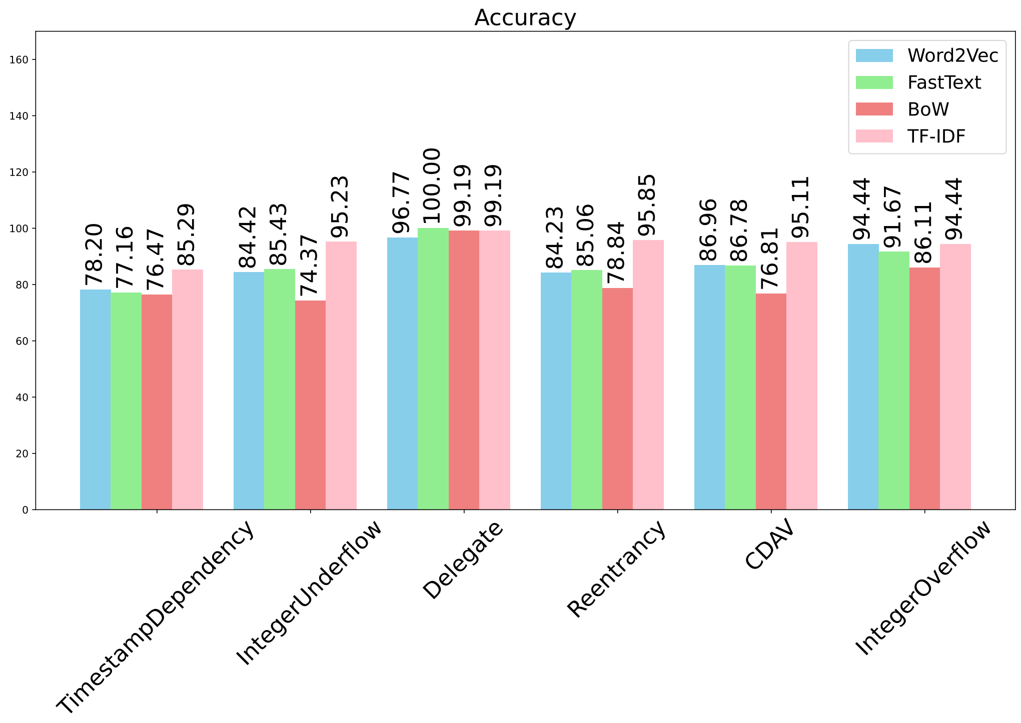


**Figure 7.** Detection accuracy for six types of smart contract vulnerability with the four types of input.

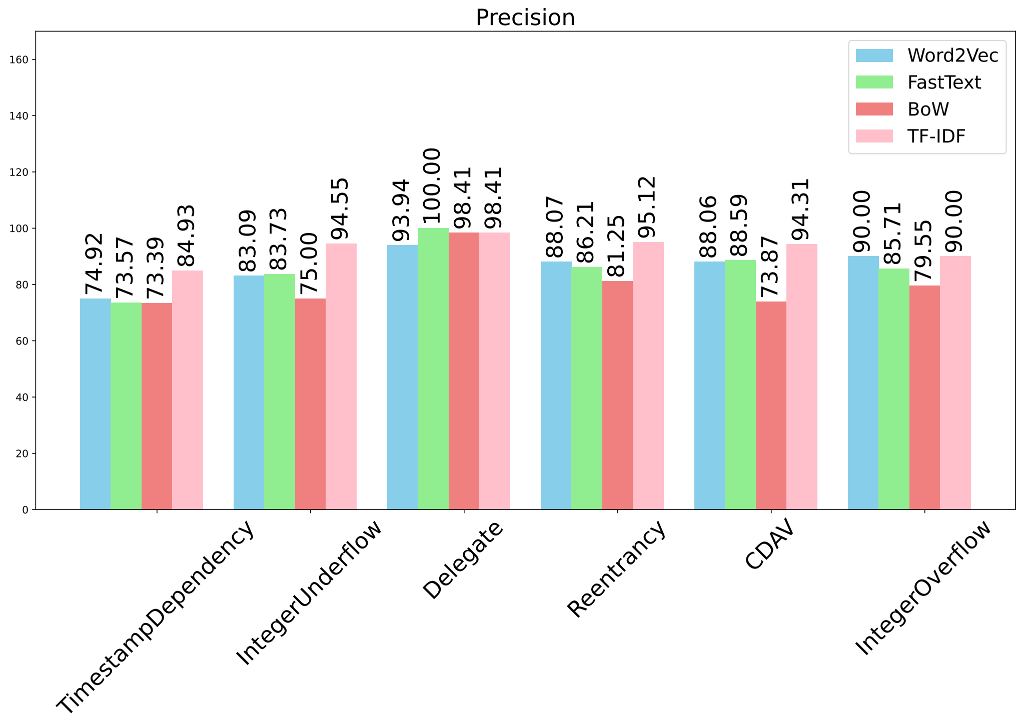


**Figure 8.** Detection precision for six types of smart contract vulnerability with the four types of input.

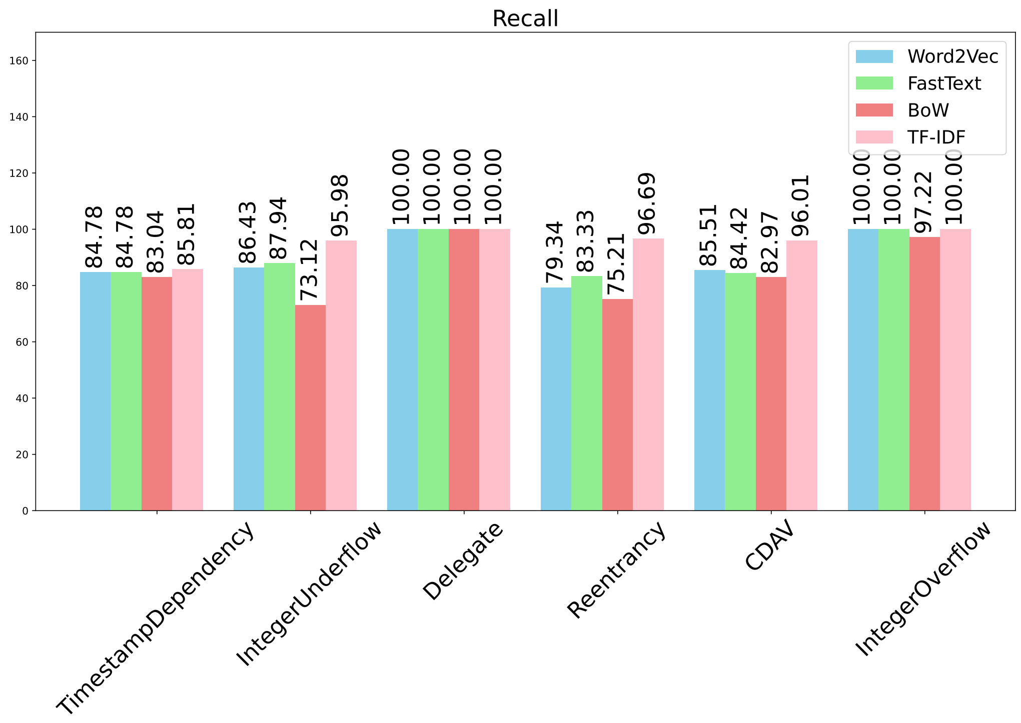


**Figure 9.** Detection recall for six types of smart contract vulnerability with the four types of input.

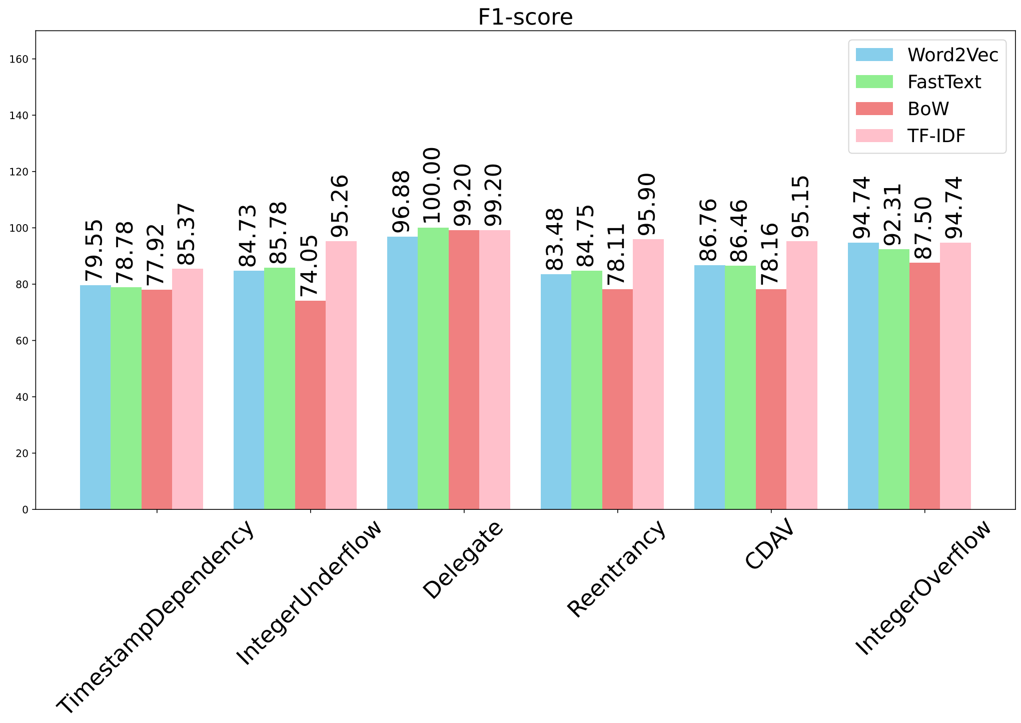


**Figure 10.** Detection F1 score for six types of smart contract vulnerability with the four types of input.

### *5.2. Multiclass Classification*

Although the detection of a single type of vulnerability has some value in research, its practical impact is very limited because one would not know which type of vulnerability exists in a given smart contract in general. Furthermore, it is also important to inform what types of vulnerability a smart contract contains. Scientifically, it is interesting to determine the power of a machine-learning model with a certain type of input to discern different types of vulnerability. Hence, it is informative to conduct multiclass classification, although this is rarely performed in machine-learning-based vulnerability detection for smart contracts.

To facilitate the multiclass classification study, we first construct the training dataset based on those for individual datasets created for binary classification studies. The six types of vulnerability and the no-vulnerability fragments together constitute seven classes. We re-labeled the data as follows: no vulnerability as class 0, timestamp dependency vulnerability as class 1, re-entrancy vulnerability as class 2, integer underflow vulnerability as class 3, delegate vulnerability as class 4, CDVA vulnerability as class 5, and integer overflow vulnerability as class 6.

Because the size of the dataset for each type of vulnerability is quite different, we are facing a class imbalance problem. To avoid the imbalance between the classes during training, the Synthetic Minority Over-sampling Technique (SMOTE) [32] and Undersampling are used. SMOTE works by oversampling the minority classes and it is used to augment minority classes. Undersampling is used on majority classes.

The vulnerability detection overall accuracy for the four types of input is summarized in Figure 11. The overall accuracy is calculated as the fraction of correctly classified samples with respect to all the samples in the test set. As we can see in Figure 11, although TF-IDF still has good performance (at 95.09%), BoW actually has the highest accuracy at 95.84%. Word2Vec comes as the third best performer at 85.07%, while FastText performs in the last place at 84.68%. This is significantly different from the results in binary classification. To understand the details of multiclass classification, it is necessary to inspect the confusion matrix [33] for each type of input, which is shown in Figure 12, Figure 13, Figure 14, and Figure 15, respectively.

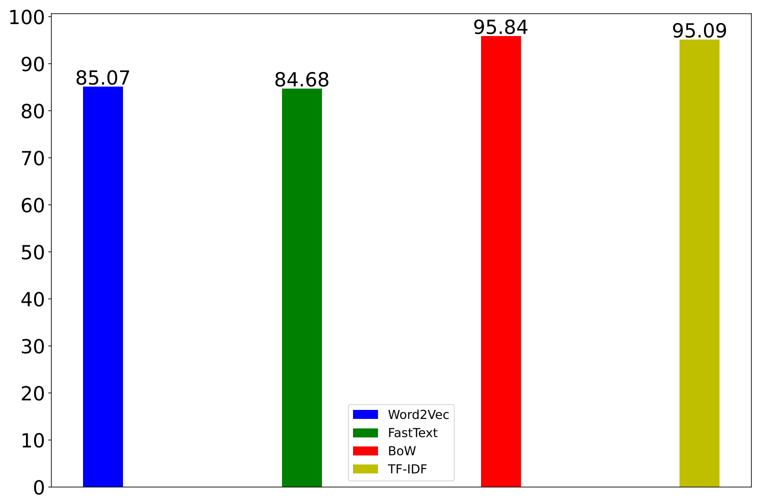
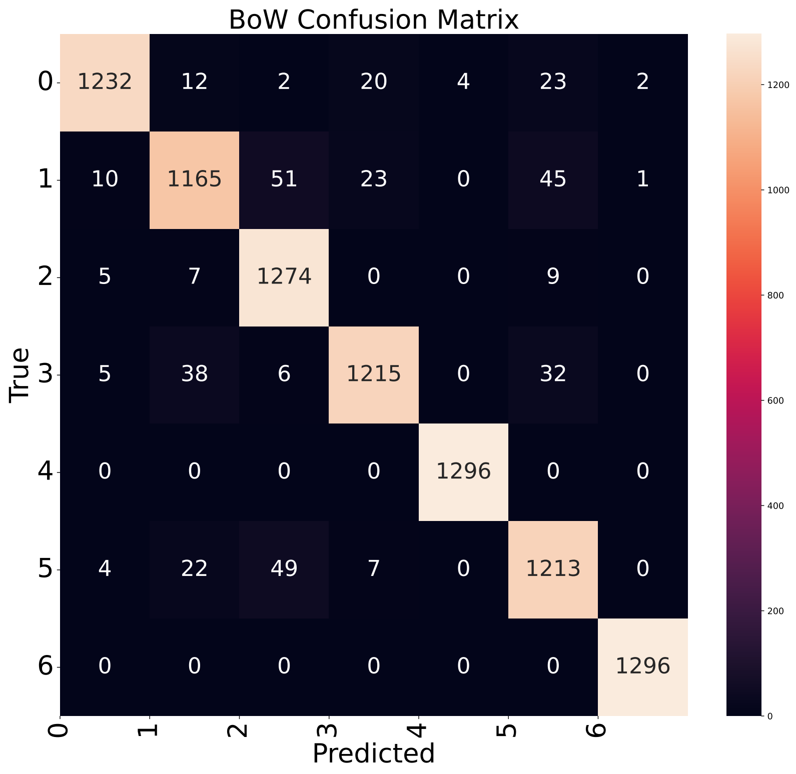


**Figure 11.** Overall accuracy with Word2Vec, FastText, BoW, and TF-IDF for multiclass classification.

Each row of the confusion matrix represents instances in an actual class, and each column represents instances in a predicted class. We define the misclassification rate as the fraction of the sum of misclassified instances in a row of the total number of instances in the row.

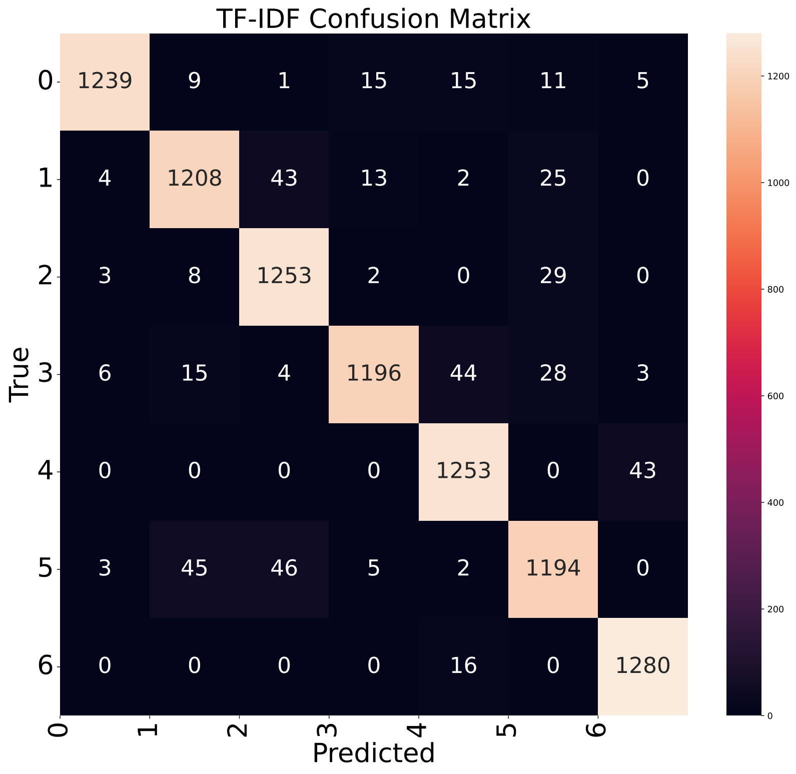
For BoW, the classifier with BoW has the most difficulty detecting timestamp dependency, and can detect delegate and integer overflow perfectly. The result is consistent with binary classification using BoW for timestamp dependency and delegate; it is somewhat surprising that BoW is capable of detecting integer overflow perfectly in multiclass classification, while it has only 87.50% F1 score in binary classification. The low F1 score is primarily due to low precision, which means BoW suffers from false positives in binary classification for integer overflow vulnerability.

For TF-IDF, the classifier with TF-IDF has the most difficulty detecting CDVA, integer underflow, and timestamp dependency, and it has the best performance in detecting integer overflow. Again, the result is not consistent with that of the binary classification with TF-IDF, where only the detection performance for timestamp dependency is noticeably lower than the remaining five types of vulnerability.



| True \ Predicted | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 1232 | 12 | 2 | 20 | 4 | 23 | 2 |
| 1 | 10 | 1165 | 51 | 23 | 0 | 45 | 1 |
| 2 | 5 | 7 | 1274 | 0 | 0 | 9 | 0 |
| 3 | 5 | 38 | 6 | 1215 | 0 | 32 | 0 |
| 4 | 0 | 0 | 0 | 0 | 1296 | 0 | 0 |
| 5 | 4 | 22 | 49 | 7 | 0 | 1213 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1296 |

**Figure 12.** Confusion matrix with BoW as the input type.



| True \ Predicted | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 1239 | 9 | 1 | 15 | 15 | 11 | 5 |
| 1 | 4 | 1208 | 43 | 13 | 2 | 25 | 0 |
| 2 | 3 | 8 | 1253 | 2 | 0 | 29 | 0 |
| 3 | 6 | 15 | 4 | 1196 | 44 | 28 | 3 |
| 4 | 0 | 0 | 0 | 0 | 1253 | 0 | 43 |
| 5 | 3 | 45 | 46 | 5 | 2 | 1194 | 0 |
| 6 | 0 | 0 | 0 | 0 | 16 | 0 | 1280 |

**Figure 13.** Confusion matrix with TF-IDF as the input type.

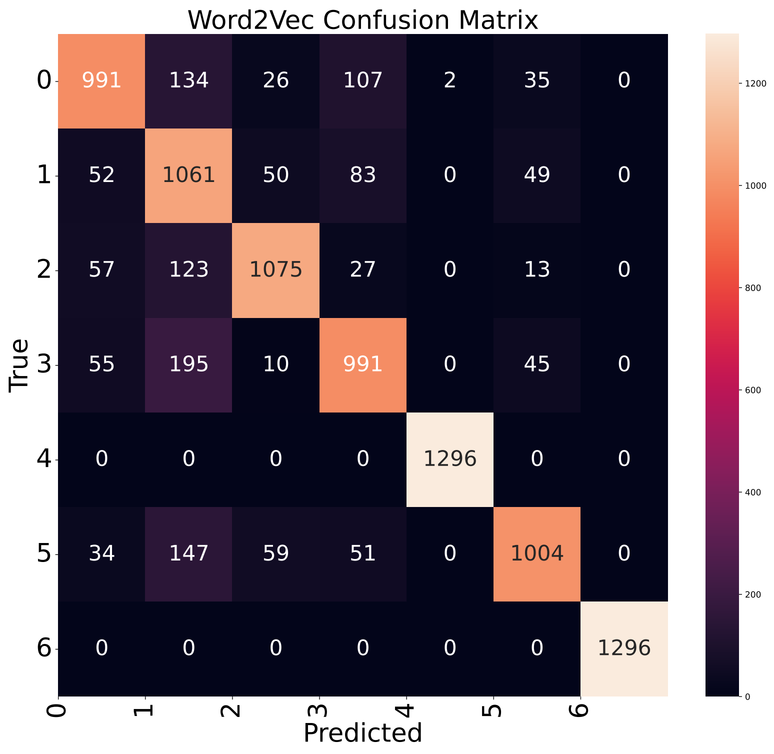


**Figure 14.** Confusion matrix with Word2Vec as the input type.

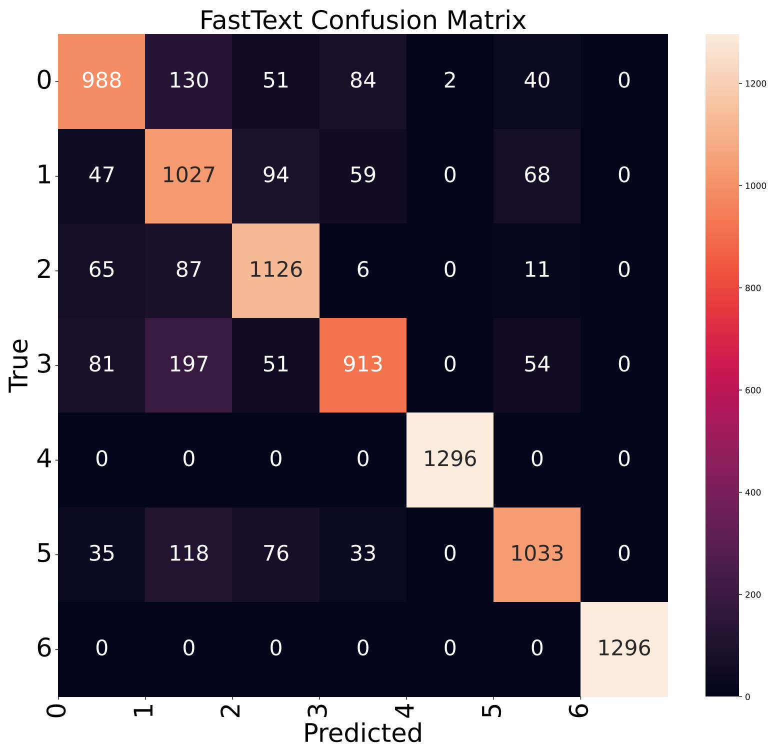


**Figure 15.** Confusion matrix with FastText as the input type.

For Word2Vec, the classifier with Word2Vec performs badly for all types except for delegate and integer overflow. Although the outstanding detection performance for delegate and integer overflow is consistent with that of the binary classification with Word2Vec, the detection performance for other types of vulnerability is noticeably lower than that in binary classification. Furthermore, the detection for integer underflow has the (tied) worst performance in multiclass classification. The result also shows that there is a significant issue of false positives (i.e., 0.235 misclassification rate for the no-vulnerability class).

The result for FastText is very similar to that for Word2Vec, but in general slightly worse. Most notably, the model with FastText struggles to detect integer underflow reliably with a misclassification rate of 0.296.

In summary, in consideration of the results from both binary classification and the multiclass classification, TF-IDF is the only input type that is capable of making excellent detection consistently, demonstrating that this input type can capture the essential characteristics of the six types of vulnerability. Although BoW has the worst overall performance in binary classification, it actually achieves the best performance in multiclass classification. Furthermore, although Word2Vec and FastText exhibit fairly good performance in binary classification, the overall accuracy in multiclass classification is only decent. Closer examination reveals that Word2Vec and FastText have very bad detection performance for timestamp dependency, integer underflow, and CDVA, and they also suffer from an unacceptable rate of false positives.

### *5.3. Comparison with Related Work*

This section compares our results with those reported in previous studies. As we have reported in Table 1, these studies have chosen different sets of smart contract vulnerabilities. Two studies experimented with the detection of a single type of vulnerability, two studies experimented with the detection of a more than a handful of types of vulnerability, and others have experimented with two or three types of vulnerability. All these studies included the re-entrancy vulnerability, and all but two studies included the reentrancy and timestamp dependency vulnerabilities. Some studies examined some types of vulnerability that are not included in our study, such as infinite loop. Our study includes the delegate vulnerability, which is not considered in the set of related studies.

For comparison, we choose to use the F1 score as the metric because all related studies reported this metric. Ideally, we should use exactly the same dataset for comparison. Unfortunately, that is not feasible because some studies do not make their datasets publicly available, and some others do not make their data preprocessing and deep-learning code publicly available. Therefore, our comparison is not meant to be conclusive but, rather, to be a way to gain insight that could guide future development in smart contract vulnerability detection.

The comparison is summarized in Table 6. To save space, we use the following symbols for the types of smart contract vulnerability: R: re-entrancy; T: timestamp dependency; IO: integer overflow; IU: integer underflow; and D: delegate; C: CDAV.

It should be noted that all the related studies we include here used binary classification. Hence, we can only compare the binary classification results. As can be seen in the table, our result (with TF-IDF) ranks the second after SPCBIG-EC by a small margin for re-entrancy vulnerability, and only the middle of the road with TF-IDF for timestamp dependency, where CBGRU reported an outstanding F1 score at 93.29, significantly better than that of other studies. SPCBIG-EC used the Word2Vec as input type, and CBGRU used both the Word2Vec+FastText input types and fused them together. Although CBGRU has an outstanding F1 score for the timestamp dependency vulnerability, one cannot deduce that the graph-based input type is superior to other types of input because it has only reasonably good F1 scores for other types of vulnerability. For CDAV, integer overflow, and integer underflow, we show that, by encoding the smart contract fragments in TF-IDF, we achieve significantly better F1 scores than those of CBGRU. Furthermore, we show that the delegate vulnerability can be detected perfectly with FastText input (TD-IDF comes very close as the second with 99.20%). It might also be interesting to note that, in our study, using a rather simple CNN model with FastText, the F1 scores for the detection of the re-entrancy and timestamp dependency vulnerabilities are only slightly lower than those of DeeSCVHunter [16], which used a fairly sophisticated deep-learning model and three types of input with FastText as the default input type.

**Table 6.** F1 scores for the detection of various vulnerability types in our study and related studies.

| Study | Input | R | T | D | C | IO | IU |
|---|---|---|---|---|---|---|---|
| DeeSCVHunter [16] | FastText (+W+G) | 86.87 | 79.93 | | | | |
| CBGRU [17] | Word2Vec+FastText | 90.92 | 93.29 | | 90.21 | 86.43 | 85.28 |
| Peculiar [18] | Graph | 92.40 | | | | | |
| AME [21] | Graph | 87.94 | 84.10 | | | | |
| BLSTM-ATT [19] | Sequential | 89.81 | | | | | |
| TMP [20] | Graph | 78.11 | 79.19 | | | | |
| DA-GCN [22] | Graph | 85.43 | 84.83 | | | | |
| SPCBIG-EC [24] | Word2Vec | 96.74 | 91.62 | | | | |
| HAM [23] | Word2Vec | 94.04 | 87.85 | | | | |
| This Study | TF-IDF | 95.90 | 85.37 | 99.20 | 95.15 | 94.74 | 95.90 |
| | Word2Vec | 83.48 | 79.55 | 96.77 | 86.76 | 94.74 | 84.73 |
| | FastText | 84.75 | 78.78 | 100.00 | 86.46 | 92.31 | 85.78 |
| | BoW | 78.11 | 74.05 | 99.20 | 78.16 | 87.50 | 74.05 |

## 6. Limitations of the Current Study

The current study is our initial exploration of the impact of different input types on smart contract vulnerability detection performance. An obvious limitation of the current study is that only four input types have been considered. The most notable missing piece is the graph-based input type. Unfortunately, the graph-based input type is highly complex and often heavily customized in each of the related studies. Other embedding methods, such has abstract syntax tree embeddings, hybrid embeddings, and semantic and contextual embeddings, also deserve to be considered. Furthermore, BERT could be used to encode the smart contract fragments. The reason is that large language models such as BERT have shown excellent performance in natural language processing. BERT-based embedding could outperform the four types of input we have experimented.

The second limitation is that the dataset we used is relatively small. In particular, the integer overflow has only 550 fragments. The number of fragments for delegate (980) and re-entrancy (1224) is also relatively small. It is desirable to build a large dataset with at least 10,000 fragments for each type of vulnerability.

Third, some tokens eliminated during the preprocessing stage for the dataset could remove some essential information from the smart contracts. For example, some vulnerabilities could be present only in certain Solidity versions. Because we used a public dataset that has already been preprocessed, we can no longer restore such tokens.

Last, but not the least, it is difficult to provide an in-depth analysis and explanation regarding why TF-IDF has the best overall performance in vulnerability detection, and why BoW has excellent performance in multiclass classification despite the fact that it has poor performance in binary classification. One plausible explanation could be that overfit occurred in some classifications, which could lead to inconsistency or artificial good or bad performance.

## 7. Conclusions and Future Work

In this paper, we systematically studied the impact of four different input types on the vulnerability detection performance using a public dataset. In addition to carrying out binary classification, which is quite pervasive in machine-learning-based vulnerability detection studies, we also conducted multiclass classification experiments. We argued that the vulnerability detection performance of multiclass classification is more useful in practice because it is unlikely to be known beforehand which particular vulnerability the smart contract might have. We experimented with four types of input, namely, Word2Vec, FastText, BoW, and TF-IDF, and six types of vulnerability using a public dataset. We showed that TF-IDF is the overall best performing input type, despite the fact that it is significantly less complex than Word2Vec and FastText, and the input dimension for TF-IDF is also drastically smaller than those of Word2Vec and FastText. Somewhat surprisingly, BoW has slightly better vulnerability detection performance than TF-IDF in multiclass classification (95.84% vs. 95.09%).

There are three desirable future research directions: (1) include more types of input, such as graph-based input, in the comparison; (2) compare the impact of deep-learning models in vulnerability detection; and (3) perhaps most importantly, investigate how to coherently fuse the different types of input for better detection performance. For (3), we attempted to concatenate the feature vectors produced by different types of input and used the combined vector as input for classification. The result is significantly lower classification performance, which proved that this is not the correct way of fusing the input together. To further improve the vulnerability detection performance, a much more sophisticated scheme will have to be developed that could identify and combine complementary features from different types of input.

**Author Contributions:** Conceptualization, I.M.A., W.Z., S.Y., and X.L.; methodology, I.M.A., W.Z., S.Y., and X.L.; investigation, I.M.A., W.Z., S.Y., and X.L.; writing—original draft preparation, I.M.A. and W.Z.; writing—review and editing, I.M.A., W.Z., S.Y., and X.L.; visualization, I.M.A. and W.Z. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported in part by the Beijing Natural Science Foundation under Grants L211020 and M21032.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The dataset used in this study is publicly available at https://github.com/smartbugs/smartbugs-wild, accessed on 1 October 2023.

**Acknowledgments:** The authors wish to express their deep gratitude to the anonymous reviewers for their invaluable comments and suggestions.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. Zhao, W. *From Traditional Fault Tolerance to Blockchain*; John Wiley & Sons: Hoboken, NJ, USA, 2021.
2. Dhillon, V.; Metcalf, D.; Hooper, M.; Dhillon, V.; Metcalf, D.; Hooper, M. The DAO hacked. In *Blockchain Enabled Applications: Understand the Blockchain Ecosystem and How to Make it Work for You*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 67–78.
3. Mehar, M.I.; Shier, C.L.; Giambattista, A.; Gong, E.; Fletcher, G.; Sanayhie, R.; Kim, H.M.; Laskowski, M. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. *J. Cases Inf. Technol. (JCIT)* **2019**, *21*, 19–32. [CrossRef]
4. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* **2018**, arXiv:1810.04805.
5. Kushwaha, S.S.; Joshi, S.; Singh, D.; Kaur, M.; Lee, H.N. Systematic review of security vulnerabilities in Ethereum blockchain smart contract. *IEEE Access* **2022**, *10*, 6605–6621. [CrossRef]
6. Mik, E. Smart contracts: Terminology, technical limitations and real world complexity. *Law Innov. Technol.* **2017**, *9*, 269–300. [CrossRef]
7. Liu, C.; Liu, H.; Cao, Z.; Chen, Z.; Chen, B.; Roscoe, B. Reguard: finding reentrancy bugs in smart contracts. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, Gothenburg, Sweden, 27 May–3 June 2018; pp. 65–68.
8. Wöhrer, M.; Zdun, U. Design patterns for smart contracts in the ethereum ecosystem. In Proceedings of the 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Halifax, NS, Canada, 30 July–3 August 2018; pp. 1513–1520.
9. Atzei, N.; Bartoletti, M.; Cimoli, T. A survey of attacks on ethereum smart contracts (sok). In Proceedings of the Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, 22–29 April 2017; pp. 164–186.

10. Gupta, B.C.; Shukla, S.K. A study of inequality in the ethereum smart contract ecosystem. In Proceedings of the 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Granada, Spain, 22–25 October 2019; pp. 441–449.
11. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv* **2013**, arXiv:1301.3781.
12. Mnih, A.; Kavukcuoglu, K. Learning word embeddings efficiently with noise-contrastive estimation. *Adv. Neural Inf. Process. Syst.* **2013**, *26*.
13. Le, Q.; Mikolov, T. Distributed representations of sentences and documents. In Proceedings of the International Conference on Machine Learning, PMLR, Beijing, China, 21–26 June 2014; pp. 1188–1196.
14. Ramos, J. Using tf-idf to determine word relevance in document queries. In Proceedings of the First Instructional Conference on Machine Learning, Citeseer, 2003; Volume 242, pp. 29–480.
15. Zhao, W. Towards frame-level person identification using Kinect skeleton data with deep learning. In Proceedings of the 2021 IEEE Symposium Series on Computational Intelligence (SSCI), Virtual, 5–7 December 2021; pp. 1–8.
16. Yu, X.; Zhao, H.; Hou, B.; Ying, Z.; Wu, B. Deescvhunter: A deep learning-based framework for smart contract vulnerability detection. In Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN), Shenzhen, China, 18–22 July 2021; pp. 1–8.
17. Zhang, L.; Chen, W.; Wang, W.; Jin, Z.; Zhao, C.; Cai, Z.; Chen, H. CBGRU: A detection method of smart contract vulnerability based on a hybrid model. *Sensors* **2022**, *22*, 3577. [CrossRef] [PubMed]
18. Wu, H.; Zhang, Z.; Wang, S.; Lei, Y.; Lin, B.; Qin, Y.; Zhang, H.; Mao, X. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), Wuhan, China, 25–28 October 2021; pp. 378–389.
19. Qian, P.; Liu, Z.; He, Q.; Zimmermann, R.; Wang, X. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access* **2020**, *8*, 19685–19695. [CrossRef]
20. Zhuang, Y.; Liu, Z.; Qian, P.; Liu, Q.; Wang, X.; He, Q. Smart contract vulnerability detection using graph neural networks. In Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, online, 7–15 January 2021; pp. 3283–3290.
21. Liu, Z.; Qian, P.; Wang, X.; Zhu, L.; He, Q.; Ji, S. Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. *arXiv* **2021**, arXiv:2106.09282.
22. Fan, Y.; Shang, S.; Ding, X. Smart contract vulnerability detection based on dual attention graph convolutional network. In Proceedings of the Collaborative Computing: Networking, Applications and Worksharing: 17th EAI International Conference, CollaborateCom 2021, Virtual Event, 16–18 October 2021; pp. 335–351.
23. Wu, H.; Dong, H.; He, Y.; Duan, Q. Smart contract vulnerability detection based on hybrid attention mechanism model. *Appl. Sci.* **2023**, *13*, 770. [CrossRef]
24. Zhang, L.; Li, Y.; Jin, T.; Wang, W.; Jin, Z.; Zhao, C.; Cai, Z.; Chen, H. SPCBIG-EC: A robust serial hybrid model for smart contract vulnerability detection. *Sensors* **2022**, *22*, 4621. [CrossRef] [PubMed]
25. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. Graphcodebert: Pre-training code representations with data flow. *arXiv* **2020**, arXiv:2009.08366.
26. Hwang, S.J.; Choi, S.H.; Shin, J.; Choi, Y.H. CodeNet: Code-targeted convolutional neural network architecture for smart contract vulnerability detection. *IEEE Access* **2022**, *10*, 32595–32607. [CrossRef]
27. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016.
28. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv* **2016**, arXiv:1603.04467.
29. Qiao, S.; Han, N.; Huang, J.; Yue, K.; Mao, R.; Shu, H.; He, Q.; Wu, X. A dynamic convolutional neural network based shared-bike demand forecasting model. *ACM Trans. Intell. Syst. Technol. (TIST)* **2021**, *12*, 1–24. [CrossRef]
30. Durieux, T.; Ferreira, J.F.; Abreu, R.; Cruz, P. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 27 June–19 July 2020; pp. 530–541.
31. Durieux, T.; Madeiral, F.; Martinez, M.; Abreu, R. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 302–313.
32. Chawla, N.V.; Bowyer, K.W.; Hall, L.O.; Kegelmeyer, W.P. SMOTE: Synthetic minority over-sampling technique. *J. Artif. Intell. Res.* **2002**, *16*, 321–357. [CrossRef]
33. Raschka, S.; Mirjalili, V. *Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow 2*; Packt Publishing Ltd.: Birmingham, UK, 2019.