The Report of CDMC 2019

An analysis on IOT malware classification dataset

NTUST CSIE 蔡湘瑩

September, 2022

Abstract

CDMC 2019 is an IOT malware classification dataset which is provided by Taiwan Information Security Center (TWISC). The goal of this task is to classify IoT malware. The provided features are sequences of the system calls captured during the runtime of malware in a sandbox environment.

To classify the malware, I had to extract the features first. I considered the system calls were a certain kind of language and analyze them with some natural language processing methods. Overall, I chose two strategies, one paying no attention on the context and the other concerning the context. For the former strategy, I used term frequency–inverse document frequency (TF-IDF) analysis which only cared about the frequency of the word. As for the latter, I used word embedding analysis which took the context into consideration when analyzing. When I generated word embedding, both continuous bag-of-words (CBOW) and Skip-gram methods were used. To get the vector of each document, I also applied two ways, arithmetic mean and TF-IDF weighted mean, to calculate it. Therefore, we had five groups of result in total.

After getting the features, I used Random Forest model, K Nearest Neighbor model (KNN), and Support Vector Machine (SVM) to help me classify the malware. The result showed that CBOW analysis performed best and TF-IDF analysis was slightly behind.

Introduction

In the task CDMC 2019, we aimed to classify IOT malware with the provided features. The feature provided were system calls captured when the malware was running. System calls of each malware were packed in a .seq file as Listing 1 shown. The title of a .seq file indicates the sample index in the dataset. Each line in the .seq file stands for a sequence of system calls while the malicious program executed a specific process. The label of each sample was provided in the label file.

```
execve ioctl ioctl prctl gettimeofday getpid gettimeofday getpid fork wait4 SIGCHLD exit EXIT
fork exit EXIT
chdir setuid32 setresuid32 rt_sigaction fork exit EXIT
socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction nanosleep fork exit EXIT
close socket fcntl fcntl connect _newselect getsockopt rt_sigaction
```

Listing 1: Content of 0001.seq

Background

The features provided are all system calls; therefore, I assumed these were able to be viewed as certain kind of languages and decided to use the method of natural languages processing to extract the features. I used two methods to extract the features, TF-IDF (term frequency–inverse document frequency) analysis and word embedding analysis.

The former, TF-IDF analysis, only cares about the frequency of a word in the document and corpus, which means it does not care about order of the words. The higher the frequency of a word in a document, the higher the word scored. On the other hand, the higher the frequency of a word in the corpus, the lower the word scored. The score each word gets implies the importance of the word to the document which it is in.

The latter, word embedding analysis, gives each word a vector to represent the word. This method generates the vector of a word according to the context. The words having similar meaning will have closer vector. In this way, the "meaning" of a word can be reserved in the analysis.

Experiment

1. Extracting features

   I implemented TF-IDF analysis with the help of scikit-learn toolkit. The function TfidfVectorizer() first analyzes the corpus then regularizes these result. The result is made of arrays of TF-IDF value. Each number in an array is the TF-IDF value of the word in the document. 0.0 means the word is not in this document but in the corpus.

```
[0.         0.23942592 0.         0.         0.         0.
 0.01428053 0.         0.         0.         0.         0.20862828
 0.22271239 0.         0.         0.         0.         0.00946265
 0.48369284 0.         0.         0.         0.         0.539705
 0.         0.         0.30477348 0.         0.         0.
 0.         0.         0.         0.         0.         0.01908489
 0.         0.         0.         0.26113529 0.         0.03809059
 0.         0.         0.02124598 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.23631938 0.
 0.         0.         0.         0.01424283 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.22737617 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.03962671 0.         0.         0.         0.
 0.03962671 0.         0.         0.         0.01165226 0.
 0.         0.         0.         0.         0.         0.22249871
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.01175873 0.         0.         0.        ]
```

   Listing 2: 0001.seq after TfidfVectorizer()

   As for word embedding, I chose gensim to help me implement Word2Vec model. I used two strategies to generate the word embedding, CBOW (continuous bag-of-words) and Skip-gram. After I got all the embedding, I also used two strategies. One is arithmetic mean. It simply did addition to vectors of all the words appearing in this document and divided by how many words in it. In this way, we get a simple arithmetic mean vector to stand for the document. The other is TF-IDF weighted mean. When calculating the mean vector of the document, I multiply the TF-IDF value of each word by its vector, and then I would get the TF-IDF weighted mean vector of the document.

2. Training models

   In this part, I needed to train the model to classify the documents within the features I had extracted before, and I still used scikit-learn to help me do the job. For the model to train, I chose Random Forest, KNN (K Nearest Neighbor), and SVM (Support Vector Machine). When evaluating the accuracy, I used 10-fold cross validation to avoid error from imbalance choose of training data.

Result

	TF-IDF	Word embedding			
		CBOW (mean)	CBOW (TF-IDF)	Skip-gram (mean)	Skip-gram (TF-IDF)
Random Forest	98.3441%	98.0319%	98.0320%	97.6719%	97.7922%
KNN	96.9042%	96.8566%	96.8086%	96.5203%	96.4724%
SVM (linear)	93.0890%	95.6568%	96.0166%	92.2249%	92.7046%
SVM (poly)	93.7125%	93.4247%	93.1845%	93.0166%	91.6488%
SVM (RBF)	93.5205%	94.0488%	93.9528%	92.8009%	91.7449%

Table 1: accuracy score (by percentage) of all experiment

As Table 1 shown, CBOW analysis has the best performance, and TF-IDF analysis is slightly behind. Skip-gram analysis performs worst, especially when it is weighted by TD-IDF value. When I classified these documents with Random Forest and KNN, the result is all similar no matter how I had extracted the features. Also, it is interesting that when I classified the documents with SVM (linear) model, only CBOW analysis' performance is outstanding. The similar phenomenon appears when I classified the documents with SVM (RBF) model.

Conclusion

In this task, I tried some methods of natural language processing to analyze system calls and classify the malware. The two major concept I applied were analyzing with or without consideration of the context. It was not surprising word embedding analysis outperformed pure TF-IDF analysis, for system calls were presented in ascending order of the creation time, which implied the context might be important. However, it surprised me that pure TF-IDF analysis surpassed the performance of word embedding analysis when I chose Skip-gram strategy. I assumed that the properties of those system calls, such as lots of repetitive system call and similar calling pattern, resulted in the consequence that TF-IDF analysis also performed well.