Tagging Malware Intentions by Using Attention-Based Sequence-to-Sequence Neural Network

Yi-Ting Huang[1](✉), Yu-Yuan Chen[2], Chih-Chun Yang[2], Yeali Sun[2], Shun-Wen Hsiao[3], and Meng Chang Chen[1,4]

[1] Institute of Information Science, Academia Sinica, Taipei, Taiwan
ythuang@iis.sinica.edu.tw
[2] Information Management, National Taiwan University, Taipei, Taiwan
[3] Management Information Systems, National Chengchi University, Taipei, Taiwan
[4] Research Center of Information Technology Innovation, Academia Sinica, Taipei, Taiwan

Abstract. Malware detection has noticeably increased in computer security community. However, little is known about a malware's intentions. In this study, we propose a novel idea to adopt sequence-to-sequence (seq2seq) neural network architecture to analyze a sequence of Windows API invocation calls recording a malware at runtime, and generate tags to describe its malicious behavior. To the best of our knowledge, this is the first research effort which incorporate a malware's intentions in malware analysis and in security domain. It is important to note that we design three embedding modules for transforming Windows API's parameter values, registry, a file name and URL, into low-dimension vectors to preserve the semantics. Also, we apply the attention mechanism [10] to capture the relationship between a tag and certain API invocation calls when predicting tags. This will be helpful for security analysts to understand malicious intentions with easy-to-understand description. Results demonstrated that seq2seq model could mostly find possible malicious actions.

Keywords: Malware analysis · Dynamic analysis · seq2seq neural network

1 Introduction

A malware, such as virus, Internet worm, trojan horse, and botnet, has been a main challenge in computer security, because it may disrupt infected network service, destroy software or data, steal sensitive information, or take control of the host. Thus, malware detection and malware classification have been widely investigated [1–5, 9].

Anti-virus products have primarily concerned with malware individual signatures to detect a malware. However, more recently, with the development of obfuscation techniques and the prevailing access to open source tools, it has been easy to create variants of a malware so that it has been greatly increase the number of malwares. Thus, rather than detecting malware individual signatures, we shifted our attention to analyze malware behavior. If malicious characteristics can be caught, they can be the

J. Jang-Jaccard and F. Guo (Eds.): ACISP 2019, LNCS 11547, pp. 660–668, 2019.
https://doi.org/10.1007/978-3-030-21548-4_38

basis of malware detection. This approach will increase the effectiveness of malware detection and decrease the operating cost of it.

As far as we know, there is no benchmark for malicious characteristics, because it is a challenge to examine the infected systems, system logs and malware binaries, and understand any possible intention. Windows APIs, an access to system resources, can be another resource to reveal malware behavior when a malware is executing. Thus, we will hook the Windows API functions at the virtualization layer to intercept the targeted malware at the runtime and record its invoked API calls.

In this paper, we propose a neural sequence-to-sequence (seq2seq) model, which analyzes a sequence of Windows API invocation calls, and labels subsequences of Windows API invocation calls with tags. These tags can be used to explain malicious intentions of a malware. This study will lead to a better understanding of malware characteristics in malware analysis. This paper makes the following contributions:

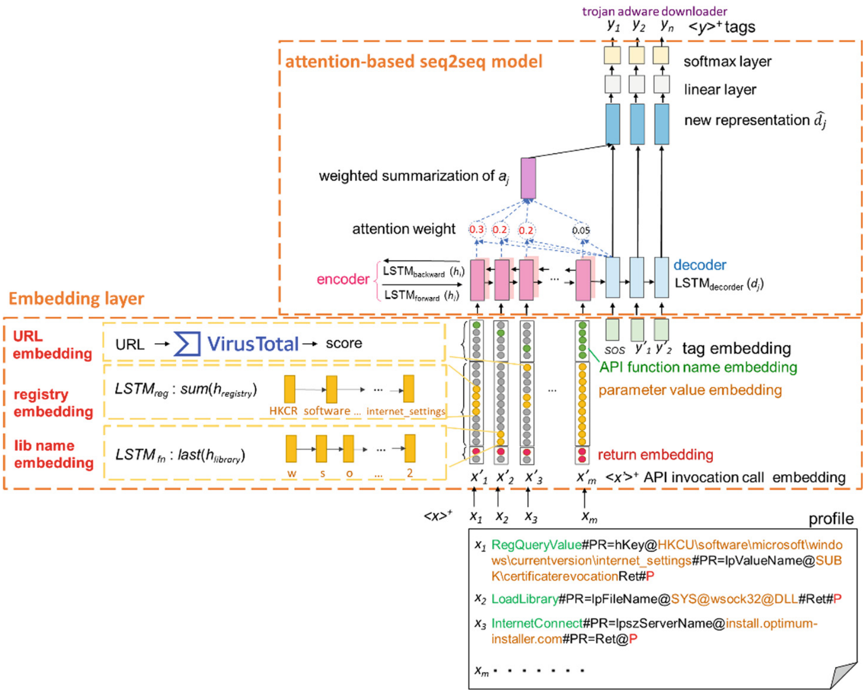
- We apply a neural network model, which predicting one or more tags, to describe malicious intentions of a malware.
- We propose approaches for transforming a Windows API invocation call into a numeric vector (embedding).

2 System Design

In this paper, our goal is to employ neural network technology to construct an automatic malware tagging system by analyzing a large set of malware samples. When given a malware sample, the system can output a list of tags which can truly capture the essence of the series of activities performed by the malicious program.

Figure 1 depicts the overall architecture of the model. The model consists of an embedding layer and an attention-based seq2seq model. Because the execution traces generated from the dynamic malware behavior profiling system are text files, we need to transform the plaintext representations of the API invocation calls into vectorized representations. Thus, an embedding layer, consisting of API function name embedding, parameter value embedding and return value embedding, takes a variable-length execution trace $x = \{x_1, \ldots, x_m\}$ as the input and outputs a sequence of embedding vectors $x' = \{x'_1, \ldots x'_m\}$. Some parameter types may have numerous categorical values, such as Registry. It is compute-inefficient to model them all in one-hot encoding format. Thus, Three embedding modules – registry value embedding, library name embedding, and URL embedding – are proposed, and will be explained.

A sequence-to-sequence (a.k.a. encoder-decoder) model is a neural network architecture which consists of an encoder and a decoder. Long Short-Term Memory (LSTM) [7] is used for sequence processing from a sequential input. A bi-directional encoder $BiLSTM_{encoder}$ processes a sequence of variable-length embedding vector $x' = \{x'_1, \ldots x'_m\}$ from forward and backward simultaneously, and outputs a series of vector representation $h = \{h_1, \ldots, h_m\}$. A decoder $LSTM_{decoder}$ is conditioned on the output h from the encoder to generate a hidden state d_j. One key component of the model is to connect subsequences of API invocation calls to an individual tag. For example, a code subsequence directly reflects the operation of self-propagation, i.e., tag "*worm.*" Hence, we use an attention mechanism to establish such relationships. We would like to pay special **attention** to the relevant subsequence as we tag. The decoder at each time step focuses on a different part of the input trace to gather the semantics information in order to generate proper tag. This attention weights are computed by the current hidden state d_j from the decoder and all hidden state h_i from the encoder. With the attention weights, we can obtain a weighted summarization a_j of the hidden vectors from the encoder. A new representation $\hat{d}_j$ is the concatenation a_j and d_j for calculate the probability distribution over tags. Finally, a linear layer projects the new presentation $\hat{d}_j$ into a prediction layer, and a *softmax* layer computes the tag distribution. The predicated tag y_j is the target class with the highest probability. More details can be found in [10].



```
x1 RegQueryValue#PR=hKey@HKCU\software\microsoft\windo
   ws\currentversion\internet_settings#PR=lpValueName@SUB
   K\certificaterevocationRet#P
x2 LoadLibrary#PR=lpFileName@SYS@wsock32@DLL#Ret#P
x3 InternetConnect#PR=lpszServerName@install.optimum-
   installer.com#PR=Ret@P
xm . . . . . . .
```

Fig. 1. When given a profile which contains $x_1 \ldots x_m$, the proposal system transforms a profile $x = \{x_1 \ldots x_m\}$ into embedding vector $x'_1, \ldots x'_m$, and predicts a list of tags $y = \{y_1 \ldots y_n\}$ by capturing the relations between each tag y_j and input sequence $x = \{x_1 \ldots x_m\}$.

2.1 The Embedding Layer

The goal of the embedding layer is to produce a fixed-size vector as the corresponding embedding x' when given a Windows API invocation call x. An API invocation call x_i consists of an API function name w_i, one or more parameter values v_i, none or one return value ret_i. Each element, x_i, is transformed as an embedding x'_i as a concatenation of a function name embedding w'_i, parameter embeddings v'_i and a return embedding ret'_i. Each element learns its identical weighted embedding matrix E.

$$x'_i = concate\left(E_w w_i, concate_k^{|pr|}(E_k v_{ik}), E_{ret} ret_i\right) \tag{1}$$

where $E_w \in R^{e_w \times |w|}, E_k \in R^{e_k \times |k|}, E_{ret} \in R^{e_{ret} \times |ret|}$ are function name, parameter, and return embedding matrices; e_w, e_k, and e_{ret} are embedding sizes respectively.

We focus on three resources and propose respective approaches to transform a parameter value into a low-dimension vector while preserving the semantics. The rest of input values, including API function names, the other parameter values, and the return value are initialized by drawing samples from a uniform distribution within Xavier initializer [6] and learn their own embedding matrices as well.

Registry Value Embedding. In Windows system, registry contains important configuration information for the operating system, services, applications and user settings. Therefore, registry-related operations are important in malicious behavior analysis and the parameter values are especially critical. The Windows registry is a hierarchical database which includes keys, subkeys and values. The structure of registry keys is similar to that of folders in the file system and they are referenced with a syntax similar to Window's path, using backslashes to indicate levels of hierarchy. Thus, we construct a registry value embedding module to tokenize keys with a backslash, '\', and then use a LSTM unit referred to as the $LSTM_{reg}$ to transform a key denoted by $key = \{key_1, ..., key_n\}$ into hidden vectors $h_{key1}...h_{keyn}$. All hidden vectors are then summed to a registry representation v_{reg}. For example, key "*HKCU\software\microsoft\windows\currentversion\internet_settings*" contains six tokens - "*HKCR", "software*", "*microsoft*", "*windows*", "*currentversion*" and "*internet_settings*." Each token is an input to the LSTM unit. The output hidden vectors constitute the registry key representation, i.e., $h_{HKCU\backslash software\backslash microsoft\backslash windows\backslash currentversion\backslash internet\_settings} = h_{HKCR} + h_{software} + h_{microsoft} + h_{windows} + h_{currentversion} + h_{internet\_settings}$. Therefore, we could preserve the hierarchical relation between tokens and have a fixed and consistent embedding dimension regardless of the number of keys.

File Name Embedding. From our analysis of malware operations, malwares often code file names where the spellings deform some familiar regular names to obfuscate the intent, e.g., "*2dvaai32.dll*" vs. "*advapi32.dll*". There are also file names which comprise a file name and a random number, e.g., "*tsu08c6ec63.dll*" and "*tsu0ac63fe4.dll*". Some file names are generated from a hash value, e.g. "*518ca2bf37e13.dll*". In other words, any possible combinations for a file name are enormous and unpredictable. Here, we separate the file name into a sequence of character strings $\{c_1, ..., c_n\}$ and input each character string to a $LSTM_{fn}$ unit one by one and obtain the corresponding hidden vectors $\{h_{c1}, ..., h_{cn}\}$. The last hidden state h_{cn} is considered as file name representation v_{lib}. For example, a file name "*wsock32*", can be split into a series of characters, $\{w, s, ..., 2\}$. Each letter is an input to the $LSTM_{fn}$ unit. They are transformed into the associated hidden vectors, i.e., $h_{wsock32} = \{h_w, h_s, \ldots, h_2\}$ and h_2 can be considered as the file name representation for '*wsock32*'. The merit of the proposed LSTM unit is that it can capture the similarities between purposely obfuscated file names or different variations of the same file name while treating each individually.

URL Embedding. Malware programs often include codes that visit remote malicious web sites in background and gain control of a computer system without being detected. However, it is difficult to literally distinguish whether a URL is malicious or not. Nonetheless, we consider URLs are important part of the information about the program's operations. Specifically, we make use of the URL reports from *VirusTotal*[1] which give the result of the ratio of number of antivirus engines that detected a scanned URL is malicious or not. This ratio is used as the score for embedding URL. For instance, a URL, "*install.optimum-installer.com*", got six over sixty-six. Since the score is a real number, the associated embedding E_{URL} is an identity matrix of 1×1.

3 Evaluation

3.1 Dataset

To capture the essentials of the execution behavior of a malware program, we used an automated dynamic malware behavior profiling and analysis system based on Virtual Machine Introspection (VMI) technique [8]. We carefully-selected 28 Windows API calls, shown in Table 1. A malware sample may create or fork one or more processes. An execution trace is generated per process. Some distinct malwares with the same intent have slightly different parameter values, such as the user-profile folders, "*user's Desktop*" and "*user's Documents*", depending on the version of operating systems or their executable strategy. To reduce this noise, values relevant to file directory and registry key are symbolized, details in [2]. Also, a trace is reformatted and present a Windows API call line by line, as a profile illustrated in Fig. 1.

We collected 19,987 malware profiles from11,939 samples, acquired from NCHC's OWL[2] project. Since a few profiles contained too many API calls or too few ones, we excluded the samples whose number of API calls were smaller than 10, or larger than 300. The final dataset includes 14,677 profiles from 9,666 samples.

In order to compile a set of tags which are descriptive terms to help users quickly grasp the characteristics of a malware program, we crawled labels from *VirusTotal* in April, 2018. The labels were changed to lowercase and tokenized by delimiters, "\,|!|\(|\)|\[|\]|@|:|/|\.|_|\-|." Only the first and second tokens are considered. We manually build an alias table for the tokens with same meaning. For example, "*troj*" and "*trj,*" are the abbreviations of "*trojan.*" Seventy-six tags are compiled, shown in Table 2. We relabeled the tags for each malware sample. If a sample has any child process file, it is labeled with the same tags as the main process. We also sorted tags in descending order by counting occurrences in order to control the variance from the order of tags. We hope a tag with a highly frequent occurrence could be predicted first.

[1] https://www.virustotal.com.

[2] https://owl.nchc.org.tw.

We randomly divide the dataset into a training set (80%), a development set (10%), and a testing set (10%). Distributions of the three sets are then validated by F-test until none of them have no significant difference. We report results on the testing set.

Table 1. Summary of Windows API function name and parameter types used in this study.

Category	API function name	Parameter type
Registry	RegCreateKey, RegDeleteKey, RegSetValue, RegDeleteValue, RegOpenCurrentUser^{+}, RegEnumValue, RegQueryValue	hKey, lpSubKey, lpValueName
Process	CreateProcess, CreateRemoteThread, CreateThread, TerminateProcess, ExitProcess*, OpenProcess^{+}, WinExec^{+}	lpApplicationName, dwCreationFlags, uExitCode
Network	InternetOpen^{+}, WinHttpConnect, InternetConnect, WinHttpOpen^{+}, WinHttpOpenRequest^{+}, WinHttpReadData^{+}, WinHttpSendRequest^{+}, WinHttpWriteData^{+}, GetUrlCacheEntryInfo^{+}, HttpSendRequest^{+}	lpszServerName, pswzServerName, nServerPort
Library	LoadLibrary	lpFileName
File	CopyFile, CreateFile, DeleteFile	lpFileName, lpExistingFileName, lpNewFileName, dwCreationDisposition, dwDesiredAccess, dwShareMode

*Only "ExitProcess" has no return value.
$^{+}$Its associated parameter values are not considered.

Table 2. Seventy-six tags are collected.

Categories	Tags
Type or family	Adware, backdoor, bot, browsermodifier, bundler, ddos, game, grayware, networm, PUP*, ransom, riskware, rootkit, spyware, trojan, virus, worm
Behavior	Autorun, binder, browserhijacker, clicker, crypt, dialer, dns, downloader, dropper, fakealert, fakeav, filecryptor, fileinfetor, flooder, fraudtool, hacktool, infostealer, installer, joke, keylog, lockscreen, memscan, monitor, packed, prochollow, procinject, virtool, webfilter
Route of infection	Air, email, im, p2p, patch, pdf, proxy, sms, uds
Programming lang.	Autoit, bat, html, js, php, vb
Other	Android, apt, avt, constructor, exploit, FAT*, fca, hllp, iframe, irc, keygen, MBR*, MSIL*, password, PE*, rat

*These tags are changed to uppercase in order to be understood easily.

3.2 Experimental Settings

We set the LSTM hidden unit size to 256 and the number of layers of LSTMs to 2 in both the encoder and the decoder. Optimization is performed using Adam optimizer, with an initial learning rate of 0.0002 for the encoder, and 2.5 for the decoder. Training runs for 600 epochs. We start halving the learning rate at epoch 300, and decay it per 100 epoch. The mini-batch size is set at 16. Dropout with probability is 0.1.

Two baselines and three input variations are examined to answer two questions: (1) is the seq2seq model suitable for our task? (2) Can the return embedding or the parameter embedding help models to predict tags? We reproduced Convolutional Neural Network (CNN) [11] and Multi-label Multi-class Classification (MLC) as baselines. Both models use the proposed embedding layer. While the CNN has three convolution layers (256, 192, 64) with an average pooling layer, MLC has the same encoder as the proposed system. Lastly, both connect to a dense layer and a sigmoid layer.

For each model, three input variations – only API function names, add the associated return values, and add the corresponding parameter values – were evaluated. To ensure that performance is not simply due to an increase in the number of model parameters, we keep the total size of the embedding layer fixed to 256. The size of the return embedding and the parameter embedding are set to 2 and 50 respectively, and the size of the function name embedding is set to bring the total size to 256.

Recall are preferably used as our metric because it means malicious patterns could be mostly found, which could help security analysis. Precision is also reported.

3.3 Results

Table 3 presents the results among different models and input settings. The results showed an obvious effect of models on recall and precision. With respect to recall, the predictions from the seq2seq models relatively approximated the ground truth. On the other hand, regarding to precision, the percentage of tags from the MLC models correctly predicted was highest, but the average number of predictions was much less than the number of ground truth (7.41). We compared the predicted tags from the MLC models against that from the seq2seq models. It showed that 84% of tags from the MLC models and 52% from the seq2seq models were the same.

Table 3. Experiment result between different models.

Model	Input setting	Recall	Precision	|Predicted tags|
CNN	API name	42.72%	69.32%	4.47
	API name + return	40.35%	72.27%	4.06
	API name + parameter + return	40.82%	69.63%	4.30
MLC	API name	45.50%	72.39%	4.46
	API name + return	44.66%	74.10%	4.34
	API name + parameter + return	46.40%	72.10%	4.63
seq2seq	API name	**57.10%**	53.02%	7.92
	API name + return	56.25%	52.39%	7.86
	API name + parameter + return	**57.18%**	53.05%	7.88

For each model, three input variants had slightly different and inconsistent results except for the seq2seq models. For seq2seq, considering all of input settings, including API function names, parameter values, and return values, had minor better performance than only considering API function names. It was surprised the performance was worse when considering API function names and return values. It implies we could analyze malware without knowing it was successful or not. We anticipate the tagging is related to attack intention, rather than the successfulness of the API call invocation.

4 Conclusion

In this paper, we present a novel neural seq2seq model to analyze Windows API invocation calls and predict tags to label a malware's intentions. Results showed that the seq2seq model, with all of input values, API function names, the associated return values, and the corresponding parameters, could find mostly possible malicious characteristics with respect to the number of prediction and the high ratio of correctly predicted tags to ground truth. This can help security experts to understand any potential malicious intentions with easy-to-understand description.

Acknowledgements. This work was supported by MOST107-2221-E-004-003-MY2.

References

1. Athiwaratkun, B., Stokes, J.W.: Malware classification with LSTM and GRU language models and a character-level CNN. In: 2017 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 2482–2486. IEEE, New Orelans (2017)
2. Chiu, W.J.: Automated malware family signature generation based on runtime API call sequence. Master thesis. National Taiwan University, Taiwan (2018)
3. Dahl, G.E., Stokes, J.W., Deng, L., Yu, D.: Large-scale malware classification using random projections and neural networks. In: Acoustics, Speech and Signal Processing, pp. 3422–3426. IEEE, Vancouver (2013)
4. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. **44**(2), 6 (2012)
5. Gandotra, E., Bansal, D., Sofat, S.: Malware analysis and classification: a survey. J. Inf. Secur. **5**, 56–64 (2014)
6. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Thirteenth International Conference on Artificial Intelligence and Statistics, pp. 249–256 (2010)
7. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
8. Hsiao, S.W., Sun, Y.S., Chen, M.C: Virtual machine introspection based malware behavior profiling and family grouping. arXiv preprint arXiv:1705.01697 (2017)
9. Huang, W., Stokes, J.W.: MtNet: a multi-task neural network for dynamic malware classification. In: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 399–418. Springer, Cham (2016)

10. Luong, M.T., Pham, H., Manning, C.D.: Effective approaches to attention-based neural machine translation. In: Proceedings of Conference on Empirical Methods in Natural Language Processing, pp. 1412–1421. Lisbon, Portugal (2015)
11. Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., Torralba, A.: Learning deep features for discriminative localization. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2921–2929. (2016)