Malicious Powershell Detection Using Graph Convolution Network

Sunoh Choi

Department of Software Engineering, Jeonbuk National University, Jeonju 54896, Jeollabuk-do, Korea; suno7@jbnu.ac.kr

Abstract: The internet’s rapid growth has resulted in an increase in the number of malicious files. Recently, powershell scripts and Windows portable executable (PE) files have been used in malicious behaviors. To solve these problems, artificial intelligence (AI) based malware detection methods have been widely studied. Among AI techniques, the graph convolution network (GCN) was recently introduced. Here, we propose a malicious powershell detection method using a GCN. To use the GCN, we needed an adjacency matrix. Therefore, we proposed an adjacency matrix generation method using the Jaccard similarity. In addition, we show that the malicious powershell detection rate is increased by approximately 8.2% using GCN.

Keywords: powershell; graph convolution network; adjacency matrix

1. Introduction

The internet’s rapid growth makes it a source of useful information for many people; however, the number of malicious files circulated is also increasing. According to AV-TEST [1], hundreds of thousands of new malicious files are created every day and approximately one billion malicious files are currently available. Malicious files include document-based files, powershell scripts, and Windows PE files. Powershell scripts are not downloaded to a user’s computer hard disk; they are directly downloaded to their computer memory and executed. Therefore, it is challenging for existing file-based anti-virus solutions to detect powershell scripts [2,3]. However, recent progress in AI techniques enable their use to recognize images and process natural languages [4,5]. In addition, AI has been used in research to detect malicious files [6,7], including malicious powershell [2,3].

For example, convolution neural network (CNN) techniques are used for image recognition [4] and recurrent neural network (RNN) techniques are used for natural language processing [5]. Recently, a graph convolution network (GCN) was proposed [8]. Figure 1 shows that in the GCN, there are nodes and links. Each node possesses feature data and adjacent nodes that are connected to it through the links. Each node also possess features such as those of the adjacent nodes. By using GCN, each node obtains additional features from the adjacent nodes. In social network services, GCNs are used for friend or item recommendations [9].

Figure 1 shows an example of a GCN recommendation system [10]. Each node represents a user and includes a feature list of their expressed interests as well as a label indicating their gender. Moreover, each user is connected to other users. For example, a node representing a user Alice indicates that she is labeled as a woman interested in clothes and cosmetics and connected to Barbie, Camilla, Daisy, and Bob. Similarly, Bob is labeled as a man whose recorded interests include cars and baseball and he is connected to Adam, Charles, Dave, and Alice.

Since Camilla is interested in clothes, cosmetics, and cooking and she is a friend of Alice, cooking may be recommended to Alice as a potential interest. Since Charles is interested in cars and games and he is Bob’s friend, games can be recommended to Bob as well.
Figure 1. Graph Convolution Network (GCN) for Recommendation System.

In addition, because Alice is identified in the system as a woman and Camilla is her friend with a feature list similar to Alice’s, the graph may indicate that Camilla may be identified as a woman with a high probability. Similarly, because Bob is labeled as a man and Charles’s feature list is similar to Bob’s list, the graph shows a high probability that Charles may be identified as a man.

As shown in this example of a recommendation system, GCNs consider the feature lists of other nodes to determine the labels of any given node. This advantage can be adapted to an AI-based malware detection system. Existing malware detection systems generally determine whether a file is malicious by considering only its own feature list [2,3]. However, by using GCNs in malware detection, we can use the features of other files as well as a file’s own features to determine whether it is malicious.

Here, we propose a new method for detecting malicious powershells using GCN. We increase the malicious powershell detection rate by using GCN when the new powershell is similar to an existing powershell scripts. First, we extract the feature data from the powershell scripts. Second, we compute the Jaccard similarities between the new powershell and existing powershell scripts. Third, we generate an adjacency matrix using Jaccard similarities [11]. Finally, we determine whether the new powershell is malicious using the GCN. In the experiments, we show that the malicious powershell detection rate is increased.

The remainder of this paper is organized as follows. In Section 2, we introduce the related work. In Section 3, we present the GCN and propose a new method for detecting malicious powershells using GCN. In Section 4, we present the experimental results and in Section 5 we provide the discussion.

2. Related Work

AI-based malicious file detection involves two steps. The first step involves extracting feature data from the files. The second step involves training the AI model for malicious file detection using feature data [5,6].

Feature data can be extracted by two methods. The first is to use a static analysis [6] and the second is to use a dynamic analysis [7]. Static analysis extracts feature data from the string information of the file. In the PE files, we used tokens of assembly codes of PE files as feature data. In powershell scripts, there are 20 types of tokens and we use these tokens as feature data. However, if a file is encrypted or encoded, it is difficult to analyze.

Dynamic analysis uses system call information as the feature data after we run a file. It analyzes encrypted or encoded files. However, it is not executed in a virtual machine environment and it takes a long time to analyze because it must run for several minutes for each file.

There are two models in the AI model for malicious file detection. The first uses a CNN model that is mainly used for image recognition [6], while the second uses an RNN model that is mainly used for natural language processing [7]. The first method involves...
transforming a file into an image. Eight bits can be transformed into gray image pixels. Then, we can determine whether it was a malicious file image using a CNN. The second method involves transforming a file into a sentence. Afterwards, we determine whether it was a malicious sentence using an RNN.

In addition, studies have been conducted to detect malicious powershells. They extracted feature data from the powershell used static analysis and detected malicious powershells by using the CNN and RNN models in combination [2,3]. Using the PSParse library, they extracted token data from powershell scripts and used them as feature data. In related research, six types of tokens were used as feature data from a total of 20 types of tokens [3].

There have been studies on detecting malicious PE files using GCN [12,13]. These studies generated an adjacency matrix from the system call graph of a PE file and determined whether it was malicious. However, in the current study, we generated an adjacency matrix using Jaccard similarities between powershell scripts.

By contrast, trace abstraction was proposed in [14]. The authors used a longest common subsequence (LCS) technique to determine whether two program traces were similar. However, since LCS methods require a relatively long processing time, they are not appropriate for malware detection, which should be performed quickly. In addition, Func2Vec was proposed in [15]. This method generated sentences using a random walk over a control-flow graph to find function synonyms. However, in this study, generating sentences from a control-flow graph is not the subject of our research.

3. Malicious Powershell Detection Method Using Graph Convolution Network (GCN)

In this section, we propose a new method for detecting malicious powershell scripts using a GCN. First, in Section 3.1, we introduce GCN. Second, in Section 3.2, we propose a method to generate an adjacency matrix using Jaccard similarity between powershell scripts and provide a method to detect malicious powershell scripts using the adjacency matrix.

3.1. Graph Convolution Network (GCN)

GCN was proposed in [8]. GCN had a \( n \times d \) feature matrix \( X \), a \( n \times n \) adjacency matrix \( A \), and a \( d \times m \) weight matrix. The GCN was defined as follows the following.

\[
H = \sigma(AXW)
\]

Feature matrix \( X \) and adjacency matrix \( A \) were the input data of the GCN and \( H \) was the output of the GCN. \( \sigma \) was the activation function. By training the GCN, we updated \( W \).

Figure 2 shows that when a graph was given, the feature matrix \( X \) was the following.

\[
X = \begin{bmatrix}
x_{11} & x_{12} & x_{13} \\
x_{21} & x_{22} & x_{23} \\
x_{31} & x_{32} & x_{33} \\
x_{41} & x_{42} & x_{43} \\
x_{51} & x_{52} & x_{53} \\
x_{61} & x_{62} & x_{63} \\
x_{71} & x_{72} & x_{73} \\
x_{81} & x_{82} & x_{83} \\
\end{bmatrix}
\]

In this case, \( N \) was the number of nodes and equal to 8, \( d \) was the number of features of each node and equal to three. If \( x_{ij} \) was equal to 1, it meant that the \( i \)-th node had the \( j \)-th feature and if \( x_{ij} \) was equal to 0, it meant that it did not have the feature.
The adjacency matrix $A$ was as follows:

$$A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\
a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\
a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\
a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\
a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\
a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\
a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\
a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88}
\end{bmatrix} = \begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 
\end{bmatrix}$$

If $a_{ij}$ is equal to 1, the $i$-th node was adjacent to the $j$-th node. If $a_{ij}$ was equal to zero, the $i$-th node was not adjacent to the $j$-th node.

The $d \times m$ weight matrix was as follows:

$$W = \begin{bmatrix}
w_{11} & w_{12} \\
w_{21} & w_{22} \\
w_{31} & w_{32}
\end{bmatrix}$$

where $m$ was the number of output classes. Let $S$ be $XW$ and $S$ is described in the following.

$$S = \begin{bmatrix}
x_{11} & x_{12} & x_{13} \\
x_{21} & x_{22} & x_{23} \\
x_{31} & x_{32} & x_{33} \\
x_{41} & x_{42} & x_{43} \\
x_{51} & x_{52} & x_{53} \\
x_{61} & x_{62} & x_{63} \\
x_{71} & x_{72} & x_{73} \\
x_{81} & x_{82} & x_{83}
\end{bmatrix} \begin{bmatrix}
w_{11} & w_{12} \\
w_{21} & w_{22} \\
w_{31} & w_{32}
\end{bmatrix} = \begin{bmatrix}
S_{11} & S_{12} \\
S_{21} & S_{22} \\
S_{31} & S_{32} \\
S_{41} & S_{42} \\
S_{51} & S_{52} \\
S_{61} & S_{62} \\
S_{71} & S_{72} \\
S_{81} & S_{82}
\end{bmatrix}$$

Then, we computed the following.

$$H = \sigma(AS) = \sigma(\begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}) = \begin{bmatrix}
S_{11} & S_{12} \\
S_{21} & S_{22} \\
S_{31} & S_{32} \\
S_{41} & S_{42} \\
S_{51} & S_{52} \\
S_{61} & S_{62} \\
S_{71} & S_{72} \\
S_{81} & S_{82}
\end{bmatrix}$$
Here, $h_{71}$ and $h_{72}$ were computed as follows:

$$h_{71} = \sigma (s_{31} + s_{41} + s_{61} + s_{81})$$
$$h_{72} = \sigma (s_{32} + s_{42} + s_{62} + s_{82})$$

This meant that the output of node $n_7$ depended on the adjacent nodes, such as $n_3$, $n_4$, $n_6$, and $n_8$.

The $n \times m$ output was as follows.

$$H = \begin{bmatrix}
    h_{11} & h_{12} \\
    h_{21} & h_{22} \\
    h_{31} & h_{32} \\
    h_{41} & h_{42} \\
    h_{51} & h_{52} \\
    h_{61} & h_{62} \\
    h_{71} & h_{72} \\
    h_{81} & h_{82}
\end{bmatrix}$$

Note that in the malicious powershell detection problem, the output was normal or malicious. Using the GCN, we determined each node’s class. For example, in a karate club network [16], they determined an unlabeled club member’s class using the GCN.

However, the adjacency matrix did not contain any node itself. We added an identity matrix $I$ to it as follows.

$$\tilde{A} = A + I$$

Then, we normalized it as follows.

$$H = \sigma \left( \tilde{D}^{-1/2} \tilde{A} \tilde{D}^{1/2} X W \right)$$

Note that $\tilde{D}$ was a degree matrix [17] of $\tilde{A}$.

### 3.2. Malicious Powershell Detection Method Using Adjacency Matrix from Jaccard Similarity

Figure 3 shows malicious powershell detection using a GCN. We attempted to generate an adjacency matrix using the Jaccard similarity between the powershell scripts. By using GCN, we could use adjacent node features as well as its own features to determine whether it was malicious. We expected an increase in the detection rate of malicious powershells.

![Figure 3. Malicious powershell detection using GCN.](image)

The powershell had 20 types of tokens described as follows [3].

- Attribute, Command, CommandArgument, CommandParameter, Comment, GroupEnd, GroupStart, Keyword, LineContinuation, LoopLabel, Member, NewLine, Number, Operator, Position, StatementSeparator, String, Type, Unknown, Variable}
We used 6 types of tokens for the feature data, which is described as follows.

\{Command, CommandArgument, CommandParameters
Keyword, Member, Variable\}

Note that in a previous study [3], we conducted many experiments using various combinations of token types and found that the best performance was exhibited when we used 6 token types.

In step 1, we generated feature lists by extracting the feature data from the powershell scripts. We extracted approximately 20,000 unique tokens from the powershell scripts. In each powershell script, if the \(j\)-th token of the \(i\)-th powershell script existed, we set \(t_{ij}\) to 1. Otherwise, it was set to 0. Note that \(d\) was 20,000. The following is described.

\[i - \text{th Powershell's Feature List } F_i = \{t_{i1}, t_{i2}, \ldots, t_{id}\}\]

Note that we considered 1000 powershell scripts, including 3780 unique tokens in 6 token types. We used all unique tokens. However, we set \(d\) to 20,000 for scalability in future work.

We had \(n\) powershell scripts and each feature list was generated from each powershell using PSParse [18]. Then, we generated an \(n \times d\) feature matrix \(X\) from \(n\) feature lists described as follows.

\[X = \begin{bmatrix}
  t_{11} & \cdots & t_{1d} \\
  \vdots & \ddots & \vdots \\
  t_{n1} & \cdots & t_{nd}
\end{bmatrix}\]

If the \(i\)-th powshell script had the \(j\)-th token, then we set \(t_{ij}\) to 1; if the \(i\)-th powershell script did not have the \(j\)-th token, then we set \(t_{ij}\) to 0.

In step 2, we computed the Jaccard similarities [11] between the two powershell scripts. The Jaccard similarity between \(F_i\) and \(F_j\) was computed as the following.

\[\text{Jaccard Similarity } Sim_{ij} = \frac{\text{Len}\left(S_i \cap S_j\right)}{\text{Len}\left(S_i \cup S_j\right)}\]

\(S_i\) was a set of powershell tokens of a file \(F_i\) and \(S_j\) was a set of powershell tokens of file \(F_j\). Note that the Jaccard similarity index was required here to determine whether \(F_i\) and \(F_j\) are similar. In contrast, we could have used a longest common subsequence (LCS) method instead of a Jaccard similarity. However, we found that doing so required substantial computational processing time. Hence, we used Jaccard similarity.

In step 3, we generated an \(n \times n\) adjacency matrix \(A\) by setting \(a_{ij}\) to 1 when the Jaccard similarity \(Sim_{ij}\) was greater than the top-k similarity, described in the following.

\[A = \begin{bmatrix}
  a_{11} & \cdots & a_{1n} \\
  \vdots & \ddots & \vdots \\
  a_{n1} & \cdots & a_{nn}
\end{bmatrix}\]

This meant that when \(a_{ij}\) was equal to 1, the powershell script \(F_i\) was similar to the powershell script \(F_j\). When \(a_{ij}\) was equal to 0, the powershell script \(F_i\) was not similar to the powershell script \(F_j\).

In step 4, we trained the GCN using feature matrix \(X\) and adjacency matrix \(A\). Figure 4 illustrates the GCN model. It had two dropout layers and two GCN layers. In the two dropout layers, we set the dropout rate to 0.5. The first GCN layer used 16 kernels and used Rectified Linear Unit (RELU) for activation. The second layer used two kernels and SoftMax for the activation.
scripts into five subsets and in the other subsets for training; we completed five experiments in total. In each experiment, we used Keras 2.3.1 [22].

For the deep learning framework, we used 800 powershell scripts for training and 200 powershell scripts for testing.

We used 5-fold cross validation [21]. Thus, we split 1000 powershell scripts for malicious powershell detection provided by the Electronics and Telecommunication Research Institute (ETRI) [19].

Figure 3 shows that for malicious powershell detection using GCN, we first implemented a feature extraction module using PSpars and Python. Figure 5 shows that the feature data were transformed into frequency data. Second, we implemented the Jaccard similarity computing module. Third, we implemented an adjacency matrix generation module based on Jaccard similarity. Fourth, we modified keras-gcn [20] for malicious powershell detection using an adjacency matrix.

The experimental environment was as follows: we used Windows 10 pro, Intel i7 3.7 GHz CPU, 16 GB RAM, and GeForce 1080 GPU. For the deep learning framework, we used Keras 2.3.1 [22].

We used the following performance metrics. The recall (detection rate), false positive rate (FPR), and accuracy were defined as the following.

\[
\text{Recall} = \frac{TP}{TP + FN}
\]

Figure 4. GCN model.

The GCN model was defined as follows.

\[
H = softmax \left( \tilde{A} \text{RELU} \left( \tilde{A} \times W^0 \right) W^1 \right)
\]

The neural network weights \(W^0\) and \(W^1\) were trained. Note that two GCN layers are included in the developed GCN model. The first weight matrix \(W^0\) was 20,000 \(\times\) 16, and the second weight matrix \(W^1\) was 16 \(\times\) 2.

Finally, by using the adjacency matrix from Jaccard similarity and GCN, we determine whether a new powershell script was malicious.

4. Experimental Results

4.1. Setup

We used 1000 powershell scripts, including 500 normal and 500 malicious powershell scripts for malicious powershell detection provided by the Electronics and Telecommunication Research Institute (ETRI) [19].

Figure 5 shows that for malicious powershell detection using GCN, we first implemented a feature extraction module using PSpars and Python. Figure 5 shows that the feature data were transformed into frequency data. Second, we implemented the Jaccard similarity computing module. Third, we implemented an adjacency matrix generation module based on Jaccard similarity. Fourth, we modified keras-gcn [20] for malicious powershell detection using an adjacency matrix.

Figure 5. Powershell frequency data.

In addition, we used 5-fold cross validation [21]. Thus, we split 1000 powershell scripts into five subsets and in the \(i\)-th experiment, we used \(i\)-th subset for the test and the other subsets for training; we completed five experiments in total. In each experiment, we used 800 powershell scripts for training and 200 powershell scripts for testing.

The experimental environment was as follows: we used Windows 10 pro, Intel i7 3.7 GHz CPU, 16 GB RAM, and GeForce 1080 GPU. For the deep learning framework, we used Keras 2.3.1 [22].

We used the following performance metrics. The recall (detection rate), false positive rate (FPR), and accuracy were defined as the following.

\[
\text{Recall} = \frac{TP}{TP + FN}
\]

\[
\text{FPR} = \frac{FP}{FP + TN}
\]

\[
\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}
\]
\[
FPR = \frac{FP}{FP + TN}
\]

Accuracy = \frac{(TP + FP)}{(TP + FP + FN + TN)}.

Here, True Positive (TP) was the number of malicious scripts predicted as malicious; False Negative (FN) was the number of malicious scripts predicted as normal; True Negative (TN) was the number of normal scripts predicted as normal; False Positive (FP) was the number of normal scripts predicted as malicious.

The second performance metric was the adjacency matrix generation time. To use GCN for malicious powershell detection, we needed an adjacency matrix to find powershell scripts such as a new powershell script. In addition, we had to determine as soon as possible whether a new powershell script was malicious. Therefore, we generated an adjacency matrix within a reasonable time period.

Note that we attempted to continue training for 200 epochs and if the loss did not improve for 10 successive epochs, then training was ceased.

In the experiment, the first goal was to increase recall and accuracy and decrease FPR using GCN. The second goal was to generate an adjacency matrix within a reasonable time.

4.2. Results

In the experiment, we set the number of adjacent nodes from zero to three. When the number of adjacent nodes was zero, we used an identity matrix [23] for GCN (see Figure 6). In Section 4.2.1, we present the experimental results based on the number of adjacent nodes. In Section 4.2.2, we provide experimental results based on the number of powershell scripts. In Section 4.2.3, we provide the adjacency matrix generation time. In Section 4.2.4, we present the GCN training time.

Figure 6. Identity matrix.

### 4.2.1. Number of Adjacent Nodes

Figure 7 shows that when the number of adjacent nodes was 0, the detection rate was 88.4% and when the number of adjacent nodes was 1, the detection rate was 89.4%, which was 1% higher than when using the identity matrix. When the number of adjacent nodes was 2, the detection rate was 96.6%, which was 8.2% higher than when using the identity matrix. However, when the number of adjacent nodes was 3, the detection rate was 95%, which was less than when the number of adjacent nodes was 2.

When the number of adjacent nodes increased, the detection rate also increased because additional feature data could be obtained from adjacent nodes. However, when there were too many adjacent nodes, the detection rate decreased compared to when the number of adjacent nodes was two, even though it was higher than when using the identity matrix.

Figure 8 shows the FPR. When the number of adjacent nodes was 0, the FPR was 1%. When the number of adjacent nodes was 1, the FPR was 0.8%, which decreased by 0.2% compared to using the identity matrix. However, when the number of adjacent nodes was 2 or 3, the FPR was 2% and it increased by 1%. This showed that if we used adjacent nodes, then the FPR decreased. However, when there were too many adjacent nodes, the FPR increased.
Figure 7. Recall (Detection rate) according to the number of adjacent nodes.

Figure 8 shows the FPR. When the number of adjacent nodes was 0, the FPR was 1%. When the number of adjacent nodes was 1, the FPR was 0.8%, which decreased by 0.2% compared to using the identity matrix. However, when the number of adjacent nodes was 2 or 3, the FPR was 2% and it increased by 1%. This showed that if we used adjacent nodes, then the FPR decreased. However, when there were too many adjacent nodes, the FPR increased.

Figure 8. False Positive Rate according to the number of adjacent nodes.

Figure 9 provides the accuracy. When the number of adjacent nodes was 0, accuracy was 93.7%. When the number of adjacent nodes was 1, it was 94.3%. When the number of adjacent nodes was 2, it was 97.3% and it increased by 3.6% compared to using the identity matrix. When the number of adjacent nodes was 3, it was 96.5% and it was higher than using the identity matrix but was less than when the number of adjacent nodes was 2. This showed that we could increase the accuracy by using adjacent nodes; however, the accuracy decreased when the number of adjacent nodes was high.

Figure 9. Accuracy according to the number of adjacent nodes.
Table 1 shows the results in each experiment.

**Table 1. Results according to the number of adjacent nodes.**

|     | TP  | FN  | FP  | TN  | Recall | FPR  | Acc  |
|-----|-----|-----|-----|-----|--------|------|------|
| Top-0 | 88.4 | 11.6 | 1   | 99  | 88.4   | 1    | 93.7 |
| Top-1 | 89.4 | 10.6 | 0.8 | 99.2| 89.4   | 0.8  | 94.3 |
| Top-2 | 96.6 | 3.4  | 2   | 98  | 96.6   | 2    | 97.3 |
| Top-3 | 95   | 5    | 2   | 98  | 95     | 2    | 96.5 |

### 4.2.2. Number of Powershell Scripts

Figure 10 shows the accuracy according to the number of powershell scripts. We provided accuracy when the number of adjacent nodes was zero and two. When it was 0, the average accuracy was 93.25%. When it was 2, the accuracy was, on average, 97.6%. When it was 2, the accuracy increased by 4.35%. This meant that we increased the accuracy using adjacent nodes rather than using an identity matrix.

Note that because we used 5-fold cross validation, the accuracy did not increase when the data size increased from 250 to 1000. However, when we used adjacent nodes (e.g., top-2), the accuracy increased compared to using an identity matrix (e.g., top-0).

### 4.2.3. Adjacency Matrix Generation

Figure 11 shows the adjacent matrix generation time measured according to the number of powershell scripts. When the number was 250, the generation time was 8 s. When the number was 500, the generation time was 31 s. When the number was 750, the generation time was 70 s. When the number was 1000, the generation time was 139 s. The adjacency matrix generation time was proportional to the number of powershell scripts. When the number was 1000, the time per powershell script was 139 ms. Thus, we concluded that these results were reasonable for malicious powershell detection.

### 4.2.4. GCN Training Time

Figure 12 shows the GCN training time. When the number of powershell scripts was 1000, the GCN training time was 45 s. This meant that it took approximately 45 ms per powershell script. This was reasonable for malicious powershell detection.
mer being slightly higher than the latter. Moreover, the accuracy of GCN was 2.84% higher.

The frequency data was similar to that of CNN using powershell sequence data, with the for-

ershell dataset. For the 1500 powershell scripts, the accuracy of CNN-seq, CNN-freq, and

The accuracy of CNN-freq was 2.84% lower than that of CNN-freq. Mimura et al. demo-

nstrated the malicious powershell detection

Therefore, in future research, we will modify Keras-GCN to process 2000 or more pow-

ershell dataset. For the 1500 powershell scripts, the accuracy of CNN-seq, CNN-freq, and

ershell scripts.

In this study, we modified GCN to use powershell frequency data. However, for fu-

ther modifications, we are considering modifying GCN to use powershell sequence data to

We randomly selected 500, 1000, and 1500 powershell scripts from the...
We randomly selected 500, 1000, and 1500 powershell scripts from the ETRI powershell dataset. For the 1500 powershell scripts, the accuracy of CNN-seq, CNN-freq, and GCN were 96%, 93.8%, and 96.6%, respectively. The accuracy of GCN using powershell frequency data was similar to that of CNN using powershell sequence data, with the former being slightly higher than the latter. Moreover, the accuracy of GCN was 2.84% higher than that of CNN-freq. Mimura et al. demonstrated the malicious powershell detection using word embedding [25]. However, we cannot compare the accuracy to our study owing to the different datasets utilized in both studies.

In this study, we modified GCN to use powershell frequency data. However, for future research, we are considering modifying GCN to use powershell sequence data to achieve higher detection accuracy. Additionally, while we tried using 2000 powershell scripts, the Keras-GCN [20] modified for detecting malicious powershell caused an error. Therefore, in future research, we will modify Keras-GCN to process 2000 or more powershell scripts.

5. Discussion

Here, we proposed a malicious powershell detection method using GCN and provided an adjacency matrix generation method using Jaccard similarity. In the experiment, we showed that the malicious powershell detection rate increased by 8.2% compared to the detection rate when using an identity matrix.

We used the powershell frequency data for the feature data to use GCN. However, we could use powershell sequence data. When we used the powershell sequence data, we could consider the sequence of the powershell tokens. In future work, we will study a method for using powershell-sequence data in GCN.

In addition, we used Jaccard similarity to generate an adjacency matrix for the GCN. When we used the powershell frequency data for the feature data, the Jaccard similarity was appropriate. However, we expected that when we used powershell-sequence data, we could use the longest common subsequence [26] to generate an adjacency matrix.

Funding: This work was supported by a National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2019R1G1A11100261), by research funds for the newly appointed professors of Jeonbuk National University in 2021, and by the HPC support project by MSIT and NIPA.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The author declares no conflict of interest.

References

1. AV-TEST. Available online: https://www.av-test.org/fileadmin/pdf/security-report/AV-TEST_Security_Report_2018-2019.pdf (accessed on 1 July 2021).
2. Hendler, D.; Kels, S.; Rubin, A. Detecting malicious powershell commands using deep neural networks. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security, Incheon, Korea, 29 May 2018.
3. Song, J.; Kim., J.; Choi, S.; Kim, J.; Kim, I. Implementation of a static powershell analysis based on the cnn-lstm model with token optimizations. In Proceedings of the WISA Workshop, Jeju, Korea, 21–24 August 2019.
4. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst. 2012, 25, 1097–1105. [CrossRef]
5. Sutskever, I.; Vinyals, O.; Le, Q.V. Sequence to sequence learning with neural networks. In Proceedings of the 27th International Conference on Neural Information Processing Systems, Montreal, QC, Canada, 8–13 December 2014.
6. Gibert, D. Convolutional Neural Networks for Malware Classification. Master’s Thesis, Universitat de Barcelona, Barcelona, Spain, 2016.
7. Pascanu, R.; Stokes, J.W.; Sanossian, H.; Marinescu, M.; Thomas, A. Malware classification with recurrent networks. In Proceedings of the 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brisbane, Australia, 19–24 April 2015; pp. 1916–1920.
8. Kipf, T.N.; Welling, M. Semi-supervised classification with graph convolutional networks. In Proceedings of the 5th International Conference on Learning Representations, Toulon, France, 24–26 April 2017.

9. Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; Yu, P.S.; Wu, Z.; Pan, S.; Chen, F.; Long, G.; et al. A Comprehensive Survey on Graph Neural Networks. *IEEE Trans. Neural Netw. Learn. Syst.* 2021, 32, 4–24. [CrossRef] [PubMed]

10. Fan, W.; Ma, Y.; Li, Q.; He, Y.; Zhao, E.; Tang, J.; Yin, D. Graph neural networks for social recommendation. In Proceedings of the World Wide Web Conference (WWW), San Francisco, CA, USA, 13 May 2019.

11. Jaccard Index. Available online: https://deepai.org/machine-learning-glossary-and-terms/jaccard-index (accessed on 1 July 2021).

12. Han, J.; Yan, G.; Jin, D. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In Proceedings of the 2019 49th IEEE/IFIP International Conference on Dependable Systems and Networks, Portland, OR, USA, 24 June 2019.

13. Pei, X.; Yu, L.; Tian, S. AMLNet: A deep learning framework based on graph convolutional networks for malware detection. *Comput. Secur.* 2020, 93, 101792. [CrossRef]

14. Hong, Y.; Hu, Y.; Lai, C.-M.; Wu, S.F.; Neamtiu, I.; McDaniel, P.; Yu, P.; Cam, H.; Ahn, G.-J. Defining and Detecting Environment Discrimination in Android Apps. In Proceedings of the International Conference on Security and Privacy in Communication Networks: SecureComm 2017, Niagara Falls, ON, Canada, 22–25 October 2017.

15. DeFreez, D.; Thakur, A.V.; Rubio-González, C. Path-Based Function Embedding and Its Application to Error-Handling Specification Mining. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, 26 October 2018.

16. Zachary’s Karate Club. Available online: https://konect.cc/networks/ucidata-zachary/ (accessed on 1 July 2021).

17. Degree Matrix. Available online: https://faculty.math.illinois.edu/Macaulay2/doc/Macaulay2-1.16/share/doc/Macaulay2/Grabs/html/_degree__Matrix.html (accessed on 1 July 2021).

18. PSParser. Available online: https://npmjs.com/package/psparser (accessed on 13 May 2021).

19. Electronics and Telecommunication Research Institute (ETRI). Available online: https://etri.re.kr (accessed on 14 May 2021).

20. Keras-gcn. Available online: https://github.com/tkipf/keras-gcn (accessed on 14 May 2021).

21. A Gentle Introduction to k-Fold Cross Validation. Available online: https://machinelearningmastery.com/k-fold-cross-validation/ (accessed on 1 July 2021).

22. Keras. Available online: https://keras.io (accessed on 14 May 2021).

23. Identity Matrix. Available online: https://sciedirect.com/topics/mathematics/identitymatrix (accessed on 1 July 2021).

24. Choi, S. Malicious PowerShell Detection Using Attention against Adversarial Attacks. *Electron* 2020, 9, 1817. [CrossRef]

25. Mimura, M.; Tajiri, Y. Static detection of malicious PowerShell based on word embeddings. *Internet Things* 2021, 15, 100404. [CrossRef]

26. Longest Common Subsequence Problem. Available online: https://ics.uci.edu/~jeppestein/151/960229.html (accessed on 1 July 2021).