Received January 27, 2022, accepted February 4, 2022, date of publication February 14, 2022, date of current version February 25, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3151357

FAM: Featuring Android Malware for Deep Learning-Based Familial Analysis

YOUNGHOON BAN[1], SUNJUN LEE[2], DOKYUNG SONG[3], HAEHYUN CHO[2], AND JEONG HYUN YI[2], (Member, IEEE)
[1]School of Software Convergence, Soongsil University, Seoul 06978, South Korea
[2]School of Software, Soongsil University, Seoul 06978, South Korea
[3]Department of Computer Science, Yonsei University, Seoul 03722, South Korea

Corresponding author: Haehyun Cho (haehyun@ssu.ac.kr)

This work was supported in part by the Mid-Career Researcher and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT), and Future Planning under Grant NRF2020R1A2C2014336 and Grant NRF2021R1F1A1057436.

ABSTRACT To handle relentlessly emerging Android malware, deep learning has been widely adopted in the research community. Prior work proposed deep learning-based approaches that use different features of malware, and reported a high accuracy in malware detection, i.e., classifying malware from benign applications. However, familial analysis of real-world Android malware has not been extensively studied yet. Familial analysis refers to the process of classifying a given malware into a family (or a set of families), which can greatly accelerate malware analysis as the analysis gives their fine-grained behavioral characteristics. In this work, we shed light on deep learning-based familial analysis by studying different features of Android malware and how effectively they can represent their (malicious) behaviors. We focus on string features of Android malware, namely the Abstract Syntax Trees (AST) of all functions extracted from each malware, which faithfully represent all string features of Android malware. We thoroughly study how different string features, such as how security-sensitive APIs are used in malware, affect the performance of our deep learning-based familial analysis model. A convolutional neural network was trained and tested in various configurations on 28,179 real-world malware dataset appeared in the wild from 2018 to 2020, where each malware has one or more labels assigned based on their behaviors. Our evaluation reveals how different features contribute to the performance of familial analysis. Notably, with all features combined, we were able to produce up to an accuracy of 98% and a micro F1-score of 0.82, a result on par with the state-of-the-art.

INDEX TERMS Malware classification, deep learning, Android security, abstract syntax tree.

I. INTRODUCTION

Life on the mobile device is still full of danger—one wrong click can send your device a malicious application that would execute malware on your device and turn your machine into an unwitting participant in a botnet. Albeit the security for the mobile ecosystem has matured and the prevalence of drive-by-downloads seems decreased, the number of reported Android malware is increasing [15].

To tame an ever-increasing number of malware, the security community have proposed various machine learning or deep learning-based approaches to analyze malware, and reported promising results. In particular, a large body of research work exist aimed at detecting Android malware by using machine learning and deep learning algorithms [6], [8], [14], [15], [18]–[20], [26]–[28], [30], [33], [34], [37], [40], [41], [43]–[45], [48]. The proposed techniques differ in features and algorithms used, but most of prior work formulate malware analysis as *malware detection*, which is a binary classification problem—classifying a given application into malware or a benign one. Even though prior work reported a high accuracy in detecting Android malware, their analysis result is limited in that such approaches are unable to cluster or classify malware into various families that describe specific malicious behavior.

To address this problem, familial analysis has been proposed [12], [13], [46], the process which classifies a given malware into families based on their behavior and, therefore,

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Sharif.

their potential impact on victims. Similar to malware detection approaches, prior familial analysis approaches employ various deep neural network models and use various features of malware, but aim to produce a more fine-grained and descriptive analysis result than malware detection approaches. A line of work use *string features* of malware, but, using those features in isolation without considering the context can yield low accuracy [13], [46]. Another line of work use *graph features* of malware such as function call graphs and control flow graphs. Recently, Fan *et al.* proposed an approach using a graph embedding technique to efficiently find similar graphs [12].

A common limitation of prior work is that their analysis results are still not fine-grained and descriptive enough. Most of prior work formulate the problem as classifying a given malware into a single family. Fan *et al.* [12], for example, propose unsupervised clustering, which means that their approach can only give a test sample a single label. This is less than ideal, because real-world malware cannot easily be described in a single, concise label. Another limitation of prior familial analysis approaches is that they employ old datasets whose representativeness nowadays is questionable. Moreover, the split between the training and test data of prior work was often arbitrary, which, arguably, does not reflect real-world analysis scenarios. A malware classification model is most useful when it predicts the behavior of *future* malware based on historical malware dataset.

This paper formulates the problem of familial analysis as a multi-label classification problem, in which the model is trained to predict multiple behavioral characteristics of a given malware. By using multiple yet concise labels, such a model can provide a fine-grained description of a given malware. Our work also distinguishes from prior work in that we use up-to-date real-world Android malware, a more representative dataset than prior work. We also study, in addition to a standard 80%-20% training and test data split, a split between years in which malware appeared. This split allows us to test whether the model can classify future malware based on historical data, which is arguably a more useful scenario. Using this formulation and the dataset for familial analysis, we study how different features affect the performance of neural networks designed to perform such familial analysis of malware. Through our study, we propose a set of features that can most effectively describe malicious behavior of malware. We use a standard deep neural network, convolutional neural network (CNN), with different sets of features extracted from malware, to train them to eventually generate a multi-label classification model. We thoroughly evaluate the model with the in-the-wild dataset that consists of real-world Android malware appeared from 2018 to 2020. Our evaluation results clearly demonstrate that the best-performing features can effectively classify recent, real-world malware (an accuracy up to 98% and a micro F1-score 0.82). Our major contributions are, thus, as follows.

- We first conduct a thorough study of familial analysis for Android malware by modeling it as a multi-label classification problem.
- We thoroughly study how different string features of malware could affect the performance of a neural network model trained for multi-label familial analysis.
- We conduct a large-scale evaluation using real-world malware in realistic deployment scenarios. We also conduct a comparative study, and demonstrate that our model performs familial analysis with the high accuracy (98%) and micro F1-score (0.822).

II. RELATED WORK

There exists a large body of research aimed at familial analysis of Android malware. Overall, previous research efforts focussing on the multi-class classification could not accurately represent behaviors of malicious applications and the multi-label classification has not been extensively studied yet. While, in this work, we conducted an in-depth study on the multi-label classification by finding and characterizing functions which perform malicious behaviors.

DroidSIFT [46] constructs the weighted contextual API dependency graph for each malware and conducts the familial analysis by calculating the graph edit distance between malicious applications. However, the computational complexity of graph edit distance is exponential in the number of nodes and graphs, and thus, DroidSIFT is feasible for the limited size of dataset. Feng *et al.* [13] proposed Astroid to discover the shared functionality between multiple malicious applications by using inter-component call relations and data-flow properties. They used the shared functionality as a signature to perform the familial analysis. Because the quality of the signatures relies on the static analysis techniques the employed, the effectiveness of Astroid had to be limited by the precision of the techniques. GefDroid [12] aimed to overcome the limitation of previous graph-based approaches by using a graph embedding technique for efficiently conduct the familial analysis. However, since GefDroid focuses on the unsupervised clustering, it can only output singly labelled results. Similarly, GSFDroid [24] also uses graph-based features using the function call graph to analyze app behavior with a graph embedding technique. Li *et al.* [25] also cluster by checking whether malware samples are shared among malware. For this purpose, fingerprint based library removal technology was used. Use the feature representing the application using the bit vector format. Fan *et al.* [10] proposed FalDroid, a system that automatically classifies malicious codes by implementing fregraphs, which are graphs related to API calls, from function call graphs to express the behavior of the same family sample. The similarity between the two graphs is calculated using the cosine similarity. It performs malignant family classification, but because it performs multi-class classification, it cannot classify unlearned classes. Also, since it implements fregraphs that use sensitive API calls, malicious actions that do not use API calls cannot be detected. CTDroid [11] proposed a method for automatically selecting useful features for malware analysis using the corpus of Android malware-related technical blogs. To this end, two semantic matching rules were proposed to bridge the gap between natural language and programming language. Malicious code detection and family classification are performed using this rule. We used technical blogs collected from 2011 to 2017, and may not be effective for malware samples after 2017. DREBIN [8] proposed a lightweight detection method that can directly identify malicious applications on mobile devices. Malicious application detection is performed by combining many features such as privileges and API calls extracted through static analysis. Also, the detection results are explained using features. However, it is difficult to expect an accurate interpretation of the actual operation performed by the application. GroupDroid [29] only focuses on repackaged malicious applications that inject malicious code into benign applications. To identify the syntactical similarity of each application, 3D-CFG centroids and API vectors are utilized to identify the similarity of the code. Aktas and Sen [7] provide UpDroid, a dataset that uses an update attacks that adds malicious payloads to the application runtime. To overcome the limitations of existing malware detection and family classification studies using static analysis, features using static and dynamic analysis were used, and a family classification algorithm strong against obfuscation was proposed. To overcome the low accuracy problem of existing deep learning by converting malicious codes to gray images, Yuan *et al.* [44] proposed MDMC, a malicious code classification model. MDMC is converting malware binaries into Markov images according to the bytes transfer probability matrix. Existing machine learning-based malware detection approaches use features extracted from malicious and normal samples, which has been proven to be ineffective against complex malware in real world. On the other hand, deep learning approaches have shown promising results due to their automatic feature extraction from raw data. Sihag *et al.* [40] proposed De-LADY, a malware detection framework based on deep learning and dynamic analysis for Android. De-LADY utilizes the behavioral characteristics of the application by using the log extracted by running the application. However, since De-LADY uses an emulator to run the application, it cannot detect malicious applications equipped with the ant-emulation technique.

III. BACKGROUND

A. MALWARE DETECTION

Motivated by an ever-increasing number of Android malware, a surge of research work proposed different machine learning approaches for detecting malware. Typically, machine learning-based malware detection systems use static features such as APIs, permissions, and function call graphs [6], [8], [14], [30], [41]. While, Gong *et al.* used APIs invoked while an application is running as a feature [15].

On the other hand, there exists a large body of deep learning-based malware detection systems [18], [20], [34], [43], [45], [48]. Because deep learning-based systems generally do not require a manual feature selection process, the proposed approaches use not only APIs, permissions, and function call graphs, but also, opcode and bytecode as inputs to deep learning models.

Those machine learning- and deep learning-based malware detection systems showed a high accuracy in the binary classification problem—classifying applications into malware or benign applications. However, they lack a capability to cluster malware into various families based on their features. Despite advances in automated analysis techniques, to completely analyze new malware for discovering its malicious behaviors and impacts on victim users, security analysts should perform detailed analysis. Classifying Android malware into known families can accelerate this process, considerably helping analysts.

B. MALWARE FAMILIAL ANALYSIS

Adversaries tend to create new malware by injecting previously used malicious code (with or without modifications) into a benign application, disguising itself as a benign application. A study showed that Android malware has distinguishable behaviors and they can be categorized into families based on their behaviors [47]. Therefore, if we can automatically classify the family of unknown malware based on their behavioral characteristics, it can greatly help security analysts to efficiently and effectively analyze malware.

Recently, deep learning-based approaches have been proposed for a familial analysis of Android malware. To train a model that can be used for a familial analysis, they extract various features from malware. A line of work use string features, e.g., APIs and permissions, of malware, but simply using these features can yield low accuracy [46]. Another line of work propose using graph-based features such as function call graphs and control flow graphs [10], [12], [24]. Based on the observation that graph matching algorithms are costly, GefDroid [12], for example, proposed using a graph embedding technique, struct2vec, for a more efficient matching. GefDroid, however, uses unsupervised clustering which only gives a test sample a single label only [12].

Moreover, a common limitation of the previous approaches is that they used existing open-source datasets whose representativeness is questionable. Although such open-source dataset is well-, singly-labeled and relatively balanced for training a model, but limited in size and old. It is, therefore, uncertain whether previous approaches also work effectively on the recent real-world malware. In addition, as recent malware possess multiple malicious behaviors as shown in Table 3, it is difficult to assign malware a single label in many cases. Hence, in order to obtain more accurate, descriptive, and representative familial analysis results, multi-label classification analysis needs to be performed and on recent, in-the-wild malware dataset.

TABLE 1. Advantages and disadvantages of malware familial analysis approaches.

Name	Advantages	Disadvantages
DroidSIFT [46]	They performed malware detection using the weighted contextual API dependency graph to counter the transformation attack.	Since they feature graphs, they are suitable for limited data due to the computational complexity of the graph editing distance for similarity they proposed. In addition, since the data used in the experiment uses outdated data, it is difficult to respond to the latest malware.
GefDroid [12]	They performed family analysis by constructing a subgraph from FCG for malware family analysis and measuring the similarity.	Because they focus on unsupervised clustering to performed family analysis, they only output singly labeled results.
Droidtec [27]	Thye performed automatic malicious code localization using depth-first invocation traversal using bytecode instructions to understand the behavior of the application.	Because Droidtec uses API call sequences as features it is difficult to analyze the exact behavior. Also, the dataset used for the experiment is old data.
De-LADY [40]	De-LADY Using the log extracted by running the application, the behavioral characteristics of the application are utilized.	Malicious applications that detect the emulated environment cannot be detected and perform binary classification.
FAM	We perform multi-label family analysis by extracting new string features representing malicious behavior from the real-world dataset.	Since we label the data ourselves, data bias occurs and there is an Out-of-vocabulary problem.

TABLE 2. The number of malware dataset after labeling with their malicious behaviors.

Year	2018	2019	2020	**Total**
# of Malware	16,345	5,126	6,707	**28,179**

TABLE 3. The histogram of malware behaviors in our dataset.

# of Behaviors	1	2	3	4	**Total**
# of Malware	21,277	6,765	136	1	**28,179**

IV. OVERVIEW

We aim to understand how different features of Android malware affect the performance of machine learning algorithms, and identify features that can most effectively represent different malicious behaviors of malware. To this end, we perform malware familial analysis with a standard neural network, Convolutional Neural Network (CNN) [21], with varying sets of features, and evaluate the effectiveness of each set of features. Among various features of Android applications, we focus on evaluating *string features*, which can be obtained from tokenizing abstract syntax trees (AST) of functions found in malware, a faithful string representation of an Android application. An example of such string feature are security-sensitive APIs used in malware, which many prior work considered as an important feature, as they indicate how an application interacts with the Android framework and the rest of the system.

In the following sections, we show how we collected dataset for our study (subsection V-A), how we generate the implentational features of malware (subsection V-B and subsection V-C), and how we use the features with a convolutional neural network for familial analysis (section VI).

V. DATA GENERATION

A. DATASET

To perform an extensive study with real-world data, we collected 189,847 malware that appeared from 2018 to 2020 in the wild by using VirusShare [3]. The malware provided by VirusTotal does not have any label. Therefore, we obtained analysis reports for each malware from various antivirus product through VirusTotal [4]. Next, we used an automatic malware tagging tool, AVclass2, that categorize malware samples according to their malware class, behaviors, etc [39]. Among the tags, we label each malware with their malicious behaviors. When we labeled malware, if the number of malicious behaviors aggregated by AVclass2 exceeds one, we use the behaviors as labels of the malware so that malware can be assigned multiple labels. If any vendor cannot find a malicious behavior of malware, we label the malware as "Undefined." Table 2 shows the result of labeling 189,847 malicious applications that we collected. We labelled each malware according to its malicious behavior. In total, we obtained 28,179 labelled malware that perform one or more malicious behaviors identified. Table 3 shows the histogram of malware based on the number malicious behaviors of them. As a result, we found that the rate of malware performing two or more complex malicious behaviors is about 24.5%, which indicates that it is difficult to assign a single behavior label to malware. Also, the number of single labels is 21 and the total number of labels used to describe all malware is 84.

It is worth noting that our dataset is not balanced: the number of malicious applications in some behavioral categories is less than 10 because we use in-the-wild malware. Hence, the unbalanced dataset may not be an optimal to show the maximum performance of our system. However, we believe that our evaluation results can demonstrate the realistic effectiveness when we launch a malware classification system in real-world. In addition, we did not exclude obfuscated malware for demonstrating for the same reason.

B. PRE-PROCESSING

In this pre-processing step, we remove known, benign functions provided by common third-party libraries such as Gson [5] and functions in Android libraries in each malware. Consequently, remain functions of the malware can be candidates that perform malicious behaviors. Next, we generate string features of a given malicious application of by extracting Abstract Syntax Trees (AST) of all functions found in the application. Finally, the extracted ASTs are further distilled into a more compact set of ASTs based on whether they use sensitive APIs or not.

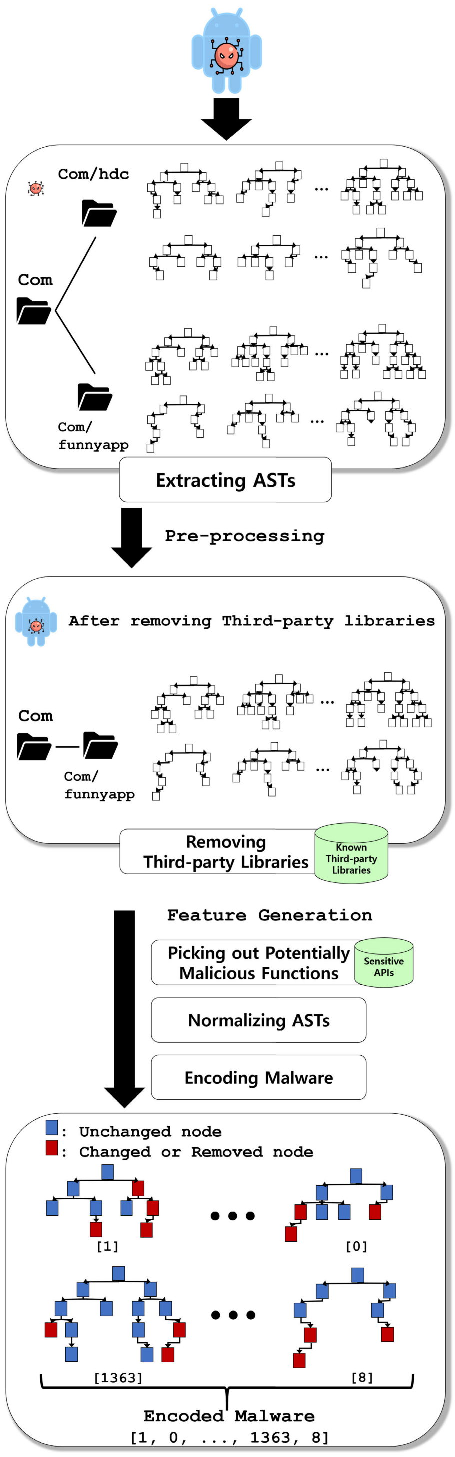


FIGURE 1. Overview of our string feature generation process.

1) THIRD-PARTY LIBRARY FUNCTION COLLECTION

Third-party libraries are widely used in Android applications including malware to avoid reimplementing common functionalities. In this work, we are interested in finding and characterizing functions which perform malicious behaviors. Therefore, known and benign third-party libraries can introduce significant noise to analysis results, and thus, we should detect and remove them. To this end, we identify 35,555 benign libraries used in Android applications, which are obtained (i) by crawling with Up2Dep [35] and (ii) from well-curated lists of third-party libraries [22], [23]. The collected third-party libraries are used to filter out benign functions in malware, which substantially contributes to the improvement of the efficiency and effectiveness of our familial analysis.

TABLE 4. The number of known third-party libraries collected.

Up2Dep [35]	Libd [23]	Li et al. [22]	**De-duplicated Total**
18,507	11,458	5,926	**35,555**

2) ABSTRACT SYNTAX TREE EXTRACTION

The Abstract Syntax Tree (AST) of a given function is a faithful string representation of the function, which is normally created as a result of parsing during the compilation process. An AST of a given function includes not only all syntactic elements but also the names of the identifiers (e.g., function and variable names) and compile-time constants (e.g., numeric constants and string literals). Android malware are available in Java bytecode, which can be decompiled into ASTs of functions found in a given malware [17]. The decompiled ASTs lack the names of local variables. We decompile all malware in our dataset to obtain ASTs of all functions in each malware by using AndroGuard [2]. In this process, we filter out the ASTs of *known* functions using the list of third-party libraries collected in the previous step; we only compile *unknown* functions into ASTs and thus potentially malicious functions in each malware.

3) SENSITIVE API-BASED DISTILLATION

Even after filtering out *benign* functions from each malware, there are still a large number of ASTs extracted in each malware. In our dataset, for example, we observe that there are on average 600 ASTs, after excluding the third-party libraries from a malicious application. Therefore, after we obtain ASTs from each malware, we further pick out potentially malicious functions by using the list of the sensitive APIs studied by previous work [1], [9]. SuSi [1] and FlowDroid [9] provide a set of *source* APIs (i.e., APIs used to read sensitive information) and *sink* APIs (i.e., APIs used to send and store sensitive information to somewhere) commonly used by malware, from which we obtain 25,308 sensitve APIs. We distill ASTs extracted from the prior step into a more compact set of ASTs by eliminating all ASTs that do not contain any sensitive API calls.

C. FEATURE GENERATION

To study how different features contribute to the classification performance of a neural network, in this section, we identify string features that can be obtained from a given AST. For each identified feature, we describe how each feature can be removed through normalization. This effectively allows us to perform an ablation study of each feature. We then present an encoding procedure that can convert each AST into a numeric feature vector, which can be fed into a neural network model.

1) AST NORMALIZATION

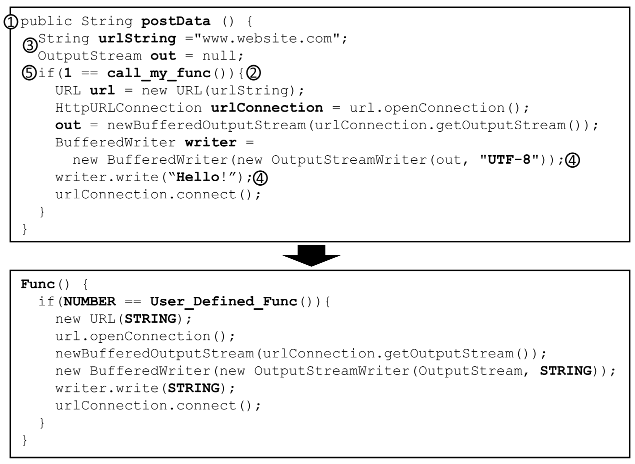
From an AST that may contribute to malicious behaviors of malware, we identify five string features, which are described in the following normalization steps, which effectively eliminates a potentially useful feature after each step: (i) we remove the name of the function that corresponds to the given AST; (ii) we remove all local variables whose names are assigned arbitrarily by the decompiler; (iii) we replace the names of all user-defined functions called within the given AST to a predefined token "User_Defined_Func"; (iv) we replace all string literals with a token "STRING"; and (v) we replace all numeric constants with a token "NUMBER". An example of the normalization process is shown in Figure 2. By conducting this normalization process, we effectively remove features of an AST one after another; by comparing the performance of a classification model trained on datasets before and after each normalization step, we can better understand each feature's role in classifying malware into its families.

2) MALWARE ENCODING

After the AST normalization process, we encode each malware with its normalized ASTs, such that the malware is represented as a sequence of tokens. To this end, we first traverse each decompiled and normalized AST in a depth-first manner, such that we can reconstruct a representation of each function in a lexical order, i.e., the order in which the original source code would have appeared. Next, we index all ASTs which are going to be used in the training step with deep learning algorithms. In our dataset, we found 954,039 unique ASTs in total. With the indexed ASTs, we generate a string for each malware using the following procedure: (i) we sort ASTs of a malicious application to arrange ASTs which have the source APIs before ASTs which have the sink type APIs—this sorting process helps us have a consistent representation of all malware; (ii) we find the index assigned to each AST found in the malicious application; and (iii) we generate a vector of indices for each malicious application. Finally, we add paddings to make the generated vectors the same length for all malware. This encoded malware is directly used for training a neural network. If we encounter, in the validation or testing phase, an AST *unseen* in the training phase, we assign a token "UNK" to it.

VI. MODEL

FAM takes the architecture of a standard convolutional neural network (CNN) [21]. The input features of a given malware are represented as a sequence of encoded ASTs. These input features first go through an embedding layer which maps each AST into a vector in the embedding space. The embedding vectors then go through one-dimensional convolutional layers (Conv1D) with ReLu activations, and dense layers stacked on top. We chose Conv1D to capture the spatial locality of our encoded ASTs sorted to put sensitive—source and sink—APIs spatially close. The last layer is flattened into a vector whose dimension is the number of classes. As we formulate our problem as a multi-label classification problem, we use Sigmoid as the activation function of the last dense layers. The neural network is trained on a given dataset by minimizing the standard binary cross entropy loss. Figure 3 depicts the CNN architecture used in FAM.



```
①public String postData () {
 ③String urlString ="www.website.com";
   OutputStream out = null;
 ⑤if(1 == call_my_func()){②
     URL url = new URL(urlString);
     HttpURLConnection urlConnection = url.openConnection();
     out = newBufferedOutputStream(urlConnection.getOutputStream());
     BufferedWriter writer =
       new BufferedWriter(new OutputStreamWriter(out, "UTF-8"));④
     writer.write("Hello!");④
     urlConnection.connect();
   }
}
```

```
Func() {
  if(NUMBER == User_Defined_Func()){
    new URL(STRING);
    url.openConnection();
    newBufferedOutputStream(urlConnection.getOutputStream());
    new BufferedWriter(new OutputStreamWriter(OutputStream, STRING));
    writer.write(STRING);
    urlConnection.connect();
  }
}
```

FIGURE 2. An example of the normalization process.

VII. EVALUATION

In this section, we evaluate our system, FAM, using the real-world malware dataset (subsection V-A) and open-source dataset (Drebin [8]). The deep learning model used in our experiment used the standard convolutional neural network (CNN) mentioned in (section VI).

A. EXPERIMENT SETUP

We conducted our experiments on Ubuntu 18.04 using four GPUs of NVIDIA GeForce RTX 2080 Ti, and 256 GB RAM. We implemented FAM by using tensorflow-gpu v1.14.0, Keras v2.2.4, CUDA v11.2, and Androguard v3.4.0a1 for extracting ASTs. For the parameters of the two Conv1D layers of FAM, we apply (1, 3) filter with ReLU activation function. In the two dense layers of FAM, we set the the dimension of hidden states as 32 and 4 respectively. We use the Adam optimizer with learning rate 0.001 and batch size 64. We randomly shuffle data of each label in the dataset and split 80% for training and the rest 20% for testing. We trained our model for 6 epochs for each experiment.

B. EVALUATION METRICS

To measure the classification performance of our neural network mode using the string feature of malware, we use the following metrics: micro-recall(1), micro-precision(2), micro-F1(3), and accuracy(4). Accuracy is the most intuitive indicator of the model performance, but since the dataset used

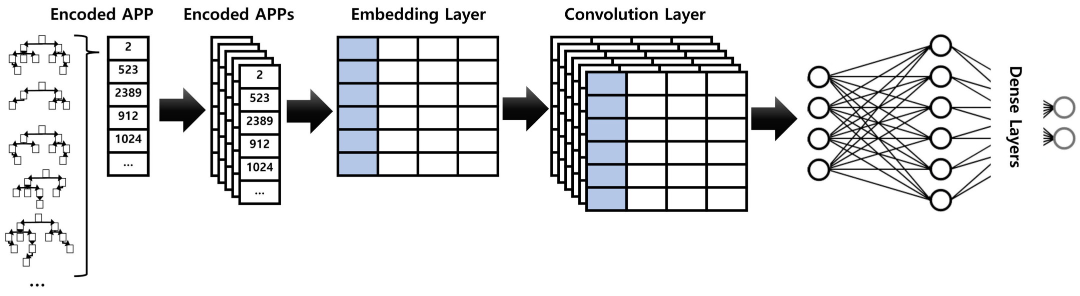
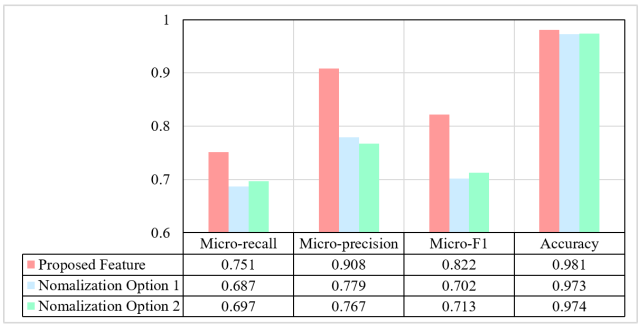
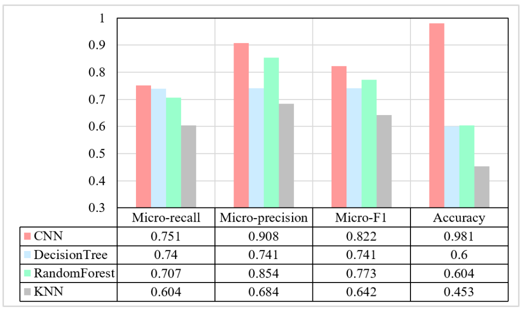


FIGURE 3. The CNN model architecture of FAM.



	Micro-recall	Micro-precision	Micro-F1	Accuracy
🟥 Proposed Feature	0.751	0.908	0.822	0.981
🟦 Nomalization Option 1	0.687	0.779	0.702	0.973
🟩 Nomalization Option 2	0.697	0.767	0.713	0.974

FIGURE 4. The effectiveness of the proposed features. The normalization option 1 is when we did not normalize string constants. The normalization option 2 is when we did not normalize string and numeric constants.



	Micro-recall	Micro-precision	Micro-F1	Accuracy
🟥 CNN	0.751	0.908	0.822	0.981
🟦 DecisionTree	0.74	0.741	0.741	0.6
🟩 RandomForest	0.707	0.854	0.773	0.604
▨ KNN	0.604	0.684	0.642	0.453

FIGURE 5. Comparison result between our CNN model and the machine learning algorithms.

in our experiments is unbalanced for each class, F1 Score and micro average were additionally used as evaluation metrics. We believe that these metrics clearly show the effectiveness of multi-label classification analysis results. The evaluation metrics we used can be mathematically expressed as follows:

$$recall_{micro} = \frac{\sum_{classes} TP}{\sum_{classes} TP + \sum_{classes} FN} \tag{1}$$

$$precision_{micro} = \frac{\sum_{classes} TP}{\sum_{classes} TP + \sum_{classes} FP} \tag{2}$$

$$F1_{micro} = 2 * \frac{precison_{micro} * recall_{micro}}{precison_{micro} + recall_{micro}} \tag{3}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4}$$

TABLE 5. Confusion matrix for the evaluation result of the real-word dataset.

		Predicted Classes	
		Positive	Negative
Actual Classes	Positive	5,681	1,875
	Negative	573	84,698

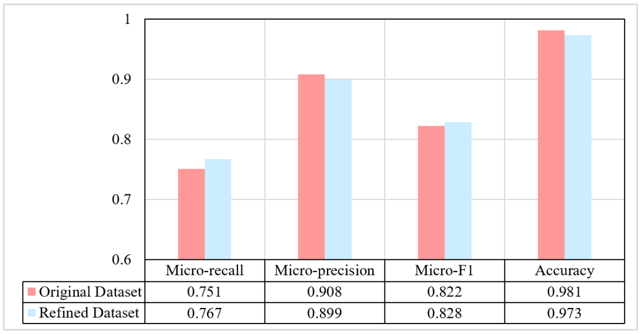
C. EFFECTIVENESS USING THE REAL-WORLD DATASET

To demonstrate the effectiveness of each string feature, we first evaluate the CNN model's performance by using the metrics as in subsection VII-B. Also, we compare it with other models generated by the same dataset, but, when we did *not* completely normalize ASTs (we removed third-party libraries and potentially benign functions as discussed in subsection V-B).

Figure 4, Table 5 shows the results. Our neural network's micro-F1 score (0.822) and the accuracy (0.981) demonstrate that the proposed implementational feature can be importantly used to classify multi-labelled in-the-wild malware. In addition, as the other models' performance results show, the normalization process considerably affects to the model's performance (especially on the micro-F1 score). In the other words, local variables, numeric constants and string constants are not very important to construct the implementational feature of Android malware, and to identify malicious behaviors based on the feature, because they can vary depending on each malware's environmental factors. Finally, We compared the effectiveness of our CNN model with three machine learning algorithms (Decision Tree, Random Forest, and KNN) by using the proposed feature as input. Figure 5 shows that our CNN model has higher scores in all four evaluation metrics than the three machine learning algorithms.

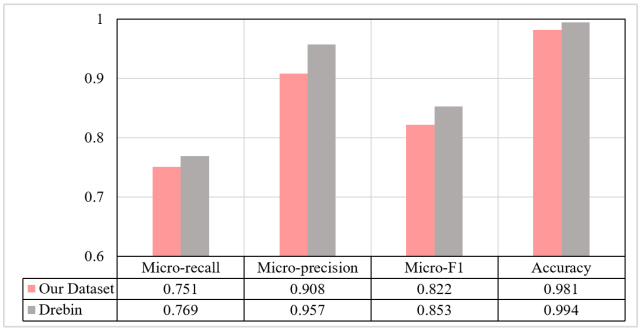
D. EFFECTIVENESS USING REFINED DATASET

Figure 6 shows experimental results when we did not use labels which have less than 10 applications. In our dataset, there are 6 such labels out of 21 kinds of single labels and we have 74 labels left in total after deleting the 6 labels. As the Figure 6 illustrates, even though we deleted the labels, the performance difference between the original dataset and the refined dataset is negligible. This result indicates that our



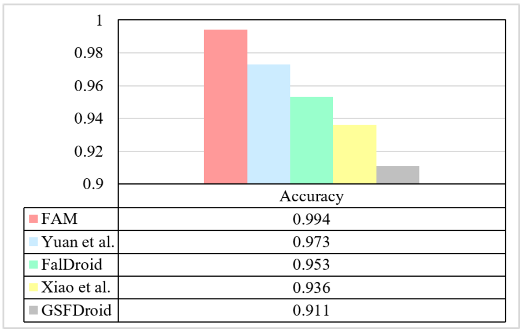
	Micro-recall	Micro-precision	Micro-F1	Accuracy
🟥 Original Dataset	0.751	0.908	0.822	0.981
🟦 Refined Dataset	0.767	0.899	0.828	0.973

FIGURE 6. Comparison of the DNN model's performance between when we used the original dataset and when we deleted labels which has less than 10 applications.



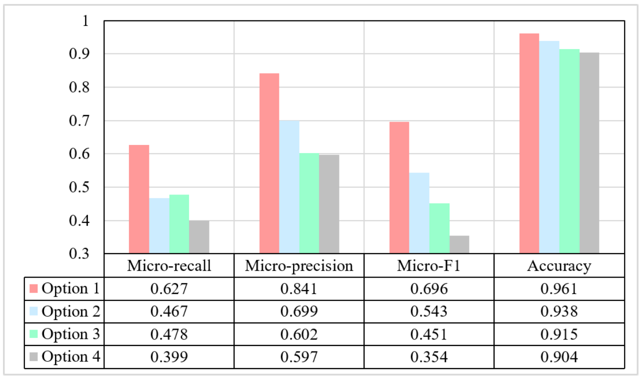
	Micro-recall	Micro-precision	Micro-F1	Accuracy
🟥 Our Dataset	0.751	0.908	0.822	0.981
⬛ Drebin	0.769	0.957	0.853	0.994

FIGURE 7. The evaluation result on a popular, public Android malware dataset, Drebin [8].



	Accuracy
🟥 FAM	0.994
🟦 Yuan et al.	0.973
🟩 FalDroid	0.953
🟨 Xiao et al.	0.936
⬛ GSFDroid	0.911

FIGURE 8. Comparison of the accuracy with four previous approaches.



	Micro-recall	Micro-precision	Micro-F1	Accuracy
🟥 Option 1	0.627	0.841	0.696	0.961
🟦 Option 2	0.467	0.699	0.543	0.938
🟩 Option 3	0.478	0.602	0.451	0.915
⬛ Option 4	0.399	0.597	0.354	0.904

FIGURE 9. The evaluation results on classifying future malware. Each option is described in subsection VII-F.

string feature can be effectively used when the number of data is low.

E. EFFECTIVENESS USING OPEN SOURCE DATASET

To compare the effectiveness of FAM with state-of-the-art approaches (GSFDroid [24], FalDroid [10], Xiao *et al.* [42] and Yuan *et al.* [44]), we evaluated FAM on a popular, public Android malware dataset, Drebin [8], that contains 5,560 singly-labelled malware samples.

Figure 7 shows that the evaluation result on Drebin dataset using FAM. Overall, the effectiveness of FAM increased when we used Drebin dataset; it achieved 99.4% of accuracy with 0.853 of micro-F1 score. On the other hand, as Figure 8 illustrates, FAM showed the highest accuracy among three approaches. Unfortunately, the other approaches did not report micro-recall, -precision, and -F1 scores, we could not concisely compare the performance of FAM with them. Also, because the Android Genome project[1] widely-used in previous research projects stopped data sharing, we could not conduct more comparisons with the other work.

F. EFFECTIVENESS USING FUTURE MALWARE

We performed another evaluation to check whether FAM can classify *future* malware based on malware sampled appeared in a previous year in our dataset. In this evaluation, we first divided our dataset by year. We, then, generated a DNN model and evaluated it as follows.

Option 1: Trained a DNN model using data in 2018 and 2019, and tested it against malware samples in 2020.

Option 2: Trained a DNN model using data in 2019, and tested it against malware samples in 2020.

Option 3: Trained a DNN model using data in 2018, and tested it against malware samples in 2019.

Option 4: Trained a DNN model using data in 2018, and tested it against malware samples in 2020.

Figure 9 illustrates the evaluation results and Table 6 shows the ratio of out-of-vocabulary ASTs in each experiment. Across all the options, we could achieve the acceptable accuracy (higher than 90%) but the micro-F1 scores are not high. Especially, in the case of option 4 and option 3, the micro-F1 scores are less than 0.5 due to the high ratio of out-of-vocabulary ASTs. This evaluation reveals that malware tends to change APIs and its implementation structures over time as the Android framework is updated and to evade anti-malware tools [31].

On the other hand, in this work, we did not employ a embedding method to handle *unknown* data, and thus, the classification performance is expected to decrease in proportion to the ratio of *unknown* data. However, since the goal of this work is to find features that can effectively represent different malicious behaviors of malware, we leave

[1]http://www.malgenomeproject.org/

TABLE 6. The ratio of Out-of-Vocabulary (OoV) ASTs in each experiment of subsection VII-F.

	Option 1	Option 2	Option 3	Option 4
OoV Ratio	4%	5%	48%	58%

this limitation as future work. We discuss this limitation in section VIII as well.

VIII. DISCUSSION

A. OUT-OF-VOCABULARY

In this work, we define the out-of-vocabulary problem as a situation where a DNN model meets an AST that did not appear in the training dataset. This problem can happen frequently when analyzing malware, as they evolve over time [31]. To remedy the OOV problem, one could decompose each AST into tokens, akin to decomposing a sentence into words for natural language processing tasks, and use tokens as input features. This way, it is less likely, during the testing phase, that we encounter tokens that are unseen in the training phase.

B. PRE-TRAINED EMBEDDING

One could use a pre-trained embedding to initialize the embedding layer in our neural network. Word2Vec and Struc2Vec are the among the most popular embedding generation techniques [32], [38]. Whether these embedding techniques can generate a representation of an AST suitable for malware analysis, or what corpus of Android applications to use for training an embedding model, is an orthogonal question, which merits further research on its own. Once an embedding is learned, the embedding can easily be tested with different neural network architectures including our CNN, by initializing the embedding layer with pre-trained ones. We performed a preliminary experiment by learning the representations of individual tokens found in each AST using a Word2Vec. Initializing the AST embedding layer with the sum of the token embedding did not, however, yield any better result. Finding an appropriate pre-training objective that is effective on embedding ASTs would be a key to solving this problem.

C. IMBALANCED DATA

In this work, we used an in-the-wild dataset in which there are various types of malicious applications and the number of malware belonged in each type is uneven, and thus, the deep learning model can be biased. To address the biased model, we need to use a data augmentation method such as Generative Adversarial Network (GAN) [16] to generate insufficient number of data in specific classes. Also, we can assign different weights when learning data belonged in a class where the number of samples is small, or we can perform K-fold Cross-Validation when training a model.

D. DATA UPDATE AND MODEL RE-TRAINING

In general, malware detection approaches using machine learning and deep learning models require frequent re-training of models and updating the training dataset with new samples to reflect trends of emerging malware [31], [36]. If the detection model is not updated, the prediction result of malware detection models can yield wrong decisions—it will not be able to detect unknown zero-day malware. Therefore, periodic re-training of the detection models through updating the training dataset is inevitable. Also, as the Android framework is updated and to evade anti-malware tools [31], Android malware tends to change APIs and its implementation structures over time. Another challenge for building a reliable malware detection model that can detect zero-day malware is to detect malware using evasion and obfuscation techniques to avoid detection from the anti-malware systems. We intend to address this challenge in our future work.

IX. CONCLUSION

In this work, we studied how effectively we can represent malicious behaviors of Android malware by formulating the problem of familial analysis as a multi-label classification problem. To this end, we proposed string features extracted by tokenizing abstract syntax trees (AST) of functions in malware. Our evaluation results demonstrate that the best-performing features can effectively classify recent, real-world malware, with an accuracy up to 98% and a micro F1-score of 0.82.

REFERENCES

[1] (2017). *SuSi*. [Online]. Available: https://github.com/secure-software-engineering/SuSi

[2] (2020). *Androguard* [Online]. Available: https://github.com/androguard/androguard

[3] (2020). *VirusShare* [Online]. Available: https://virusshare.com/

[4] (2020). *VirusTotal*. [Online]. Available: https://www.virustotal.com/gui/home/upload, 2020.

[5] (2021). *Gson*. [Online]. Available: https://github.com/google/gson

[6] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Springer, 2013, pp. 86–103.

[7] K. Aktas and S. Sen, "Updroid: Updated Android malware and its familial classification," in *Proc. Nordic Conf. Secure IT Syst.* Springer, 2018, pp. 352–368.

[8] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.

[9] S. Arzt, "Static data flow analysis for Android applications," Tech. Rep., 2017.

[10] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 8, pp. 1890–1905, Aug. 2018.

[11] M. Fan, X. Luo, J. Liu, C. Nong, Q. Zheng, and T. Liu, "CTDroid: Leveraging a corpus of technical blogs for Android malware analysis," *IEEE Trans. Rel.*, vol. 69, no. 1, pp. 124–138, Mar. 2020.

[12] M. Fan, X. Luo, J. Liu, M. Wang, C. Nong, Q. Zheng, and T. Liu, "Graph embedding based familial analysis of Android malware using unsupervised learning," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 771–782.

[13] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand, "Automated synthesis of semantic malware signatures using maximum satisfiability," 2016, *arXiv:1608.06254*.

[14] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *Proc. ACM workshop Artif. Intell. Secur.*, Nov. 2013, pp. 45–54.

[15] L. Gong, Z. Li, F. Qian, Z. Zhang, Q. A. Chen, Z. Qian, H. Lin, and Y. Liu, "Experiences of landing machine learning onto market-scale mobile malware detection," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–14.

[16] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 27, 2014, pp. 1–9.

[17] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "The strengths and behavioral quirks of Java bytecode decompilers," in *Proc. 19th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2019, pp. 92–102.

[18] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4MalDroid: A deep learning framework for Android malware detection based on Linux kernel system call graphs," in *Proc. IEEE/WIC/ACM Int. Conf. Web Intell. Workshops (WIW)*, Oct. 2016, pp. 104–111.

[19] G. Iadarola, F. Martinelli, F. Mercaldo, and A. Santone, "Towards an interpretable deep learning model for mobile malware detection and family identification," *Comput. Secur.*, vol. 105, Jun. 2021, Art. no. 102198.

[20] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic framework for Android malware detection using deep learning," *Digit. Invest.*, vol. 24, pp. S48–S59, Mar. 2018.

[21] Y. Kim, "Convolutional neural networks for sentence classification," 2014, *arXiv:1408.5882*.

[22] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in Android apps," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Eng. (SANER)*, vol. 1, Mar. 2016, pp. 403–414.

[23] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in Android markets," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 335–346.

[24] Q. Li, Q. Hu, Y. Qi, S. Qi, X. Liu, and P. Gao, "Semi-supervised two-phase familial analysis of Android malware with normalized graph embedding," *Knowl.-Based Syst.*, vol. 218, Apr. 2021, Art. no. 106802.

[25] Y. Li, J. Jang, X. Hu, and X. Ou, "Android malware clustering through malicious payload mining," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Springer, 2017, pp. 192–214.

[26] T. Lu, Y. Du, L. Ouyang, Q. Chen, and X. Wang, "Android malware detection based on a hybrid deep learning model," *Secur. Commun. Netw.*, vol. 2020, pp. 1–11, Aug. 2020.

[27] Z. Ma, H. Ge, Z. Wang, Y. Liu, and X. Liu, "Droidetec: Android malware detection and malicious code localization through deep learning," 2020, *arXiv:2002.03594*.

[28] A. Mahindru and A. L. Sangal, "MLDroid—Framework for Android malware detection using machine learning techniques," *Neural Comput. Appl.*, vol. 33, no. 10, pp. 5183–5240, May 2021.

[29] N. Marastoni, A. Continella, D. Quarta, S. Zanero, and M. D. Preda, "GroupDroid: Automatically grouping mobile malware by extracting code similarities," in *Proc. 7th Softw. Secur., Protection, Reverse Eng. Softw. Secur. Protection Workshop*, Dec. 2017, pp. 1–12.

[30] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android malware by building Markov chains of behavioral models," 2016, *arXiv:1612.04433*.

[31] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving Android malware for auditing anti-malware tools," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 365–376.

[32] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.

[33] S. Millar, N. McLaughlin, J. M. del Rincon, and P. Miller, "Multi-view deep learning for zero-day Android malware detection," *J. Inf. Secur. Appl.*, vol. 58, May 2021, Art. no. 102718.

[34] S. Millar, N. McLaughlin, J. M. del Rincon, P. Miller, and Z. Zhao, "Dan-droid: A multi-view discriminative adversarial network for obfuscated Android malware detection," in *Proc. 10th ACM Conf. Data Appl. Secur. Privacy*, 2020, pp. 353–364.

[35] D. C. Nguyen, E. Derr, M. Backes, and S. Bugiel, "Up2Dep: Android tool support to fix insecure code dependencies," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2020, pp. 263–276.

[36] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *Proc. 28th USENIX Secur. Symp. (USENIX Security)*, 2019, pp. 729–746.

[37] Z. Ren, H. Wu, Q. Ning, I. Hussain, and B. Chen, "End-to-end malware detection for Android IoT devices using deep learning," *Ad Hoc Netw.*, vol. 101, Apr. 2020, Art. no. 102098.

[38] L. F. R. Ribeiro, P. H. P. Saverese, and D. R. Figueiredo, "Struc2vec: Learning node representations from structural identity," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2017, pp. 385–394.

[39] S. Sebastián and J. Caballero, "AVclass2: Massive malware tag extraction from AV labels," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2020, pp. 42–53.

[40] V. Sihag, M. Vardhan, P. Singh, G. Choudhary, and S. Son, "De-lady: Deep learning based Android malware detection using dynamic features," *J. Internet Services Inf. Secur. (JISIS)*, vol. 11, no. 2, pp. 34–45, 2021.

[41] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Proc. 7th Asia Joint Conf. Inf. Secur.*, Aug. 2012, pp. 62–69.

[42] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and LSTM," *Multimedia Tools Appl.*, vol. 78, no. 4, pp. 3979–3999, 2019.

[43] K. Xu, Y. Li, R. H. Deng, and K. Chen, "DeepRefiner: Multi-layer Android malware detection system applying deep neural networks," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Apr. 2018, pp. 473–487.

[44] B. Yuan, J. Wang, D. Liu, W. Guo, P. Wu, and X. Bao, "Byte-level malware classification based on Markov images and deep learning," *Comput. Secur.*, vol. 92, May 2020, Art. no. 101740.

[45] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: Deep learning in Android malware detection," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 371–372.

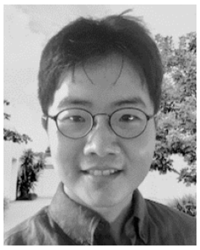
[46] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 1105–1116.

[47] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 95–109.

[48] D. Zhu, H. Jin, Y. Yang, D. Wu, and W. Chen, "DeepFlow: Deep learning-based malware detection by mining Android application for abnormal usage of sensitive data," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jul. 2017, pp. 438–443.

YOUNGHOON BAN received the B.S. degree from the Department of Computer Engineering, Chungwoon University, Incheon, South Korea, in 2020. He is currently pursuing the master's degree with the School of Software Convergence, Soongsil University. Currently, he is also a Research Staff at the Cyber Security Research Center. His research interests include mobile application security, mobile platform security, and machine learning.

SUNJUN LEE received the B.S. and M.S. degrees in computer science and engineering from Soongsil University, in 2019 and 2021, respectively. His research interests include binary analysis, reverse engineering, system security, and mobile security.



DOKYUNG SONG received the B.S. degree from the Department of Electrical and Computer Engineering, Seoul National University, and the M.S. and Ph.D. degrees from the Department of Computer Science, University of California at Irvine. He is an Assistant Professor with the Department of Computer Science and the Director of Cyber Security Laboratory, Yonsei University, Seoul, South Korea. His research interests include understanding and finding security vulnerabilities in systems software.

HAEHYUN CHO received the Ph.D. degree in computer science and with a focus on information assurance from the School of Computing, Informatics and Decision Systems Engineering, Arizona State University. He is an Assistant Professor with the School of Software and the Co-Director of the Cyber Security Research Center, Soongsil University. His research interests include systems security to discover and mitigate security concerns and analyzing, finding, and resolving security issues in a wide range of topics.

JEONG HYUN YI (Member, IEEE) received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, South Korea, in 1993 and 1995, respectively, and the Ph.D. degree in information and computer science from the University of California at Irvine, in 2005. He was a Principal Researcher at the Samsung Advanced Institute of Technology, South Korea, from 2005 to 2008, and a Member of Research Staff at the Electronics and Telecommunications Research Institute (ETRI), South Korea, from 1995 to 2001. From 2000 to 2001, he was a Guest Researcher at the National Institute of Standards and Technology (NIST), Maryland, U.S. He is a Professor at the School of Software and the Director of Cyber Security Research Center, Soongsil University, Seoul. His research interests include mobile security and privacy, IoT security, and applied cryptography.