HAWK: Rapid Android Malware Detection Through Heterogeneous Graph Attention Networks

Yiming Hei, Renyu Yang, *Member, IEEE*, Hao Peng, Lihong Wang, Xiaolin Xu, Jianwei Liu, Hong Liu, Jie Xu, *Member, IEEE*, and Lichao Sun

***Abstract*— Android is undergoing unprecedented malicious threats daily, but the existing methods for malware detection often fail to cope with evolving camouflage in malware. To address this issue, we present HAWK, a new malware detection framework for evolutionary Android applications. We model Android entities and behavioral relationships as a heterogeneous information network (HIN), exploiting its rich semantic meta-structures for specifying implicit higher order relationships. An incremental learning model is created to handle the applications that manifest dynamically, without the need for reconstructing the whole HIN and the subsequent embedding model. The model can pinpoint rapidly the proximity between a new application and existing in-sample applications and aggregate their numerical embeddings under various semantics. Our experiments examine more than 80 860 malicious and 100 375 benign applications developed over a period of seven years, showing that HAWK achieves the highest detection accuracy against baselines and takes only 3.5 ms on average to detect an out-of-sample application, with the accelerated training time of 50× faster than the existing approach.**

***Index Terms*— Android, graph representation learning, heterogeneous information network (HIN), malware detection.**

I. INTRODUCTION

WITH the highest market share worldwide on mobile devices, Android is experiencing unprecedented dependability issues. Due to Android's extensibility and openness of development, users are put at high risk of a variety of threats and illegal operations from malicious software, i.e., malware including privacy violations, data leakage, and advertisement spams. Common vulnerabilities and exposures (CVEs) reveal 414 Android vulnerabilities that can be easily attacked in realistic environments. This phenomenon calls for more reliable and accessible detection techniques.

Conventionally, Android Applications (Apps) are analyzed by either static analysis through predetermined signatures/semantic artifacts or dynamic analysis through multilevel instrumentation [1]. However, static analysis could become invalid by simple obfuscation, whereas dynamic analysis heavily depends on OS versions and the Android runtime, which is inherently cost-expensive and time-consuming. To tackle this, numerous machine learning-based detection techniques [2]–[8] typically leverage feature engineering to extract key malware features and apply classification algorithms—each app is represented as a vector—to distinguish benign software from malicious software. Nevertheless, these approaches often fail to capture emerging malware that either conducts evolving camouflage and attack type or hides certain features deliberately.[1] Hence, it is imperative to build an inductive and rapid mechanism for constantly capturing software evolution and detecting malware without heavily relying on domain-specific feature selection.

Graph neural network (GNN), which is used to model the relationship between entities, is developing rapidly in both theoretical [9]–[12] and applied fields [13], [14]. Heterogeneous information network (HIN) [15], [16], as a special case of GNN, has been widely adopted in many areas such as operating systems, the Internet of Things, and cybersecurity by exploiting the abundant node and relational semantic information before embedding into representation vectors [17]–[20]. More specifically, in the context of malware detection, if App_1 and App_2 share permission `SEND_SMS`, while App_2 and App_3 share permission `READ_SMS`, HIN is able to capture the implicit semantic relationship between App_1 and App_3 that can be hardly achieved by feature engineering-based approaches. HIN-based modeling is even more meaningful because malware developers are extremely difficult to hide such implicit relationships [18]. While promising, HIN is inherently concerned about static networks/graphs [21]. The complication is, however, how to efficiently embed the out-of-sample nodes (i.e., incoming nodes out of the established HIN). Considering the continuous software updates and the huge volume of Apps, it is impossible to involve all Apps in the stage of HIN construction and inefficient

Manuscript received March 15, 2021; revised June 13, 2021; accepted August 13, 2021. This work was supported in part by NSFC under Grant 62002007, Grant U20B2053, Grant 62073012, and Grant 62072184; in part by the S&T Program of Hebei Province under Grant 20310101D; in part by the Fundamental Research Funds for the Central Universities; in part by the Project of Science and Technology Commitment of Shanghai under Grant 20511106002; in part by U.K. Engineering and Physical Sciences Research Council (EPSRC) under Grant EP/T01461X/1; in part by U.K. White Rose University Consortium; and in part by the Opening Project of Shanghai Trusted Industrial Control Platform. *(Renyu Yang and Yiming Hei are co-first authors.) (Corresponding author: Hao Peng.)*

Yiming Hei, Hao Peng, and Jianwei Liu are with the School of Cyber Science and Technology, Beihang University, Beijing 100083, China (e-mail: penghao@buaa.edu.cn; liujianwei@buaa.edu.cn).

Renyu Yang and Jie Xu are with the School of Computing, University of Leeds, Leeds LS2 9JT, U.K. (e-mail: r.yang1@leeds.ac.uk; j.xu@leeds.ac.uk).

Lihong Wang and Xiaolin Xu are with the National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing 100029, China (e-mail: wlh@isc.org.cn; xxl@isc.org.cn).

Hong Liu is with the School of Computer Science and Software Engineering, East China Normal University, Shanghai 200241, China, and also with Shanghai Trusted Industrial Control Platform Company Ltd., Shanghai 200062, China (e-mail: liuhong@ticpsh.com).

Lichao Sun is with the Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA 18015 USA (e-mail: james.lichao.sun@gmail.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNNLS.2021.3105617.

Digital Object Identifier 10.1109/TNNLS.2021.3105617

[1]https://www.mcafee.com/blogs/other-blogs/mcafee-labs

to reconstruct the entire embedding model when new Apps are seen emerging. This drawback impedes the practicality and the scale this native technique can perform. Although AiDroid [21] attempts to tackle this problem and represents each out-of-sample App with convolutional neural network (CNN) [22], it requires heavily multiple convolution operations resulting in nonnegligible time inefficiency.

In this article, we present HAWK, a novel Android malware detection framework with the aid of network representation learning model and HIN to explore abundant but hidden semantic information among different Apps. In particular, we extract seven types of Android entities—including App, permission, permission type, Application Programming Interface (API), class, interface, and `.so` file—from the decompiled Android application package (APK) files and establish an HIN mainly through transforming entities and their relationships into nodes and edges, respectively. We exploit rich semantic meta-structures as the templates to define relation sequence between two entity types. This includes both meta-path [23] and meta-graph [24] that can specify the implicit relationships among heterogeneous entities. A certain meta-structure corresponds to an adjacency matrix associated with a homogeneous graph. The graph only contains App nodes and is the target in the procedure of malware detection.

The numerical embedding of all App entities is at the core of HAWK, which can be then fed into a binary classifier. In particular, HAWK involves two distinct learning models for in-sample and out-of-sample nodes, respectively. To embed an in-sample App, we propose MSGAT, a meta-structure guided graph attention network (GAT) mechanism [25] that incorporates its neighbors' embedding within any meta-structure and integrates the embedding results of different meta-structures into the final node embedding. This design considers not only the informative connectivity of neighbor nodes but also the diverse semantic implications over different entity relationships. In addition, to efficiently embed an out-of-sample App, we present MSGAT++, a new incremental learning model upon MSGAT to make good use of the embedding of certain existing nodes. Given a specific meta-structure and its corresponding graph, our model first pinpoints a specific set of in-sample App nodes that are most similar to the target new node, before aggregating their embedding vectors to form the node embedding under this meta-structure. Likewise, we entitle particular weights to individual embedding vector of each meta-structure and aggregate them to obtain the final embedding. This incremental design can quickly calculate the embedding based on the established HIN structures without relearning the holistic embedding for all nodes, thereby significantly improving the training efficiency and model scalability.

We demonstrate the effectiveness and efficiency of HAWK based on 80 860 malicious and 100 375 benign Apps collected and decompiled across VirusShare, CICAndMal, and Google AppStore. Experiments show that HAWK outperforms all baselines in terms of accuracy and F1 score, indicating its effectiveness and suitability for malware detection at scale. It takes merely 3.5 ms on average to detect an out-of-sample App with accelerated training time of 50× against the native approach that rebuilds the HIN and reruns the MSGAT. To enable replication and foster research, we make HAWK publicly available at `github.com/RingBDStack/HAWK`. This article makes the following contributions.

1) It examines more than 200 000 Android Apps and decompiled more than 180 000 APKs, spanning over seven years across multiple open repositories. This discloses abundant data source to establish the HIN and uncovers the hidden high-order semantic relationships among Apps (Section III).
2) It presents a meta-structure guided attention mechanism based on HIN for node embedding, by fully exploiting neighbor nodes within and across meta-structures (Section IV-A). Experiments show that the capture of semantics can support excellent forward and backward compatible detection capabilities.
3) It proposes an incremental aggregation mechanism for rapidly learning the embedding of out-of-sample Apps, without compromising the quality of numerical embedding and detection effectiveness (Section IV-B).

Organization: Section II shows the motivation and outlines the system overview. Section III discusses the procedure of feature engineering and data reshaping by leveraging HIN, while Section IV details the core techniques to tackle in-sample and out-of-sample malware detection. Experimental setup and results are presented in Sections V and VI. Related work is discussed in Section VIII before we conclude this article and discuss the future work.

II. BACKGROUND AND OVERVIEW

A. Motivation and Problem Scope

The Android platform is increasingly exposed to various malicious threats and attacks. As malware detection for Android systems is a response-sensitive task, our work addresses two primary research challenges—inductive capability and detection rapidness. Anomaly identification should allow for forecasting new applications that we have not seen (the so-called out-of-sample Apps) and rapidly catch up the up-to-date malicious attacks and threats, particularly considering the vast diversity and rapid growth of emerging malicious software.

The detection procedure is typically regarded as a binary classification. Formally, we aim to take as input features $\mathcal{X}$ of Android Apps and their previous labels (malicious/benign) $\mathcal{T}$ to predict the type t of any target App either old or new. Unfortunately, the existing approaches for malware detection are inadequate in tackling inductive problems where new application is arbitrary and unseen beforehand. Most of the prior works on network embedding [23], [24], [26], [27] are transductive, i.e., if a new data point is added to the testing dataset, one has to thoroughly retrain the learning model. Hence, malware detection is in great need of a generic inductive learning model where any new data would be predicted, based on an observed set of training set, without the need to rerun the whole learning algorithm from scratch.

B. Our Approach of HAWK

1) Key Idea: We consider this problem as a semisupervised learning based on graph embedding. The first innovation of our approach, as a departure from prior work, is to encode the information as a structured HIN [15], [16] in which nodes depict entities and their characteristics. An HIN is a graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{R})$ with an entity-type mapping $\phi : \mathcal{V} \to \mathcal{A}$ and a relationship-type mapping $\psi : \mathcal{E} \to \mathcal{R}$, where $\mathcal{V}$ and $\mathcal{E}$ represent node and edge set, respectively.

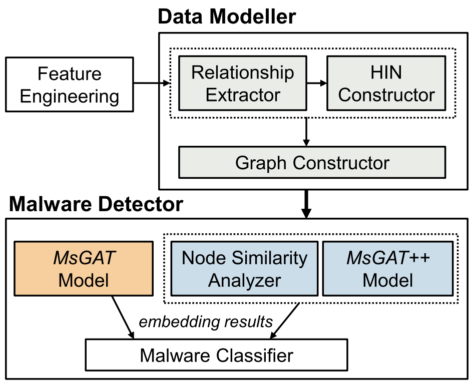

Fig. 1. HAWK architecture overview.

$\mathcal{A}$ and $\mathcal{R}$ denote the type set of nodes and edge, respectively, where $|\mathcal{A}| + |\mathcal{R}| > 2$. Edges represent the relationships between a pair of entities (e.g., an App owns a specific permission or a permission belongs to a permission type). Since the detection problem is App entity oriented, it is effective to deduce the information from a self-contained HIN to homogeneous relational subgraphs that can be directly absorbed by GNN. As the fundamental requirement of graph embedding is to obtain the graph structure, we need to calculate the adjacency matrix from the constructed HIN—the best option to reflect the proximity and the node connectivity in the graph. GNN models can be subsequently carried out to learn the numerical embedding for in-sample App nodes. To underpin the continuous embedding learning for out-of-sample nodes, the learning model is desired to make the best use of the embedding result of the existing in-sample App nodes, in an incremental manner.

2) Architecture Overview: Fig. 1 shows the HAWK's architecture, encompassing data modeler and malware detector components. Specifically, relationship extractor in data modeler first offers an extraction of Android entities based on feature engineering—massive Android Apps are compiled and investigated. There are seven types of nodes ("App" together with six characteristics) and six types of edges. HIN constructor then builds up the HIN by organizing entities and the extracted relationships into nodes and edges of HIN (Section III-B). App graph constructor is responsible for generating homogeneous relational subgraphs from HIN that only contains App entities. This is enabled by employing meta-structures, including both meta-path [23] and meta-graph [24] (Section III-C).

Malware detector then involves two distinct representation learning models to numerically embed in-sample and out-of-sample nodes. It is in great need of fully exploiting node affinities within a given meta-structure and aggregates the embeddings of the same node under different meta-structures. Specifically, we design separate strategies to learn the embedding.

1) To represent in-sample App nodes, the proposed MSGAT, a meta-structure enabled GAT solution, first aggregates intra-meta-structure attention aggregation mechanism for accumulating the embedding of a target node among its neighbor nodes within the graph pertaining to a certain meta-structure. In the second inter-meta-structure phase, we further fuse the obtained embedding among different meta-structures so that their semantic meanings can be represented in the final embedding (Section IV-A).
2) To efficiently tackle the out-of-sample node embedding, we generate the embedding, incrementally, for a new node through reusing and aggregating the embedding result of selective in-sample App nodes in close proximity to the target node. This requires the model to ascertain the similarity between the existing in-sample App nodes and the target node. Similarly, the embedding is first gathered at neighbor node level under a given meta-structure before conducting the inter-meta-structure aggregation (Section IV-B).

Malware classifier digests the learned vector embeddings to learn a classification model to determine whether a given App is malicious or benign and then validates its effectiveness. General-purpose techniques, such as random forest (RF), logistic regression (LR), and support vector machine (SVM), can be adopted as the classifier implementation. We select the training set from in-sample Apps to train our classifier while using the testing set from in-sample Apps and all out-of-sampling Apps to test the models.

III. HIN-BASED DATA MODELING

A. Feature Engineering

An Android application needs to be packaged in the APK format and installed on Android system. An APK file contains code files, the configuration AndroidManifest.xml file, the signature and verification information, the lib (the directory containing platform-dependent compiled codes), and other resource files. To better analyze Android Apps, reverse tools (e.g., APKTool[2]) are widely leveraged to decompile the APK files so that the `.dex` source file can be decompiled into a `.smali` file. To describe the key characteristics of an App, we extracted the following six types of entities.

1) *Permission (P):* The permission determines specific operations that an App can perform. For example, only Apps with `READ_SMS` permission can access the user's email information.
2) *Permission Type (PT):* The permission type[3] describes the category of a given permission. Table I outlines the permission types and representative permissions.
3) *Class (C):* Class is an abstract module in Android codes, where APIs and variables can be directly accessed. HAWK uses the class name in `.smali` codes to represent a class.
4) *API:* It provisions the callable function in the Android development environment.
5) *Interface (I):* The interface refers to an abstract data structure in Java. We extract the name from `.samli` files.
6) *.so file (S):* `.so` file is Android's dynamic link library, which can be extracted from the decompiled lib folder.

Following this methodology, we downloaded over 200 000 APKs from open repositories and after de-duplication and decompilation, and 181 235 APKs are finally filtered and extracted; 63 902 entities are then selected according to [3]. This provisions abundant data sources for establishing the HIN and mining intrinsic semantics.

[2]https://ibotpeaches.github.io/Apktool
[3]https://developer.android.google.cn/guide/topics/permissions

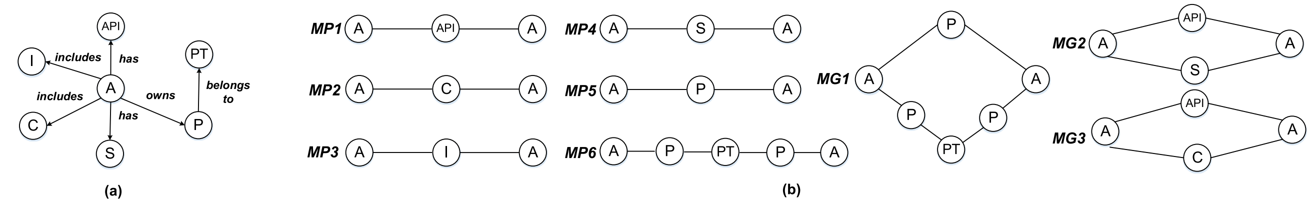

Fig. 2. (a) Meta-schema. (b) Meta-structure.

TABLE I
CATEGORIES OF REPRESENTATIVE PERMISSIONS

Type	Representative Permissions
NORMAl	ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE
CONTACTS	WRITE_CONTACTS, GET_ACCOUNTS
PHONE	READ_CALL_LOG, READ_PHONE_STATE,
CALENDAR	READ_CALENDAR, WRITE_CALENDAR
LOCATION	ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION
STORAGE	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE
SMS	READ_SMS, RECEIVE_MMS, RECEIVE_SMS

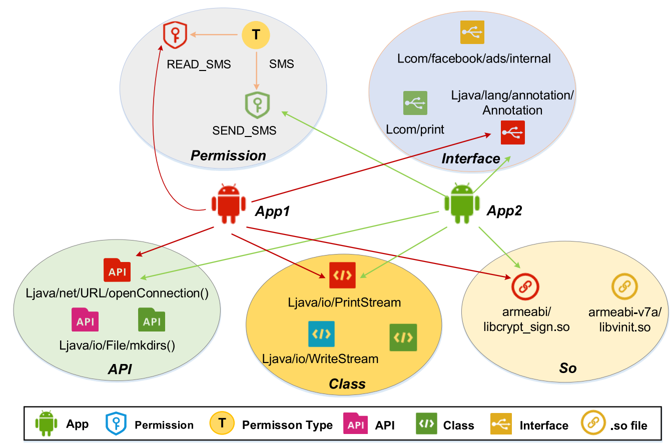

Fig. 3. Example of Android HIN that contains two Android Apps.

B. Constructing HIN

1) Extracting Entity Relationships Into an HIN: Meta-schema is a meta-level template that defines the relationship and type constraints of nodes and edges in the HIN. As shown in Fig. 2(a), we figure out a meta-schema that can encode necessary relationships between Android entities. Based on the domain knowledge, we elaborately examine the following inherent semantic relationships.

1) *[R1] App-API:* It indicates that an App has a specific API. Using the relationship between App and API is effective to dig out and represent the link between two Apps [18].
2) *[R2] App-Permission:* It specifies that an App owns a specific permission. Apps with permissions, such as `READ_SMS`, `SEND_SMS`, and `WRITE_SMS`, are strongly correlative [3]. If `SEND_SMS` is shared between App_1 and App_2 and `READ_SMS` is shared between App_2 and App_3, an implicit association between App_1 and App_3 is highly likely to manifest.
3) *[R3] Permission-PermissionType:* It describes that the permission belongs to a specific permission type. Normally, permissions can be categorized into different types.[4]
4) *[R4] App-Class:* It means that the App includes a specific class in the external SDK. A malware tends to generate instances by using classes in a vicious SDK.[5]
5) *[R5] App-Interface:* It indicates that the App includes the specific interface in the external SDK.
6) *[R6] App-.so:* It denotes that the App has a specific `.so` file. Fan *et al.* [17] demonstrated the effectiveness of associating dynamic link libraries with software in Windows system.

[4]https://developer.android.google.cn/guide/topics/permissions
[5]https://research.checkpoint.com/2019/simbad-a-rogue-adware-campaign-on-google-play

Fig. 3 shows an HIN that contains two Apps and their semantic relationships. For instance, App_1 has API `Ljava/net/URL/openConnection`. Both App_1 and App_2 own the Class `Ljava/io/PrintStream`." The permission `READ_SMS` belongs to the permission type `SMS`" and so on.

2) Storing Entity Relationships: We use a relation matrix to store each relationship individually. For instance, we generate an matrix $\mathbb{A}$ where the element $\mathbb{A}_{i,j}$ denotes if App_i contains API_j. Intuitively, the transpose of a matrix depicts the backward relationship, e.g., API_j belongs to App_i. As summarized in Table II, six matrices are used to represent and store the relationships **[R1]–[R6]**. Nevertheless, it is necessary to obtain the connectivity between two Apps if there are sophisticated semantic links, i.e., higher order relationships.

C. Constructing App Graph From HIN

To form a homogeneous graph that only contains App nodes, the key step is to incorporate the relationship between App entity and other entities into the combined connectivity between Apps. To ascertain the hidden higher order semantic, we mainly calculate Apps' proximity via exploiting a meta-path or meta-graph within a given HIN and then obtain the node adjacency matrix for the graph. In other words, given a meta-structure, the HIN can be converted to an exclusive homogeneous graph in which each node has meta-structure-specific neighbor nodes.

In fact, a meta-path connects a pair of nodes with a semantically meaningful relationship. We enrich the meta-structures further to involve the meta-graph—in the form of directed acyclic graph (DAG)—that can be used as an extended template to capture arbitrary but meaningful combination of existing relationships between a pair of nodes. In effect, a meta-structure provides a filter view to extract a homogeneous

TABLE II
DESCRIPTIONS OF RELATION MATRICES

Relation	Matrix	Description
R1	$\mathbb{A}$	if App i *contains* the API j, $a_{i,j}$ is 1; otherwise 0.
R2	$\mathbb{P}$	if App i *has* the permission j, $\mathbb{P}_{i,j}$ is 1; otherwise 0.
R3	$\mathbb{T}$	if the type of permission i is j, $\mathbb{T}_{i,j}$ is 1; otherwise 0.
R4	$\mathbb{C}$	if App i owns the Class j, $\mathbb{C}_{i,j}$ is 1; otherwise 0.
R5	$\mathbb{I}$	if App i uses the interface j, $\mathbb{I}_{i,j}$ is 1; otherwise 0
R6	$\mathbb{S}$	if App i calls the so file j, $\mathbb{S}_{i,j}$ is 1; otherwise 0.

node graph, in which all nodes satisfy particular complicated semantics. Arguably, depending upon different meta-structures, nodes will be organized distinctly within different graphs. To some extent, each graph can be regarded as a subgraph of the holistic HIN under a certain view—each subgraph satisfies the semantic constraints given by the meta-structure.

1) Meta-Structures: We leverage domain knowledge from system security expertise to elaborately pick up meta-structures for covering the inherent relationships. We first combine all possible meaningful semantic meta-structures and then carefully select those meta-structures with sufficient precision through numerous experiments. The detailed procedure is discussed in Section VI-C. As shown in Fig. 2(b), we eventually present six meta-paths and three meta-graphs that can effectively outline the structural semantics and capture rich relationships between two Android Apps in the HIN. For example, A-P-A describes the relationship where two Apps have the same permission ($\mathcal{MP}_5$) and A-P-PT-P-A indicates two Apps co-own the same type of permission ($\mathcal{MP}_6$). $\mathcal{MG}_2$ simultaneously combines A-API-A with A-S-A. Accordingly, the semantic constraints will be tightened, i.e., the selected nodes have to satisfy all predefined constraints. Nevertheless, models [28], [29] without the manual design of original meta-structures could also be applied into our scheme.

2) Homogeneous App Graph for Each Meta-Structure: Performing a sequence of matrix operations over the modeled relationship matrices, we can precisely calculate the adjacency of nodes within a graph. For a given meta-path $\mathcal{MP}$, $(A_1, \ldots, A_n)$, the adjacency matrix can be calculated by

$$\Psi^{\mathcal{MP}} = R_{A_1A_2} \cdot R_{A_2A_3} \cdots R_{A_{n-1}A_n} \tag{1}$$

where $R_{A_jA_{j+1}}$ is the relation matrix between entity A_j and A_{j+1} (one instance of **[R1]** to **[R6]** in Table II). For example, the adjacency matrix for the graph under $\mathcal{MP}_1$ A-API-A is $\Psi^{\mathcal{MP}_1} = \mathbb{A} \cdot \mathbb{A}^T$. $\Psi_{i,j} > 0$ indicates that App$_i$ and App$_j$ are associated with each other, i.e., they are neighbors based on the meta-path $\mathcal{MP}_1$. Specifically, the value represents the count of meta-path instances, i.e., the number of pathways, between nodes i and j. Likewise, for a given meta-graph $\mathcal{MG}$, a combination of several meta-paths, i.e., $(\mathcal{MP}_1, \ldots, \mathcal{MP}_m)$, the node adjacency matrix is

$$\Psi^{\mathcal{MG}} = \Psi^{\mathcal{MP}_1} \odot \cdots \odot \Psi^{\mathcal{MP}_m} \tag{2}$$

where $\odot$ is the operation of Hadamard product. For instance, $\mathcal{MG}_2$, the adjacency matrix can be calculated by $\Psi^{\mathcal{MG}_2} = (\mathbb{A} \cdot \mathbb{A}^T) \odot (\mathbb{S} \cdot \mathbb{S}^T)$. By conducting graph modeling for each meta-structure, the original HIN is converted to multiple App homogeneous graphs, each of which pertains to an adjacency matrix. Given K meta-structures, we have a collection of K adjacency matrices, i.e., $\{\Psi^{\mathcal{M}_1}, \ldots, \Psi^{\mathcal{M}_K}\}$.

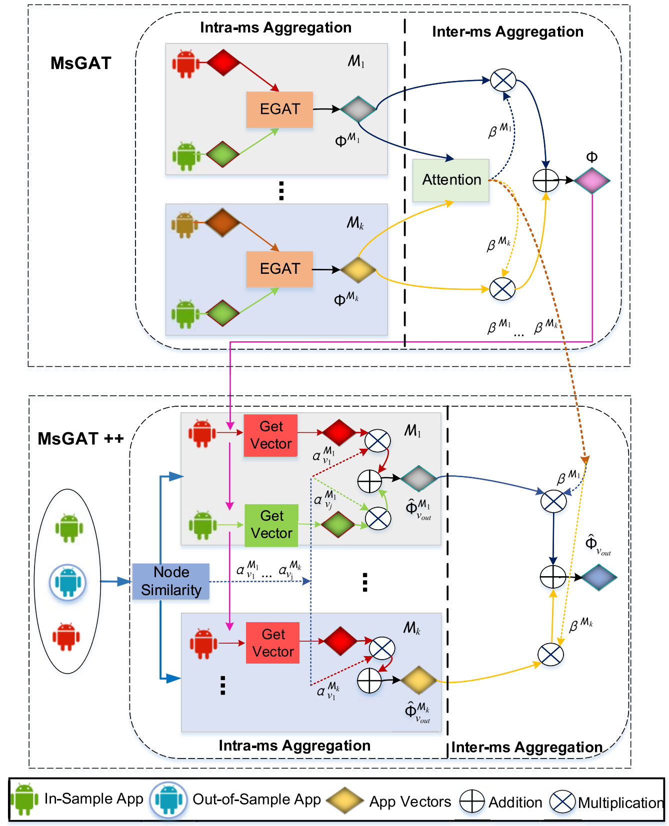

Fig. 4. MSGAT and MSGAT++ models for node embedding.

IV. NODE EMBEDDING MODELS

A. MSGAT*: In-Sample Node Embedding*

We introduce a series of innovative GAT optimizations enhanced by meta-structures—we employ the attention mechanism [25] among neighbor nodes within a given meta-structure (intra-ms) and coordinate the attention among different meta-structures (inter-ms). Fig. 4 shows the flowchart of our models and important notations used in the models are outlined in Table III.

1) Intra-ms Aggregation: Intra-ms aggregation learns how a node pays different attention to its neighbor nodes in a graph pertaining to a meta-structure. Formally, it aggregates the neighbors' representation vectors with weights considering the feature information of entities and the edge information between entities. To do so, we initially encode the vector of each in-sample App in the form of one-hot and concatenate them into a matrix H. $H_{i\cdot}$, the ith row of H, represents the embedding vector of the ith App node. Thereafter, we design an edge weight-aware GAT (EGAT) model to combine H and the adjacency matrix pertaining to a given meta-structure $\mathcal{M}_k$. To implement the EGAT model, feature information and edge weight information are fully utilized to aggregate features from neighbors. More specifically, we first construct the adjacency matrix $\Psi^{\mathcal{M}'_k}$ with a normalization operation

$$\Psi^{\mathcal{M}'_k} = \text{Normalize}(H \cdot H^T \odot \Psi^{\mathcal{M}_k}) \tag{3}$$

TABLE III
SYMBOL NOTATIONS

Symbol	Definition
$\mathcal{M}_k$, $\mathcal{MP}$, $\mathcal{MG}$	kth meta-structure, a meta-path or meta-graph
$R_{A_iA_j}$	Relation matrix between two entities in the HIN
$Sim_{\mathcal{M}_k}(v_i, v_j)$	The similarity value between node v_i and node v_j under meta-structure $\mathcal{M}_k$
$\mathbb{X}_{\mathcal{M}_k}$	Similarity matrix under meta-structure $\mathcal{M}_k$
$\Psi^{\mathcal{M}_k}$	Adjacency matrix under $\mathcal{M}_k$ that can depicts node connectivity in a homo graph
$\widehat{\Psi}^{\mathcal{M}_k}$	incremental segment of the adjacency matrix, connecting in-sample nodes to new nodes
$\Phi^{\mathcal{M}_k}$	Embedding matrix under $\mathcal{M}_k$; each single row $\Phi_i^{\mathcal{M}_k}$ represents the vector embedding for ith node
$\widehat{\Phi}^{\mathcal{M}_k}$	Embedding matrix under $\mathcal{M}_k$ for new nodes

and elements in $\Psi^{\mathcal{M}'_k}$ that are lower than a predefined threshold τ (τ is set to be 0.1 in our model) will be set to zero. Thereafter, we update $\Phi^{\mathcal{M}_k}$ with the GAT model [10]

$$\Phi^{\mathcal{M}_k} = \mathrm{GAT}(H; \Psi^{\mathcal{M}'_k}). \tag{4}$$

Eventually, the low-dimensional vector embedding for all in-sample App nodes, in a form of matrix $\Phi^{\mathcal{M}_k}$ with a collection of row vectors, can be obtained in this stage.

We then repeatedly calculate the vector matrix for all predefined meta-structures and obtain a collection of embedding vectors, i.e., $[\Phi^{\mathcal{M}_1}, \ldots, \Phi^{\mathcal{M}_K}]$, where K is the totality of meta-structures. Concretely, the embedding matrix $\Phi^{\mathcal{M}_k}$ is of shape $L \times D$, where L denotes the number of in-sample Apps in the HIN and D denotes the dimension of each App vector. As a result, the embedding of App_i node can be identified as the ith row, i.e., $\Phi_{i\cdot}^{\mathcal{M}_k}$.

2) Inter-ms Aggregation: Since each meta-structure provisions an individual semantic view, we propose an inter-ms attention aggregation to integrate embedding $[\Phi^{\mathcal{M}_1}, \ldots, \Phi^{\mathcal{M}_K}]$ under different semantics and thus enhance the quality of node embedding. Specifically, we exploit a multilayer perceptron (MLP) procedure for learning the weight $\beta^{\mathcal{M}_k}$ of each meta-structure $\mathcal{M}_k$ in the fusion

$$(\beta^{\mathcal{M}_1}, \ldots, \beta^{\mathcal{M}_K}) = \mathrm{softmax}(\mathrm{NN}(\Phi^{\mathcal{M}_1}), \ldots, \mathrm{NN}(\Phi^{\mathcal{M}_K})) \tag{5}$$

where NN is a native neural network that maps a given matrix to a numerical value. Consequently, the final embedding for all in-sample App nodes can be obtained through adding up the weighted representation matrices

$$\Phi = \sum_{k=1}^{K} \beta^{\mathcal{M}_k} \cdot \Phi^{\mathcal{M}_k}. \tag{6}$$

We then pass Φ on to another neural network so that the loss function between the neural network's outputs and ground-truth labels can be calibrated via iterative backpropagation.

B. MSGAT++: *Incremental Embedding*

To best embed unknown Apps not included in the training procedure, we present MSGAT++, an increment learning mechanism for utilizing the in-sample embedding already learned from MSGAT to rapidly represent those out-of-sample Apps. To make clear, we use v_{out} to generally stand for any out-of-sample node out of the HIN.

1) Exploring Node Similarity: Pinpointing the underlying connections between new nodes and existing nodes in the HIN plays a pivotal role in providing rapid numerical representation and cost-effective malware detection. To do so, it is imperative to calculate and accumulate the similarity between v_{out} and existing nodes. Following similar methodology presented in [30], the node similarity between node v_i and node v_j under a given meta-path is defined as

$$\mathrm{Sim}^{\mathcal{MP}}(v_i, v_j) = \frac{2 * \Psi_{ij}^{\mathcal{MP}}}{\Psi_{ii}^{\mathcal{MP}} + \Psi_{jj}^{\mathcal{MP}}} \tag{7}$$

where $\Psi_{ij}^{\mathcal{MP}}$ implies the number of meta-structures between two connected nodes and, thus, a higher similarity indicates a tighter association between these two nodes. Accordingly, the node similarity between nodes v_i and v_j under a meta-graph $\mathcal{MG}$ is

$$\mathrm{Sim}^{\mathcal{MG}}(v_i, v_j) = \mathrm{Sim}^{\mathcal{MP}_1}(v_i, v_j) \odot \cdots \odot \mathrm{Sim}^{\mathcal{MP}_m}(v_i, v_j). \tag{8}$$

2) Incremental Aggregation for Embedding Learning: The initial task is to catch the incremental relationships and construct the graph information. Within a given meta-structure, we aim to only update an adjacency matrix that quantifies the connectivity between the out-of-sample nodes and existing in-sample App nodes. This should be done in an incremental manner to reduce the training cost. In practice, we first repeat the steps aforementioned in Section III-B to calculate all relation matrices in Table II merely for out-of-sample App nodes. Second, we concatenate the relation matrices of new App nodes and those of existing App nodes to form an incremental segment of the node adjacency $\widehat{\Psi}^{\mathcal{M}_k}$—a pathway from an in-sample App node to a new node. Take $\mathcal{MP}_1$ as an example; we first obtain the relation matrix $\mathbb{A}_{\mathrm{out}}$ for all new nodes and then generate the matrix by $\widehat{\Psi}^{\mathcal{M}_1} = \mathbb{A}_{\mathrm{in}} \cdot \mathbb{A}_{\mathrm{out}}^T$. This design ensures that the incremental adjacency matrix $\widehat{\Psi}^{\mathcal{M}_k}$ can function independently of the established adjacency matrix $\Psi^{\mathcal{M}_k}$, while they together serve as the holistic abstract of connectivity among all nodes.

We propose MSGAT++ to entitle numerical embedding to new nodes while calibrating the existing node's representation. Similar to MSGAT, the model consists of two steps: intra-ms and inter-ms aggregation. Given a semantic meta-structure $\mathcal{M}_k$, we substitute $\widehat{\Psi}^{\mathcal{M}_k}$ into (7) or (8) to calculate $\mathrm{Sim}^{\mathcal{M}_k}(v_j, v_{\mathrm{out}})$, the similarity between a new node v_{out} and any in-sample App node v_j. Repeating this for all out-of-sampling nodes and all in-sample App nodes forms a similarity matrix $\mathbb{X}^{\mathcal{M}_k}$ where a larger value inherently indicates a closer proximity between two nodes. Accordingly, we can obtain a collection of similarity matrix for all meta-structures $\{\mathbb{X}^{\mathcal{M}_1}, \ldots, \mathbb{X}^{\mathcal{M}_K}\}$.

Arguably, to better represent the new node in a numerical vector, we should fully aggregate existing embedding results of existing nodes in close proximity to the new node. To this end, we select top-σ in-sample App nodes $(v_{n1}, \ldots, v_{n\sigma})$, based on the similarity matrix $\mathbb{X}^{\mathcal{M}_k}$, and aggregate their vectors for the embedding of the new node

$$\widehat{\Phi}_{v_{\mathrm{out}}}^{\mathcal{M}_k} = \sum_{s=1}^{\sigma} \alpha_{v_{\mathrm{ns}}}^{\mathcal{M}_k} \cdot \Phi_{v_{\mathrm{ns}}}^{\mathcal{M}_k} \tag{9}$$

Algorithm 1 Incremental embedding algorithm in MsGAT++

Input: An out-of-sample App v_{out}
Output: v_{out}'s vector embedding $\widehat{\Phi}_{v_{out}}$ and the updated embedding matrix Φ for existing in-sample App nodes

```
for k ∈ {1, ..., K} do
    // select σ in-sample App nodes with the highest similarity
    {v_n1, ..., v_nσ} ← DescendSort(X^{M_k}).topK(σ)
    // Calculate the weights
    {α_{v_1}^{M_k}, ..., α_{v_σ}^{M_k}} ← Eq.10
    // Calculate the embedding of v_out under M_k
    Φ̂_{v_out}^{M_k} ← Eq.9.
end for
// Embedding fusion from all meta structures
Φ̂_{v_out} ← Eq. 11
return Φ̂_{v_out}, Φ
```

where $\alpha_{v_j}^{\mathcal{M}_k}$ denotes the weight of the node v_j ($v_j \in (v_{n1}, \ldots, v_{n\sigma})$) under $\mathcal{M}_k$ and $\widehat{\Phi}$ implies the incremental embedding information for the out-of-sample node exclusively. The weight can be easily calculated by

$$\alpha_{v_j}^{\mathcal{M}_k} = \frac{\text{Sim}^{\mathcal{M}_k}(v_{\text{out}}, \quad v_{\text{ns}})}{\sum_{s=1}^{\sigma} \text{Sim}^{\mathcal{M}_k}(v_{\text{out}}, v_{\text{ns}})}. \tag{10}$$

Eventually, we recalibrate the embedding by conducting inter-ms aggregation over K individual representations under all meta-structures

$$\widehat{\Phi}_{v_{\text{out}}} = \sum_{k=1}^{K} \beta^{\mathcal{M}_k} \cdot \widehat{\Phi}_{v_{\text{out}}}^{\mathcal{M}_k} \tag{11}$$

where $\beta^{\mathcal{M}_k}$ can be obtained from (5) (in fact, to improve the performance of our model, we need to fine-tune these weights). Algorithm 1 outlines the whole procedure of our rapid incremental embedding learning in the malware detection.

3) Time Complexity: Algorithm 1 demonstrates a simple but efficient approach with an acceptable complexity. The overall complexity is $\mathcal{O}(KLN\log N)$, where K and L are the number of meta-structures and the number of out-of-sample Apps, respectively, and N represents the number of in-sample Apps.

V. Experiment Setup

A. Methodology

1) Environment: HAWK is evaluated on a 16-node GPU cluster, where each node has a 64-core Intel Xeon CPU E5-2680 v4@2.40 GHz with 512 GB RAM and 8 NVIDIA Tesla P100 GPUs, Ubuntu 20.04 LTS with Linux kernel v.5.4.0. HAWK depends upon tensorflow-gpu v1.12.0 and scikit-learn v0.21.3. ApkTool and aapt.exe are used for parsing Apps.

2) Datasets: According to the aforementioned discussion of feature engineering in Section III-A, we overall decompiled 181 235 APKs (i.e., 80 860 malicious Apps and 100 375 benign Apps) from 2013 to 2019. With the help of AndroZoo,[6] benign Apps are primarily collected from Google Play store while malicious Apps are obtained from VirusShare and CICAndMal. To validate the compatibility, both forward and backward, of the proposed model in HAWK, we train our model based on Apps released in 2017 (amid the seven time spans) and then utilize it to detect Apps published from 2013 to 2019.

[6]https://androzoo.uni.lu

TABLE IV
DESCRIPTIONS OF EVALUATION METRICS

Metrics	Description
TP	The number of malicious Apps that are correctly identified
TN	The number of benign Apps that are correctly identified
FP	The number of benign Apps that are mistakenly identified
TN	The number of malicious Apps that are mistakenly identified
$Precision$	$TP/(TP+FP)$
$Recall$	$TP/(TP+FN)$
$FP\text{-}Rate$	$FP/(FP+TN)$
$F1$	$2 * Precision * Recall/(Precision + Recall)$
Acc	$(TP+FN)/(TP+TN+FP+FN)$

Specifically, we extracted 14 000 benign and 9865 malicious Apps released in 2017, as in-sample Apps, to construct the HIN and train the detection model. For generating the out-of-sample sample data, we collected seven malware subsets (v2013–v2019), each of which contains roughly 10 000 samples, from VirusShare over consecutive seven years, together with another two subsets from CICAndMal, including 242 scarewares/adwares samples in 2017 (c2017) and 253 samples in 2019 (c2019). Meanwhile, we extracted benign Apps to match the same number of benign Apps in each subset above.

3) Methodology and Metrics: The experiments are threefold. We first evaluate the effectiveness of HAWK against traditional feature-based ML approaches and numerous baselines in terms of in-sample and out-of-sample scenarios (Section VI-A). Afterward, we demonstrate the efficiency of HAWK by comparing the training time consumption with other approaches (Section VI-B). We further conduct several microbenchmarkings, including an ablation analysis of performance gains, an evaluation of meta-structure's importance, and the impact of the sampled neighbor number on detection precision (Section VI-C).

We use metrics precision, recall, false positive (FP) rate, F1, and accurate to measure the effectiveness (see Table IV) and use time consumption to measure the efficiency. The execution time includes the process of generating embedding vectors and detecting Apps while excluding the process of extracting Apps relation matrix. We use fivefold cross validation and calculate the average accuracy to provide an assurance of unbiased and accurate evaluation.

B. Baselines

To evaluate the performance of MsGAT in HAWK, the baselines encompass generic models and specific models used by some well-known malware detection systems.

1) Generic Models: We first implement the following generic models as comparative approaches.

1) *Node2Vec [31]:* It is a typical model generalized from DeepWalk [32] based on homogeneous graph network.
2) *GCN [9]:* It is a semisupervised homogeneous graph convolutional network model that retains feature information and structure information of the graph nodes.
3) *RS-GCN:* It represents the approach to converting the HIN into homogeneous graphs, applying native GCN to

each graph and reporting the best performance among different graphs.

4) *GAT [10]:* It is a semisupervised homogeneous graph model that utilizes attention mechanism for aggregating neighborhood information of graph nodes.
5) *RS-GAT:* It denotes the approach to converting the HIN into homogeneous graphs based on rich semantic meta-structures, applying native GAT to each homogeneous graph and reporting the best performance among different graphs.
6) *Metapath2Vec [23]:* It is a heterogeneous graph representation learning model that leverages meta-path-based random walk to find neighborhood and uses skip-gram with negative sampling to learn node vectors.
7) *Metagraph2Vec [24]:* It is an alternative model to Metapath2Vec; both meta-paths and meta-graphs are applied to the random walk.
8) *HAN [26]:* It is a heterogeneous graph representation learning model that utilizes predefined meta-paths and hierarchical attentions for node vector embedding.

For Node2Vec, GCN, and GAT, we treat all the nodes in HIN as the same type to obtain the homogeneous graph. Since all these models are toward static graphs, we compare the capability of out-of-sample detection between MsGAT++ and three generic strategies that can be easily adopted in any comparative models.

1) *Neighbor Averaging (NA):* It directly averages the vector embedding of the in-sample neighbors pertaining to a given new App as the targeted embedding.
2) *Sampled Neighbor Averaging (SNA):* It further filters the neighbor range by sampling a fixed number of in-sample neighbors based on the sorted node similarity and simply averaging their embedding as the targeted embedding.
3) *Rerunning (RR):* It primarily merges the out-of-sample Apps with in-sample Apps and rebuilds the entire HIN and the malware detection model.

2) Specific Models Deriving From Specialized Systems: Second, we compare our models in HAWK against the following models used by the existing malware detection systems.

1) *Drebin [33]:* It is a framework that inspects a given App by extracting a wide range of features sets from the `manifest` and `dex` code and adopts the SVM model in the classifier.
2) *DroidEvolver [34]:* It is a self-evolving detection system to maintain and rely on a model pool of different detection models that are initialized with a set of labeled Apps using various online learning algorithms. It is worth noting that we do not directly compare against MamaDroid [35] because it has been demonstrated less effective than DroidEvolver.
3) *HinDroid [18]:* It constructs a heterogeneous graph with entities such as App and API and the rich in-between relationships. It aggregates information from different semantic meta-paths and uses multikernel learning to calculate the representations of Apps.
4) *MatchGNet [19]:* It is a graph-based malware detection model that regards each software as a heterogeneous graph and learns its representation. It determines the threat of unknown software primarily through matching the graph representation of the unknown software and that of benign software.
5) *Aidroid [21]:* It is among the first attempts to tackle out-of-sample malware representations with heterogeneous graph model and CNN network. Following the detailed description in this article, we utilize one- and two-hop neighbors to best function its model performance.

TABLE V
F1 VALUE AND ACCURACY OF IN-SAMPLE APPS DETECTION

Metrics	Approaches	20%	40%	60%	80%
F1	Node2Vec	0.8355	0.8378	0.8542	0.8601
	GCN	0.8653	0.8677	0.8721	0.8763
	GAT	0.8435	0.8633	0.8752	0.8801
	Metapath2Vec	0.9231	0.9321	0.9328	0.9395
	RS-GCN	0.9212	0.9510	0.9515	0.9560
	RS-GAT	0.9507	0.9631	0.9653	0.9664
	HAN	0.9511	0.9617	0.9671	0.9705
	Metagraph2Vec	0.9750	0.9766	0.9764	0.9771
	SVM (Drebin)	0.9312	0.9387	0.9446	0.9477
	DroidEvolver	0.9412	0.9517	0.9566	0.9605
	HinDroid	0.9643	0.9669	0.9684	0.9746
	MatchGNet	0.9395	0.9511	0.9604	0.9753
	Aidroid	0.9321	0.9399	0.9414	0.9455
	MsGAT (HAWK)	**0.9857**	**0.9859**	**0.9871**	**0.9878**
Acc	Node2Vec	0.8254	0.8388	0.8405	0.8593
	GCN	0.8558	0.8663	0.8630	0.8692
	GAT	0.8461	0.8645	0.8758	0.8833
	Metapath2Vec	0.9259	0.9321	0.9335	0.9388
	RS-GCN	0.9199	0.9494	0.9527	0.9544
	RS-GAT	0.9486	0.9620	0.9652	0.9664
	HAN	0.9521	0.9657	0.9675	0.9699
	Metagraph2Vec	0.9686	0.9698	0.9748	0.9762
	SVM (Drebin)	0.9295	0.9356	0.9407	0.9455
	DroidEvolver	0.9329	0.9506	0.9557	0.9623
	HinDroid	0.9688	0.9698	0.9722	0.9764
	MatchGNet	0.9302	0.9508	0.9536	0.9689
	Aidroid	0.9227	0.9356	0.9367	0.9437
	MsGAT (HAWK)	**0.9843**	**0.9855**	**0.9867**	**0.9854**

3) Model Parameters: For Node2Vec and Metapath2Vec, we set the number of walks per node, the max walk length, and the window size to be 10, 100, and 8, respectively. For GCN, GAT, and HAN, we set up the parameters suggested by their original papers. For the fairness of comparison, each model will be trained 200 times. The length of embedding vectors delivered by these models is set to be 128.

VI. EXPERIMENT RESULTS

A. Detection Effectiveness

1) In-Sample Malware Detection Against DL Models: We choose 20%, 40%, 60%, and 80% of the in-sample Apps to train the LR model and the residual for testing. Table V shows the F1 and Acc scores of each model. In general, MsGAT can achieve competitive classification accuracy when compared the popular malware detectors such as Drebin, DroidEvolver, MatchGNet, HinDroid, and AiDroid. Compared with F1 and Acc scores, similar observations can be found in Table VI when measuring FP rate. This is because our graph-based representation learning models can fully integrate the feature information of Apps and the implied semantic information between Apps, which improves the expression ability. In addition, the accuracy of RS-GCN and RS-GAT can be improved by over 5% compared with native GCN and GAT. Such approaches convert the original HIN into homogeneous graph and the improvement derives from preserving the

TABLE VI
FP RATE OF IN-SAMPLE APPS DETECTION

Metrics	Approaches	20%	40%	60%	80%
$FP-Rate$	Node2Vec	0.0425	0.0393	0.0388	0.0342
	GCN	0.0350	0.0323	0.0333	0.0318
	GAT	0.0343	0.0334	0.0299	0.0268
	Metapath2Vec	0.0177	0.0175	0.0169	0.0165
	RS-GCN	0.0184	0.0118	0.0109	0.0107
	RS-GAT	0.0115	0.0088	0.0079	0.0075
	HAN	0.0108	0.0098	0.0085	0.0087
	Metagraph2Vec	0.0071	0.0068	0.0059	0.0057
	SVM (Drebin)	0.0163	0.0155	0.0135	0.0139
	DroidEvolver	0.0154	0.0116	0.0101	0.0108
	HinDroid	0.0075	0.0078	0.0071	0.0068
	MatchGNet	0.0193	0.0129	0.0122	0.0081
	Aidroid	0.0184	0.0171	0.0150	0.0139
	MSGAT (HAWK)	**0.0038**	**0.0034**	**0.0032**	**0.0035**

semantic information in the heterogeneous networks through our proposed semantic meta-structures.

It is worth noting that Metagraph2Vec and MSGAT achieve the highest precision, particularly compared against Metapath2Vec and HAN that only involve meta-paths. The accuracy gain, obviously, stems from introducing meta-graphs that bring rich semantics to mine more complex semantic associations. In addition, MSGAT outperforms Metagraph2Vec as our models adopt the aggregation mechanisms for both inter-meta-structure and intra-meta-structure, thereby aggregating semantic information from far more comprehensive views.

2) Out-of-Sample Malware Detection Against DL Models: Tables VII and VIII show the F1 score and FP rate, respectively, when we adopt different in-sample models and out-of-sample policies. Overall, the NA and SNA policies have the lowest detection accuracy under all cases due to the substantial loss of semantic information. Obviously, direct averaging operation ignores the discrepancies among neighbors, thereby reducing the precision of node embedding and the resultant detection effectiveness. It is also observable that NA and SNA have very similar precision in almost all cases. This indicates that sampling a certain number of neighbor nodes is able to achieve approximate information in comparison to averaging all neighbor nodes.

Intuitively, the RR policy will deliver the best performance of detection over all datasets since all data either new or old will involve in embedding retraining. Metagraph2Vec, RS-GAT, and RS-GCN outperform Metapath2Vec, GAT, and GCN due to the benefit from abundant meta-structures, respectively. This performance improvement again demonstrates that applying abundant semantic meta-structures into embedding models can bring a stronger generalization capacity.

As shown in Table VII, MSGAT, together with the rerunning policy, achieves the best detection effectiveness on 2/3 datasets. This can be attributed to the highly rich meta-structures used to include all possible contributions from both intra- and inter- meta-structure aspects. Nevertheless, rerunning has nonnegligible overheads particularly in terms of long training time (we will demonstrate the time consumption later). By contrast, MSGAT++ is proven to be a compromising but competitive solution; the precision of MSGAT++ is in close proximity to the rerunning baselines over all datasets. To demonstrate the generalization, we also implement our MSGAT++ mechanism upon the HAN model. Similarly, the incremental learning scheme makes far better improvements when compared against native NA and SNA, only with neglectable margin from the rerunning baseline.

Hindroid, MatchGNet, HG2Img, and Drebin observably deliver unstable outcomes across different datasets, indicating a limited generalization ability. This is probably because Hin2Img and Hindroid are more dependent upon large training samples and thus has lower precision on some specific datasets. MatchGNet may have limited its performance by neglecting the correlation information between Apps during the construction of the graph. In Drebin, SVM is leveraged as the feature-based machine learning technique, making it difficult to deal with malware with rapidly changing features. DroidEvolver is also based on feature engineering and updates its model in an online manner according to out-of-sample Apps, leading to a competitive classification accuracy. Nevertheless, purely relying on explicit features is intrinsically deficient compared with semantic-rich approaches.

3) Comparison Against Traditional Feature-Based ML Models: We mainly use RF, LR, decision tree (DT), gradient boosting decision tree (GBDT), and AdaBoost as comparative baselines. In this experiment, we particularly use v2017 as the train set to build the HIN while leveraging the out-of-sample Apps with various released times or various sources as the test set. Following the method in [3], we extract information from permission, API, class name, interface name, and `.so` file to construct the feature vector with 63 902 dimensions, which are reduced to 128 dimensions via principal component analysis (PCA).

Fig. 5 shows the F1 score and accuracy score produced by different models over different test sets. Observably, HAWK stably outperforms all traditional baselines in all cases when carrying out the App classification. Traditional ML approaches are competitive (with Acc or F1 score around 0.95) only when the testing set is aligned with the training set (v2017), while HAWK can constantly deliver precise results. Interestingly, the performance of traditional approaches is constantly poor over the dataset of some specific years, e.g., v2014 and c2019. After examining the features involved in the PCA, we infer that the root cause for this phenomenon is because some features are preferably used by malicious Apps in those years but have yet been captured in the training set. For example, "`Ljava/lang/Cloneable`" and the .so file "`libshunpayarmeabi`" manifest in v2014 as the dominating features in the PCA, but they are less important in the principle components in v2017. Similar observations can also be found for the c2019. This is an interesting research finding, while the further in-depth study is currently beyond the scope of this article and will be left for future work.

To sum up, the disparity of precision implies the difficulty in applying traditional ML models—merely relying on explicit feature extraction—into reliable malware detection considering the explosively growing types and numbers of Apps in the market. In comparison, HAWK is able to mine the high-order relations between Apps, with the help of HIN, and thus has strong generalization, i.e., high effectiveness regardless of the type and size of datasets.

B. Detection Efficiency

1) Time Consumption: In this experiment, we compare the time efficiency of our incremental detection design MSGAT++ against those comparative approaches with an

TABLE VII
F1 VALUE OF OUT-OF-SAMPLE APPS DETECTION

Metrics	In-sample Approaches	Out-of-sample Approaches	v2013	v2014	v2015	v2016	v2017	v2018	v2019	c2017	c2019
F1	Node2Vec	NA	0.5888	0.6746	0.6965	0.6740	0.6811	0.6744	0.6680	0.6533	0.6995
		SNA	0.6541	0.6732	0.6965	0.6935	0.6851	0.6665	0.6685	0.6638	0.6845
		Rerunning	0.7564	0.8102	0.7956	0.8124	0.8236	0.7549	0.7968	0.7765	0.7945
	GCN	Rerunning	0.8637	0.8705	0.8459	0.8496	0.8697	0.8743	0.8637	0.8567	0.8537
	GAT	NA	0.7364	0.7423	0.7153	0.7155	0.7545	0.6225	0.7203	0.6352	0.6442
		SNA	0.7433	0.7521	0.7056	0.6962	0.6842	0.7121	0.6831	0.6720	0.6318
		Rerunning	0.8242	0.8448	0.8531	0.8474	0.8731	0.8595	0.8457	0.8511	0.8476
	Metapath2Vec	NA	0.7414	0.8424	0.7835	0.7784	0.7537	0.8243	0.8473	0.8160	0.8183
		SNA	0.7564	0.8531	0.7765	0.7496	0.7365	0.8359	0.8363	0.8242	0.8156
		Rerunning	0.9240	0.9321	0.9195	0.9214	0.9342	0.9326	0.9285	0.9094	0.9052
	HAN	NA	0.7455	0.7405	0.6361	0.7433	0.7292	0.7443	0.7245	0.7101	0.7253
		SNA	0.7593	0.7635	0.7793	0.7723	0.8046	0.7803	0.7566	0.7543	0.7768
		Rerunning	0.9155	0.9626	0.9678	0.9588	0.9758	0.9522	0.9677	0.9482	0.9574
		MsGAT++	0.8896	0.9611	0.9512	0.9462	0.9466	0.9655	0.9583	0.9358	0.9386
	RS-GCN	Rerunning	0.9532	0.9549	0.9487	0.9499	0.9656	0.9651	0.9745	0.9539	0.9471
	RS-GAT	NA	0.7564	0.9400	0.8104	0.6755	0.7345	0.6423	0.7520	0.6152	0.5931
		SNA	0.7564	0.9400	0.8601	0.6744	0.5290	0.7253	0.7323	0.5807	0.7707
		Rerunning	0.9260	0.9321	0.9428	0.9582	0.9498	0.9392	0.9372	0.9485	0.9593
	Metagraph2Vec	NA	0.7658	0.9763	0.8041	0.7955	0.7693	0.8665	0.7614	0.8267	0.8084
		SNA	0.7672	0.7769	0.8155	0.7996	0.7805	0.8665	0.7628	0.8239	0.8084
		Rerunning	0.9533	0.9688	0.9255	0.9382	0.9201	0.9667	0.9718	0.9234	0.9040
	Drebin		0.7442	0.7723	0.7856	0.8277	0.9432	0.7761	0.7891	0.7559	0.7413
	DroidEvolver		0.7972	0.8469	0.8519	0.8996	0.9605	0.9265	0.9028	0.8539	0.8584
	HinDroid		0.8946	0.9232	0.9298	0.9277	0.9712	0.9159	0.9466	0.9396	0.9245
	MatchGNet		0.8981	0.8965	0.9323	0.8833	0.9675	0.9265	0.9053	0.9123	0.9137
	HGiNE (AiDroid)	HG2Img	0.8842	0.9723	0.9556	0.9272	0.9455	0.8761	0.8991	0.8959	0.9013
	MsGAT	NA	0.7693	0.7601	0.6465	0.7725	0.7693	0.7741	0.7741	0.7401	0.7454
		SNA	0.7795	0.7845	0.7996	0.8058	0.8241	0.7955	0.7832	0.7791	0.8071
		Rerunning	**0.9569**	**0.9824**	**0.9876**	**0.9720**	**0.9769**	**0.9808**	**0.9805**	**0.9621**	**0.9693**
		MsGAT++	0.9007	0.9804	0.9736	0.9687	0.9695	0.9665	0.9658	0.9461	0.9393

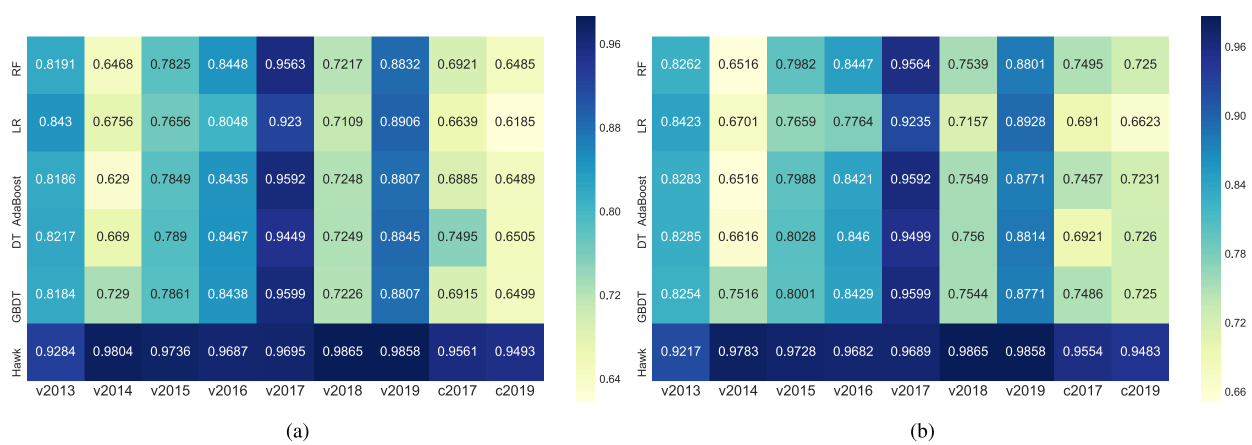

Fig. 5. Comparisons with traditional machine learning methods. (a) F1 Score. (b) Acc Score.

acceptable detection accuracy (demonstrated in Section VI-A), i.e., rerunning HAN, rerunning Metagraph2Vec, Drebin, DroidEvolve, and HG2Img. It is worth mentioning that we exclude the extraction time from calculating the overall execution time for the sake of simplicity because all approaches in our experiment share the same procedure of feature extraction. In fact, it approximately takes 6.9 s per App to extract the feature information from its original APK file.

As observed in Fig. 6, the execution time of MsGAT++ is much shorter than other approaches. MsGAT++ takes

TABLE VIII
FP RATE OF OUT-OF-SAMPLE APPS DETECTION

Metrics	In-sample Approaches	Out-of-sample Approaches	v2013	v2014	v2015	v2016	v2017	v2018	v2019	c2017	c2019
$FP-Rate$	Node2Vec	NA	0.1052	0.0846	0.0819	0.0782	0.0776	0.0846	0.0763	0.0971	0.0819
		SNA	0.0968	0.0831	0.0758	0.0811	0.0862	0.0883	0.0852	0.0806	0.0789
		Rerunning	0.0682	0.0531	0.0576	0.0534	0.0508	0.0698	0.0579	0.0643	0.0569
	GCN	Rerunning	0.0377	0.0359	0.0428	0.0412	0.0366	0.0356	0.0374	0.0394	0.0406
	GAT	NA	0.0711	0.0708	0.0754	0.0736	0.0648	0.0981	0.0727	0.0963	0.0911
		SNA	0.0675	0.0655	0.0779	0.0804	0.0836	0.0754	0.0830	0.0859	0.0966
		Rerunning	0.0461	0.0408	0.0387	0.0403	0.0334	0.0370	0.0406	0.0394	0.0403
	Metapath2Vec	NA	0.0690	0.0419	0.0575	0.0593	0.0655	0.0460	0.0398	0.0474	0.0459
		SNA	0.0616	0.0371	0.0565	0.0634	0.0667	0.0416	0.0415	0.0455	0.0467
		Rerunning	0.0192	0.0173	0.0205	0.0201	0.0167	0.0171	0.0182	0.0230	0.0241
	HAN	NA	0.0644	0.0657	0.0921	0.0650	0.0686	0.0647	0.0701	0.0737	0.0701
		SNA	0.0614	0.0603	0.0563	0.0581	0.0496	0.0559	0.7566	0.0625	0.0568
		Rerunning	0.0215	0.0094	0.0091	0.0104	0.0061	0.0121	0.0081	0.0131	0.0108
		MSGAT++	0.0279	0.0098	0.0123	0.0136	0.0135	0.0087	0.0105	0.0162	0.0165
	RS-GCN	Rerunning	0.0119	0.0115	0.0131	0.0127	0.0087	0.0088	0.0065	0.0117	0.0134
	RS-GAT	NA	0.0619	0.0153	0.0484	0.0822	0.0672	0.0906	0.0628	0.0975	0.1039
		SNA	0.0622	0.0153	0.0358	0.0835	0.1203	0.0702	0.0683	0.1071	0.0585
		Rerunning	0.0189	0.0172	0.0145	0.0106	0.0127	0.0154	0.1586	0.0130	0.0106
	Metagraph2Vec	NA	0.0591	0.0059	0.0494	0.0521	0.0586	0.0339	0.0607	0.0441	0.0485
		SNA	0.0591	0.0565	0.0467	0.0507	0.0556	0.0338	0.0599	0.0444	0.0483
		Rerunning	0.0117	0.0079	0.0188	0.0156	0.0202	0.0084	0.0071	0.0196	0.0242
	Drebin		0.0653	0.0583	0.0547	0.0440	0.0145	0.0572	0.0538	0.0623	0.0653
	DroidEvolver		0.0517	0.0391	0.0376	0.0255	0.0101	0.0187	0.0248	0.0372	0.0365
	HinDroid		0.0241	0.0177	0.0253	0.0157	0.0061	0.0201	0.0149	0.0153	0.0162
	MatchGNet		0.0257	0.0218	0.0137	0.0236	0.0065	0.0156	0.0201	0.0185	0.0173
	HGiNE (AiDroid)	HG2Img	0.0295	0.0071	0.0113	0.0185	0.0139	0.0316	0.0257	0.0265	0.0252
	MSGAT	NA	0.0589	0.0608	0.0895	0.0576	0.0584	0.0572	0.0577	0.0659	0.0648
		SNA	0.0561	0.0549	0.0510	0.0494	0.0448	0.0521	0.0552	0.0563	0.0491
		Rerunning	**0.0109**	**0.0044**	**0.0032**	**0.0071**	**0.0058**	**0.0049**	**0.0049**	**0.0097**	**0.0078**
		MSGAT++	0.0232	0.0049	0.0067	0.0079	0.0077	0.0085	0.0086	0.0136	0.0154

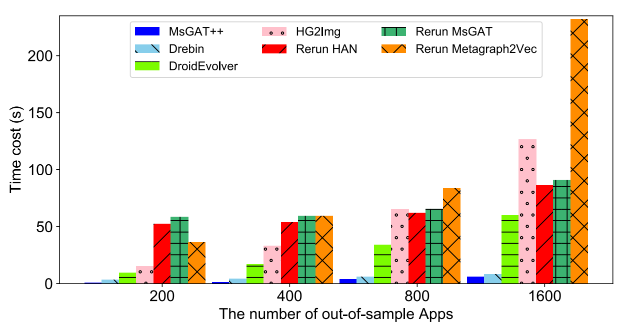

Fig. 6. Efficiency comparison of detecting out-of-sample Apps.

only 3.5 ms on average to detect a single out-of-sample App. This millisecond-level detection by HAWK illustrates its suitability in the real-time malware detection scenario at scale. In particular, MSGAT++ can accelerate the training time by 50× against the native approach that rebuilds the HIN and reruns the MSGAT. The acceleration primarily derives from our incremental learning design that can make full use of previously learned information without the need of rerunning the entire model. In addition, MSGAT++ merely selects a fixed number of neighbor nodes to recalibrate the embedding so that the time consumption only increases linearly with the increment of out-of-sample number.

By contrast, other rerunning HIN-based baselines are predominantly dependent on updating embedding for all nodes based on the starting relation matrix. This leads to discrepancies between MSGAT++ and others with the rerunning policy when tackling out-of-sample Apps. HG2Img relies on a certain amount of update operations to learn new features, resulting in a nonnegligible time consumption.

2) System Overhead: Overall, the overheads are generally low, mainly generated from loading model data and carrying out the multitiered aggregation operations. Runtime memory consumption is typically determined by the number of nodes and features involved in the model training. The total memory consumption of HAWK is roughly 330 MB on average, far lower than the consumption of RR-based baselines (20.88 GB on average). This is because all in-sample and out-of-samples have to fully loaded into memory and involved in the embedding calculation, while our incremental design significantly reduces such costs. Correspondingly, HAWK merely

TABLE IX
ABLATION ANALYSIS

Model	*Acc*	*F*1	AvgDetectionTime
HAWK	0.9695	0.9689	3.5ms
HAWK-I (w/o MSGAT)	0.8731	0.8725	1.8ms
HAWK-R (w/o MSGAT++)	0.9769	0.9769	205ms

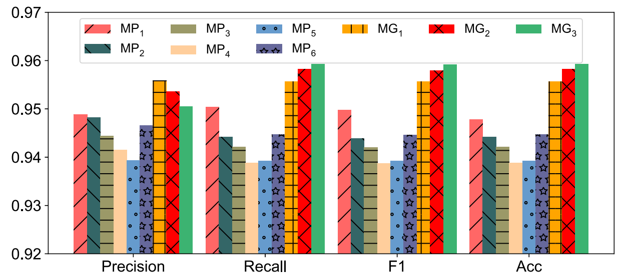

Fig. 7. Model performance under different path combinations.

uses 3.1% additional CPU utilization on average, mainly for sorting out top-σ samples. By contrast, the CPU utilization is up to 76% in rerunning baselines, in which CPU-intensive matrix operations have to be performed. The low system cost also indicates the suitability of applying HAWK into massive-scale malware detection.

C. Microbenchmarking

1) Ablation Analysis: To investigate the impact of each component, we remove one component at a time from our model and study the individual impact on the effectiveness of detecting the out-of-sample Apps. We identify two tailored subsystems: 1) projtitle-I by only retaining native GAT model and removing the hierarchical GAT structure from HAWK and 2) projtitle-R by excluding the incremental design. Table IX reports their accuracy and average time to detect a single App on v2017.

Without multistep and hierarchical aggregation within a meta-structure and across meta-structures, projtitle-I can reduce the average detection time to 1.8 ms. However, both accuracy and F1 score are reduced by 9.9% compared with HAWK. This phenomenon demonstrates the accuracy gain stemming from fusing embedding results under different meta-structures. projtitle-R takes far longer time to detect a malware App, simply because no incremental model is loaded and everything needs to be retrained from scratch. Inherently, although the accuracy experiences a negligible increase due to the full data involved in the model training, the detection efficiency of projtitle-R is still unacceptable considering the long execution time. Hence, it is necessary to adopt the incremental MSGAT++ to ensure a reliable and rapid malware detection.

2) Importance of Meta-Structures: In our model design, a group of meta-paths and meta-graphs are adopted to represent different semantic information. To ascertain the individual contribution to the detection effectiveness, we select a single meta-structure at a time in this experiment. Fig. 7 shows the metric disparities among different meta-structures. More specifically, among all meta-paths, $\mathcal{MP}_1$ and $\mathcal{MP}_4$ have the highest and lowest contributions to the detection precision.

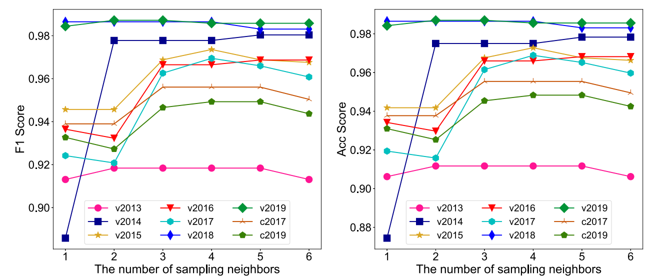

Fig. 8. Impact of sampling neighbor number.

In fact, when analyzing the decompiled codes, we are able to extract far more API information than `.so` files so that the relation matrix $\mathbb{A}$ is denser than $\mathbb{S}$ and thus contains more connection information for node embedding.

Observably, using meta-graphs can achieve higher detection precision when compared to purely using meta-paths, for a combination of meta-paths that can find neighbors with closer affinity. Likewise, if comparing with the results in Table V, MSGAT that involves the full set of semantic meta-structures unsurprisingly outperforms any situation where only a single semantic meta-structure is adopted. This implicates that introducing sophisticated semantics is significantly meaningful to precisely uncover hidden association between entities for better classification.

3) Impact of the Sampling Neighbor Number: As shown in Fig. 8, the precision will first pick up within a certain range but descend once the number of sampling neighbors becomes larger (surpassing four in our experiment setting). In effect, increasing neighbors can provide more relevant and informative embedding for the reference of the new nodes. However, as the neighbors begin to accumulate, noises generated by more irrelevant neighbors will, in turn, negatively impact the embedding aggregation, i.e., diminishing the representation learning effectiveness. This implication reveals that gauging an appropriate number of neighbors is very critical to the holistic performance of embedding incoming Apps and identifying their types. We choose 3–4 neighbors to generate a good enough effectiveness, but one can tune the number either manually according to specific datasets or automatically empowered by reinforcement learning. This is currently beyond the scope of this article and will be left for future work.

4) Case Study of True Negative Detection: The experiments also reveal that the true negative result manifests occasionally. In other words, a small minority of malicious Apps may not be correctly identified by our model. For example, `VirusShare_ecc4c2e7` and `VirusShare_f21ff00cf` in v2013 bypass our detection. An in-depth investigation ascertains that the embedding of such malicious apps will be assimilated by its benign neighbor nodes that are overwhelming in the process of MSGAT++. In fact, since these malicious Apps have far fewer entities (no more than 30 entities) than others (normally with more than 200 entities) used in the training, the neighbors of these malicious apps obtained by HAWK are sparser and tend to be benign Apps, resulting in inaccurate classification. To address this problem, we plan to employ a label-aware neighbor similarity measure based on node attributes to better navigate the neighbor selection and distinguish the malware more efficiently in the future. Nevertheless, HAWK can achieve better detection accuracy against the up-to-date baselines, with far lower time consumption, particularly when detecting the out-of-sample Apps.

VII. DISCUSSION

A. Interpretablity

HAWK is a data-driven modeling and detecting mechanism based on HIN and network representation model empowered by GATs. The model's interpretability can be significantly enhanced due to the inherent nature of rich semantics, stemming from the combinations of meta-paths and meta-graphs, in the HIN and the multitiered aggregation of attention from different semantics. Such an approach intrinsically outperforms the SVM-based approaches such as Drebin [33] and RF-based approaches such as MaMaDroid [35] which has inadequate interpretability.

B. Scalability

The current HIN-based data modeling is scalable and can be easily extended, to any arbitrary entities and relationships, as long as the semantics can be demonstrated beneficial to the process of detection, either by domain knowledge or experimental assessment. In addition, since our design does not require any model rerun, the scalability can be inherently guaranteed when coping with sizable samples.

C. Robustness to Obfuscation

The semantic meta-structures based on multiple entities—including permission, permission types, classes, and interfaces—can overcome the inefficiency of API-alone detection approaches and provide a robust and accurate mechanism for detecting potential malware, in the face of API obfuscation, packing, or dataset skew (e.g., samples with less visible features such as.so files in the dataset v2013). In particular, the multitiered attention aggregation can automatically set the weight of different meta-paths or meta-graphs, thereby substantially reducing the impact of a single factor, e.g., the API obfuscation, on the numerical embedding and increasing the capability of generalization over different datasets and scenarios.

D. Model Aging and Decays

Concept drift (also known as model aging and model decays) usually makes trained models fail to function on new testing samples, primarily due to the changed statistical properties of samples over time. The existing work [36]–[38] measured how a model performs over time facing the concept drift, underpinned the root causes for such drift, and proposed enhanced approaches to improve the model sustainability. However, active learning typically involves massive labeling for tens of thousands of malware samples, usually at a significant cost of human efforts. By far, this issue is not the focus and objective of HAWK. In contrast, MSGAT++ in HAWK aims to rapidly embed and detect the out-of-sample Apps, based upon the existing embedding results, assuming a relatively stable statistical characteristics of the existing Apps. At present, model evolving will be carried out through rerunning of MSGAT, which is demonstrated acceptable in terms of accuracy and time consumption (detailed in Section VI-B). More advanced mechanism for improving the model evolution will be left for future work.

VIII. RELATED WORK

A. Malware Detection Based on Traditional Feature Engineering

Feature engineering and machine learning-based malware detection methods are twofold: static/dynamic feature analysis. Static features analysis approaches [2]–[4], [33]–[35] typically include features including permissions, signatures, and API sequences, and directly employ such machine learning models as RF, SVM, or CNN for malware detection. However, they inevitably overassume that all behaviors reflected by features should be involved within the model training, thereby having inadequate capability of tackling unknown out-of-sample cases and causing much higher FP [3]. Meanwhile, cunning developers can also use obfuscation techniques to hide the malicious codes [7] or perform repackaging attacks [39] to bypass detection. Xu *et al.* [34] can automatically and continually update itself when detecting malware without any human involvement. Nevertheless, this scheme only proves that it has the ability to adapt to updates but does not show its compatibility with previous datasets. In comparison, dynamic feature analysis relies on behavior detection at runtime. Specifically, Dimjaševic *et al.* [5] and Hou *et al.* [6] extracted Linux kernel system calls from Apps executed in Genymotion (Android Virtual Machine), while log analysis [7], [40] and traffic analysis [8], [41] facilitate to capture Apps' real-world behavior. However, it is time-consuming and unrealistic to be applied in malware detection at scale. Other models from natural language processing and image recognition can be customized and reused in malware detection. McLaughlin *et al.* [2] used a deep CNN to analyze raw opcode sequence. Vinayakumar *et al.* [42] transformed the sequences of Android permissions into features by using the long short-term memory (LSTM) layer and uses nonlinear activation function for classification. Xiao *et al.* [43] exploited the LSTM to investigate potential relationships from system call sequences before classification. However, since Apps are constantly updated, explicit features extraction from limited Apps is ineffective in detecting unseen Apps.

B. Malware Detection Based on Graph Networks

Gotcha [17] builds up an HIN and utilizes a meta-graph-based approach to depict the relevance over PE files, which captures both content- and relation-based features of windows malware. HinDroid [18] is primarily on the basis of an HIN built upon relationships between APIs and Apps and employs multikernel SVM for software classification. MatchGNet [19] combines the HIN model with GCN [9] to learn graph representation and node similarity based on the invariant graph modeling of the program's execution behaviors. Wang *et al.* [20] constructed a heterogeneous program behavior graph, particularly for IT/OT systems, and then introduced graph attention mechanism [25] to aggregate information learned through GCN on different semantic paths with weights. However, all these methods are impeded by the static nature of the HIN, i.e., they have limited capability of tackling emerging Apps outside the constructed graph. AiDroid [21] represents each out-of-sample App with CNN [22]. However, the nonnegligible time inefficiency stemming from multiple convolution operations becomes a potential bottleneck. HAWK presents the first attempt to bridge the HIN-based embedding model and GAT to underpin incremental and rapid malware detection particularly for out-of-sample Apps.

IX. Conclusion and Future Work

Malware detection is a critical but nontrivial task particularly in the face of ubiquitous Android applications and the increasingly intricate malware. In this article, we propose HAWK, an Android malware detection framework to rapidly and incrementally learn and identify new Android Apps. HAWK presents the first attempt to marry the HIN-based embedding model with GAT to obtain the numerical representation of Android Apps so that any classifier can easily catch the malicious ones. In particular, we exploit both meta-path and meta-graph to best capture the implicit higher order relationships among entities in the HIN. Two learning models, MsGAT and incremental MsGAT++, are devised to fuse neighbors' embedding within any meta-structure and across different meta-structures and pinpoint the proximity between a new App and existing in-sample Apps. Through the incremental representation learning model, HAWK can carry out malware detection dynamically for emerging Android Apps. Experiments show that HAWK outperforms all baselines in terms of accuracy and time efficiency. In the future, we plan to integrate HAWK to smart mobile devices by devising lightweight and efficient graph convolution models, such as [44] and [45] to replace the existing modules. We also plan to investigate more advanced mechanism for underpinning the model evolving in the face of model decays, particularly in federated learning environments.

Acknowledgment

Renyu Yang would also appreciate the birth of Ruisi and numerous sleepless but encouraging nights with her when preparing this manuscript.

References

[1] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Comput. Surv.*, vol. 50, no. 3, pp. 1–40, Oct. 2017.

[2] N. McLaughlin *et al.*, "Deep Android malware detection," in *Proc. ACM CODASPY*, 2017, pp. 301–308.

[3] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye, "Significant permission identification for machine-learning-based Android malware detection," *IEEE Trans. Ind. Informat.*, vol. 14, no. 7, pp. 3216–3225, Jul. 2018.

[4] S. Hou, A. Saas, Y. Ye, and L. Chen, "Droiddelver: An Android malware detection system using deep belief network based on API call blocks," in *Proc. WAIM*, 2016, pp. 54–66.

[5] M. Dimjaševic, S. Atzeni, I. Ugrina, and Z. Rakamaric, "Evaluation of Android malware detection based on system calls," in *Proc. CODASPY*, 2016, pp. 1–8.

[6] 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. WIC*, Oct. 2016, pp. 104–111.

[7] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "DL-Droid: Deep learning based Android malware detection using real devices," *Comput. Secur.*, vol. 89, Feb. 2020, Art. no. 101663.

[8] S. Wang *et al.*, "Deep and broad URL feature mining for Android malware detection," *Inf. Sci.*, vol. 513, pp. 600–613, Mar. 2020.

[9] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. SIGKDD*, 2017, pp. 1–14.

[10] P. Veliäkoviä, G. Cucurull, A. Casanova, A. Romero, P. Liü, and Y. Bengio, "Graph attention networks," 2017, *arXiv:1710.10903*. [Online]. Available: http://arxiv.org/abs/1710.10903

[11] Z. Hu, Y. Dong, K. Wang, K.-W. Chang, and Y. Sun, "GPT-GNN: Generative pre-training of graph neural networks," in *Proc. SIGKDD*, 2020, pp. 1857–1867.

[12] Y. G. Wang, M. Li, Z. Ma, G. Montufar, X. Zhuang, and Y. Fan, "Haar graph pooling," in *Proc. ICML*, 2020, pp. 9952–9962.

[13] H. Peng *et al.*, "Streaming social event detection and evolution discovery in heterogeneous information networks," *ACM Trans. Knowl. Discovery Data*, vol. 15, no. 5, pp. 1–33, Jun. 2021.

[14] H. Peng *et al.*, "Hierarchical taxonomy-aware and attentional graph capsule RCNNs for large-scale multi-label text classification," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 6, pp. 2505–2519, Jun. 2021.

[15] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu, "PathSim: Meta path-based top-K similarity search in heterogeneous information networks," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 992–1003, 2011.

[16] H. Peng *et al.*, "LIME: Low-cost incremental learning for dynamic heterogeneous information networks," *IEEE Trans. Comput.*, early access, Feb. 11, 2021, doi: 10.1109/TC.2021.3057082.

[17] Y. Fan, S. Hou, Y. Zhang, Y. Ye, and M. Abdulhayoglu, "Gotcha-sly malware!: Scorpion a metagraph2vec based malware detection system," in *Proc. SIGKDD*, 2018, pp. 253–262.

[18] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "Hindroid: An intelligent Android malware detection system based on structured heterogeneous information network," in *Proc. SIGKDD*, 2017, pp. 1507–1515.

[19] S. Wang *et al.*, "Heterogeneous graph matching networks for unknown malware detection," in *Proc. IJCAI*, 2019, pp. 3762–3770.

[20] S. Wang *et al.*, "Attentional heterogeneous graph neural network: Application to program reidentification," in *Proc. ICDM*, 2019, pp. 693–701.

[21] Y. Ye *et al.*, "Out-of-sample node representation learning for heterogeneous graph in real-time Android malware detection," in *Proc. IJCAI*, 2019, pp. 4150–4156.

[22] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. ICLR*, 2015, pp. 1–372.

[23] Y. Dong, N. V. Chawla, and A. Swami, "Metapath2vec: Scalable representation learning for heterogeneous networks," in *Proc. SIGKDD*, 2017, pp. 135–144.

[24] D. Zhang, J. Yin, X. Zhu, and C. Zhang, "Metagraph2vec: Complex semantic path augmented heterogeneous network embedding," in *Proc. PAKDD*, 2018, pp. 196–208.

[25] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[26] X. Wang *et al.*, "Heterogeneous graph attention network," in *Proc. WWW*, 2019, pp. 2022–2032.

[27] H. Zhao, Q. Yao, J. Li, Y. Song, and D. L. Lee, "Meta-graph based recommendation fusion over heterogeneous information networks," in *Proc. SIGKDD*, 2017, pp. 635–644.

[28] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim, "Graph transformer networks," in *Proc. NIPS*, 2019, pp. 11983–11993.

[29] Z. Hu, Y. Dong, K. Wang, and Y. Sun, "Heterogeneous graph transformer," in *Proc. WWW*, 2020, pp. 2704–2710.

[30] Y. Gao, X. Li, H. Peng, B. Fang, and P. Yu, "HinCTI: A cyber threat intelligence modeling and identification system based on heterogeneous information network," *IEEE Trans. Knowl. Data Eng.*, early access, Apr. 20, 2020, doi: 10.1109/TKDE.2020.2987019.

[31] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 855–864.

[32] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2014, pp. 701–710.

[33] 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*, 2014, pp. 23–26.

[34] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: Self-evolving Android malware detection system," in *Proc. IEEE EuroS&P*, Dec. 2019, pp. 47–62.

[35] 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*. [Online]. Available: http://arxiv.org/abs/1612.04433

[36] X. Zhang *et al.*, "Enhancing state-of-the-art classifiers with API semantics to detect evolved Android malware," in *Proc. ACM CCS*, 2020, pp. 757–770.

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

[38] R. Jordaney *et al.*, "Transcend: Detecting concept drift in malware classification models," in *Proc. USENIX Secur.*, 2017, pp. 625–642.

[39] K. Tian, D. Yao, B. G. Ryder, G. Tan, and G. Peng, "Detection of repackaged Android malware with code-heterogeneity features," *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 1, pp. 64–77, Jan. 2020.
[40] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, "Malware detection with deep neural network using process behavior," in *Proc. COMPSAC*, 2016, pp. 577–582.
[41] Z. Li, L. Sun, Q. Yan, W. Srisa-an, and Z. Chen, "Droidclassifier: Efficient adaptive mining of application-layer header for classifying Android malware," in *Proc. SecureComm*, 2016, pp. 597–616.
[42] R. Vinayakumar, K. Soman, and P. Poornachandran, "Deep Android malware detection and classification," in *Proc. ICACCI*, 2017, pp. 1677–1683.
[43] 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.
[44] R. S Srinivasa, C. Xiao, L. Glass, J. Romberg, and J. Sun, "Fast graph attention networks using effective resistance based graph sparsification," 2020, *arXiv:2006.08796*. [Online]. Available: http://arxiv.org/abs/2006.08796
[45] M. Li, Z. Ma, Y. G. Wang, and X. Zhuang, "Fast Haar transforms for graph neural networks," *Neural Netw.*, vol. 128, pp. 188–198, Aug. 2020.

Yiming Hei is currently pursuing the Ph.D. degree with the School of Cyber Science and Technology, Beihang University, Beijing, China.

His research interests include deep learning, information security, and applied cryptography.

Renyu Yang (Member, IEEE) is currently an EPSRC-funded Research Fellow with the University of Leeds, Leeds, U.K. He has industrial experience in building large-scale distributed systems with machine learning and coauthored/co-led many research grants including, U.K. EPSRC, Innovate UK, and EU Horizon 2020. His research interests include distributed systems, resource management, and applied machine learning.

Hao Peng is currently an Assistant Professor at the School of Cyber Science and Technology and the Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing, China. His research interests include representation learning, machine learning, and graph mining.

Lihong Wang is currently a Professor with the National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China. Her current research interests include information security, cloud computing, big data mining and analytics, information retrieval, and data mining.

Xiaolin Xu is currently a Professor with the National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China. Her current research interests include information security, big data mining and analytics, and network security detection.

Jianwei Liu is currently a Professor at the School of Cyber Science and Technology, Beihang University, Beijing, China. His current research interests include information security, communication network, and cryptography.

Hong Liu is currently an Associate Professor with East China Normal University, Shanghai, China, and Shanghai Institute of Intelligent Science and Technology, Tongji University, Shanghai. She is also the Chief Technology Officer of Shanghai Trusted Industrial Control Platform Company Ltd., Shanghai. She has published more than 30 SCI articles, and Google Scholar citations are 2800 times. Her research interests include the security and privacy issues in vehicular edge computing, and the industrial Internet of Things.

Jie Xu (Member, IEEE) is currently the Chair Professor of computing at the University of Leeds, Leeds, U.K., the Leader for a Research Peak of Excellence at Leeds, the Director of U.K. EPSRC WRG e-Science Centre, and the Chief Scientist with BDBC, Beihang University, Beijing, China. He has published in excess of 400 academic articles, book chapters, and edited books. His research interests include large-scale dependable distributed systems, cloud systems, and big data processing.

Prof. Xu is an Executive Board Member of U.K. Computing Research Committee (UKCRC) and a steering/executive committee member for numerous IEEE conferences and led or co-led many research projects to the value of over $30M.

Lichao Sun received the Ph.D. degree from the University of Illinois at Chicago, Chicago, IL, USA, in 2020.

He is currently an Assistant Professor with Lehigh University, Bethlehem, PA, USA. His research interests include deep learning and data mining. He mainly focuses on security and privacy, social networks, and natural language processing applications.