MalFlows: Context-aware Fusion of Heterogeneous Flow Semantics for Android Malware Detection

Zhaoyi Meng, Fenglei Xu, Wenxiang Zhao, Wansen Wang, Wenchao Huang, Jie Cui, Hong Zhong, and Yan Xiong

***Abstract*—Static analysis, a fundamental technique in Android app examination, enables the extraction of control flows, data flows, and inter-component communications (ICCs), all of which are essential for malware detection. However, existing methods struggle to leverage the semantic complementarity across different types of flows for representing program behaviors, and their context-unaware nature further hinders the accuracy of cross-flow semantic integration. We propose and implement *MalFlows*, a novel technique that achieves context-aware fusion of heterogeneous flow semantics for Android malware detection. Our goal is to leverage complementary strengths of the three types of flow-related information for precise app profiling. We adopt a heterogeneous information network (HIN) to model the rich semantics across these program flows. We further propose *flow2vec*, a context-aware HIN embedding technique that distinguishes the semantics of HIN entities as needed based on contextual constraints across different flows and learns accurate app representations through the joint use of multiple meta-paths. The representations are finally fed into a channel-attention-based deep neural network for malware classification. To the best of our knowledge, this is the first study to comprehensively aggregate the strengths of diverse flow-related information for assessing maliciousness within apps. We evaluate MalFlows on a large-scale dataset comprising over 20 million flow instances extracted from more than 31,000 real-world apps. Experimental results demonstrate that MalFlows outperforms representative baselines in Android malware detection, and meanwhile, validate the effectiveness of *flow2vec* in accurately learning app representations from the HIN constructed over the heterogeneous flows.**

I. Introduction

With the highest market share worldwide on mobile operating systems [1] and the openness of development, the Android platform is targeted by various malware and other potentially harmful apps. Such apps steal users' privacy, abuse SMS/CALL, subscribe to premium services silently, *etc*. Meanwhile, the characteristics of Android malware have been evolving over years [2]. The circumstance calls for effective and reliable detection techniques in the Android ecosystem.

Static analysis techniques commonly extract control flows, data flows, and inter-component communications (ICCs) as fundamental products for the examination of inherent semantics within app behaviors to uncover hidden malicious activities [3]–[5]. Control flows determine the execution orders of program statements, data flows trace the paths that data takes through the system from input to output, and ICCs manage the interactions between different app components. The three types of flow-related semantic information are inherently associated with various APIs, involving triggering conditions, data transmissions and communication modes, which serve as crucial indicators for assessing maliciousness of app behaviors. Thus, effectively leveraging these flow-related semantics is essential for achieving precise malware detection.

Existing malware detection schemes have not fully leveraged the synergistic capabilities of heterogeneous flow semantics. In particular, flow-based approaches usually utilize only a part of the information within apps in isolation. For instance, MUDFLOW [6], TriFlow [7], and Complex-Flows [8] detected malware by learning usage characteristics of sensitive data flows. ICCDetector [9] focused on ICC-based features for detection. VAHunt [10] and Difuzer [11] respectively identified malicious virtualization-based apps and logic bombs, using control-flow analysis and taint tracking in separate steps. These approaches can result in incomplete analysis and potentially miss certain malicious behaviors. For non-flow-based methods, the lack of the fine-grained program clues makes it challenging to assess intentions of app behaviors. This limitation applies to conventional [12], machine learning(ML)-based [13]–[16], and large language model(LLM)-based [17] techniques.

To address the problem above, combining the semantics of different types of program flows offers a promising direction for capturing malicious behaviors more effectively. Specifically, each flow type provides a distinct view for characterizing app behaviors. For example, a data-flow path can explicitly trace how sensitive data moves from a source to a sink [4], providing crucial insights into potential data leaks. Each individual view is capable of profiling particular characteristics of malware but is limited by its scope of assessment. Therefore, properly integrating the semantics of multiple flow types holds promise for leveraging their complementary strengths to enhance the understanding of app behaviors and uncovering hidden maliciousness within app code.

Existing detection methods with multi-view fusion techniques struggle to model flow-related information accurately, which in turn weakens the expressiveness of the fused representations. Many methods do not explicitly model the relations between the flow-based features across different views. Instead, they utilize only partial aspects of flow-related features indirectly, *e.g.*, sensitive API usage, function calls [15], [18]–[20]. Given the continuity and interdependence of flow characteristics within the views, the indirect modeling hinders the extraction of meaningful semantic associations. Furthermore, the methods that explicitly represent flow-related information

Zhaoyi Meng, Fenglei Xu, Wansen Wang, Jie Cui, Hong Zhong are with the School of Computer Science and Technology, Anhui University, Hefei, 230039, China. Wenxiang Zhao, Wenchao Huang, Yan Xiong are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China. Corresponding authors: Wansen Wang, Wenchao Huang. (e-mail: zymeng@ahu.edu.cn; 23762@ahu.edu.cn; huangwc@ustc.edu.cn)

TABLE I
STATISTICAL RESULTS OF TOP-10 KEY ELEMENTS FROM THE VIEWS OF CONTROL FLOWS, DATA FLOWS, AND ICCS ON OUR WHOLE DATASET

Triggering Conditions				**Guarded APIs (Simplified)**			
Benignware	**#**	**Malware**	**#**	**Benignware**	**#**	**Malware**	**#**
NO_CATEGORY	72828	NO_CATEGORY	149114	res.AssetManager.open	4035	res.AssetManager.open	46456
NETWORK_INFORMATION	6690	NETWORK_INFORMATION	43512	HashMap.put	3920	res.Resources.getAssets	46167
DATABASE_INFORMATION	2908	DATABASE_INFORMATION	824	String.startsWith	3474	FileOutputStream. write	27411
LOCATION_INFORMATION	819	UNIQUE_IDENTIFIER	620	res.Resources.getAssets	3419	File.getParentFile	18802
BLUETOOTH_INFORMATION	241	LOCATION_INFORMATION	301	FileOutputStream.write	3317	File.getAbsolutePath	13612
CALENDAR_INFORMATION	181	BLUETOOTH_INFORMATION	59	File.getPath	2620	File.getCanonicalPath	11543
UNIQUE_IDENTIFIER	169	CALENDAR_INFORMATION	33	Activity.getIntent	2256	URLConnection.setRequestProperty	1721
FILE_INFORMATION	90	ACCOUNT_INFORMATION	4	File.getAbsolutePath	2197	URLConnection.getInputStream	893
NFC	62	NFC	1	File.getParentFile	2062	HttpURLConnection.setRequestMethod	856
ACCOUNT_INFORMATION	2	FILE_INFORMATION	0	Class.getName	1772	Camera.setFlashMode	467
Source APIs (Simplified)				**Sink APIs (Simplified)**			
Benignware	**#**	**Malware**	**#**	**Benignware**	**#**	**Malware**	**#**
Class.getName	1140128	Class.getDeclaredMethod	640052	HashMap.put	2234284	HashMap.put	1663037
HashMap.get	554018	Class.getName	481456	String.substring	1950565	String.substring	710207
reflect.Field.get	534559	HashMap.get	235556	String.startsWith	1005406	String.startsWith	477782
Hashtable.get	297028	GregorianCalendar.get	212738	reflect.Field.set	550355	JSONObject.put	193678
Class.getSimpleName	285725	reflect.Field.get	201261	URL.openConnection	382903	reflect.Field.set	168789
ArrayList.get	275310	ArrayList.get	198304	Log.d	317969	URL.openConnection	130237
Array.newInstance	255319	System.getProperty	103814	Log.v	265275	Camera.setPreviewSize	82974
System.getProperty	250467	SQLiteDatabase.query	54504	StringBuffer.setLength	216059	FileOutputStream.write	65697
ThreadLocal.get	247968	reflect.Array.get	53393	JSONObject.put	168076	Log.w	65323
Class.getMethod	178051	File.getPath	51140	ThreadLocal.set	158014	StringBuffer.setLength	61138
Components				**Actions of Intents (Simplified)**			
Benignware	**#**	**Malware**	**#**	**Benignware**	**#**	**Malware**	**#**
Activity	1072155	Activity	2720802	VIEW	93533	VIEW	135251
Service	238136	Service	387539	MAIN	69091	MAIN	57902
Broadcast Receiver	172222	Broadcast Receiver	212052	MESSAGING_EVENT	22181	CONNECTIVITY_CHANGE	36872
Conent Provider	96457	Conent Provider	149763	BOOT_COMPLETED	20984	BOOT_COMPLETED	28689
/	/	/	/	RECEIVE	18204	USER_PRESENT	23371
/	/	/	/	CHOOSER	17334	PACKAGE_REMOVED	21449
/	/	/	/	INSTANCE_ID_EVENT	13911	PACKAGE_ADDED	13972
/	/	/	/	CONNECTIVITY_CHANGE	12664	com.taobao.accs.intent.action.RECEIVE	8856
/	/	/	/	REGISTER	11680	ACTION_POWER_CONNECTED	8788
/	/	/	/	INSTALL_REFERRER	11300	ACTION_POWER_DISCONNECTED	8738

as pre-defined graph structures often overlook the heterogeneity inherent in various flows [21], [22], which potentially leads to the loss of critical behavioral semantics. For instance, source APIs, sink APIs, and ICC links contribute differently to the semantics of data flows and should therefore be weighted accordingly in maliciousness analysis.

Directly fusing the semantics of different types of flows using existing heterogeneous graph-based methods [23]–[27] is technically non-trivial. Specifically, the semantics of a program flow depend on its context, *e.g.*, the partial order among its constituent entities. However, the methods above are typically context-unaware, making it difficult to distinguish entities of the same type that appear in different relations, which will be illustrated in Section III-D1. As a result, they fail to capture the mutual constraints and dependencies across the flows, and consequently allow unrelated flows from different apps to be mistakenly combined, leading to semantic confusion and compromising the reliability of the learned representations.

We propose and implement *MalFlows*, a novel technique that achieves context-aware fusion of heterogeneous flow semantics for Android malware detection. Our primary goal is to leverage complementary strengths of the three types of flow-related semantic information above for profiling app precisely. Specifically, we explicitly model the relations among entities from various flows using a heterogeneous information network (HIN) [28]. To incorporate high-level semantics for establishing inter-app relatedness, we construct a meta-path group for each view. Each group consists of content-oriented and action-oriented meta-paths designed to characterize components, structures, or usage patterns of flows. To address the computational and storage costs of mining HIN and integrate context awareness, we propose a new HIN embedding approach named *flow2vec*, which learns low-dimensional representations for HIN nodes while accurately preserving both the structural and semantic properties of flows. Unlike existing techniques [29]–[31], our approach effectively distinguishes the semantics of HIN entities based on contextual constraints across different flows as needed and jointly samples the HIN along multiple meta-paths to enhance app representation learning. We finally develop a channel-attention-based model to fuse semantic embeddings from the views, weighting their contributions for precise detection.

Our main contributions are summarized as follows:

- We propose and implement MalFlows, a novel technique that achieves context-aware fusion of heterogeneous flow semantics for Android malware detection. To the best of our knowledge, it is the first study to comprehensively aggregate the strengths of diverse program-flow-related information for assessing maliciousness of apps.
- We design a new context-aware HIN embedding approach named *flow2vec*, which distinguishes the semantics of HIN entities based on contextual constraints across different flows and learns accurate app representations through the joint use of multiple meta-paths.
- Our comprehensive evaluations on over 31,000 real-world apps and more than 20 millions program flow instances demonstrate that MalFlows outperforms representative baselines and validate the effectiveness of *flow2vec* in representing flow-related semantics.

The rest of the paper is organized as follows: Section II

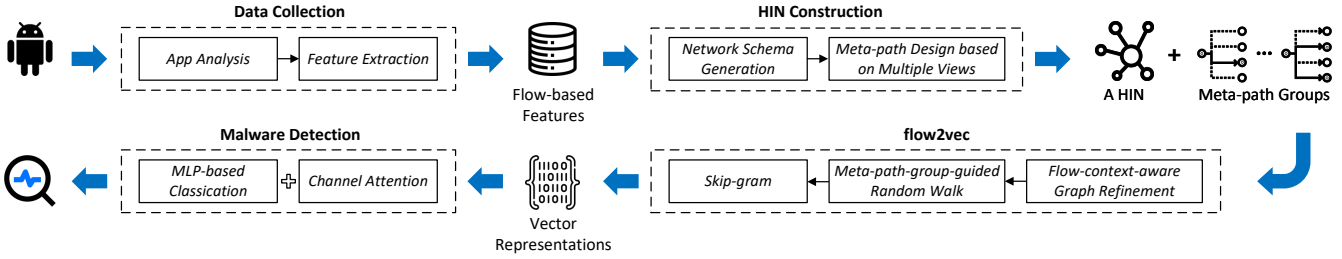

Fig. 1. Overall architecture of MalFlows.

introduces our motivation. Section III explains MalFlows's architecture and the detailed methodology of MalFlows. Section IV presents experimental results on MalFlows. Section V discusses the limitations and future work. Section VI shows the related work, and we conclude in Section VII.

II. Motivation

To motivate our work, we conduct detailed statistical analysis of heterogeneous flows, including control flows, data flows, and ICCs on large-scale apps, and then explain why the three types of flows contribute to effective malware detection. Specifically, we collect 31,301 real-world samples, including 16,667 benignware samples and 14,634 malware samples, spanning multiple years from AndroZoo [32]. We then use publicly available tools [4], [5], [33], [34] to extract triggering conditions, the guarded APIs, source APIs, sink APIs, components, and actions of Intents corresponding to the three types of flows respectively. The details of our dataset are depicted in Section IV-A.

Table I lists top-10 key elements extracted from 2018-2022 samples, where common prefixes (*e.g.*, *java.lang*, *android.content*) are removed for brevity. Based on that, we make the following observations:

(a) There exist clear distinctions between benignware and malware in triggering conditions and guarded APIs. Malware show higher frequencies of control logic associated with sensitive contexts, *e.g.*, *NETWORK_INFORMATION* and *UNIQUE_IDENTIFIER*, compared to benignware. Moreover, APIs about file and network operations, *e.g.*, *FileOutputStream.write* and *URLConnection.setRequestProperty*, are more frequently guarded in malware. In contrast, benignware tend to guard APIs of general functionalities (*e.g.*, *HashMap.put*, *Activity.getIntent*). The observations indicate that the combination of triggering conditions and guarded APIs provides discriminative features for malware detection.

(b) The differences emerge in the usage of source APIs and sink APIs between benignware and malware. While benignware predominantly access standard reflection and collection sources, *e.g.*, *Class.getName* and *HashMap.get*, malware exhibit increased reliance on sensitive sources like *GregorianCalendar.get*, and *SQLiteDatabase.query*. In the aspect of sink APIs, benign usage is dominated by common data manipulation APIs, *e.g.*, *HashMap.put* and *String.substring*, whereas malware frequently invoke sensitive sinks, *e.g.*, *JSONObject.put*, *FileOutputStream.write*, and *Camera.setPreviewSize*. This suggests that sources and sinks provide meaningful features for malware detection.

(c) The analysis of ICCs reveals divergence in how benignware and malware utilize components and actions of Intents. Malware invoke more components, particularly Activity and Service, which suggest a higher reliance on ICCs to coordinate background or stealthy behaviors. Moreover, malware frequently register for system-level actions, *e.g.*, *CONNECTIVITY_CHANGE*, and *PACKAGE_REMOVED*, which are less used by benignware. The patterns indicate that the ICC-based features provide discriminative clues for malware detection.

(d) The three types of flows exhibit clear semantic relations and complementarity. On the one hand, there exists an intersection between guarded APIs and source/sink APIs. For example, *FileOutputStream.write* appears in both guarded APIs and sink APIs of malware. *Class.getName* occurs in both guarded APIs and source APIs of benignware. On the other hand, the components of different types of flows are also explanatorily related. For instance, *CONNECTIVITY_CHANGE* in actions of Intents of malware can be seen as an explanatory indicator for the use of a sink API named *URL.openConnection*. This demonstrates that the semantics of these heterogeneous flows exhibit the potential for integration, which is beneficial for malware behavior analysis.

III. Methodology

A. Architecture

Figure 1 depicts the overall architecture of MalFlows, consisting of four main components as follows:

Data Collection. The component obtains flow-based features by off-the-shelf static analysis tools. Specifically, MalFlows employs IccTA [5] to collect explicit and implicit ICC links. Additionally, it uses FlowDroid [4] to gather intra- and inter-component sensitive data-flow paths. Furthermore, it gets conditional statements of the guarded sensitive APIs by the Soot framework [33], [34]. Based on the raw data above, MalFlows automatically extracts meaningful features (*e.g.*, trigger conditions, sensitive APIs), and then analyzes various relations (*e.g.*, *condition-trigger-API*) among different types of entities (*e.g.*, condition, API).

HIN Construction. The component constructs a HIN using the flow-based features extracted by the previous component. Specifically, a network schema is generated to model the relations among various entities of heterogeneous flows. Based on the schema, the HIN is constructed. After that, three groups of meta-paths are designed from the HIN to capture

the relatedness over apps from different views of flows. To simplify our design, each group of meta-paths include one content-oriented meta-path and one action-oriented meta-path.

flow2vec. The component utilizes a context-aware HIN embedding approach to generate low-dimensional representations of nodes in the HIN, while accurately preserving both semantics and structural correlations between different types of nodes. To ensure the semantic correctness of flow-related embeddings on the HIN, a flow-context-aware graph refinement method is proposed to separate nodes based on their associated contexts. Next, a random walk strategy guided by predefined meta-path groups is designed to map the word-context concept in a text corpus into the HIN. This strategy guarantees the semantic completeness of flow-related information during the sampling process on the HIN. Skip-gram is finally used to learn node representations for the HIN.

Malware Detection. The component trains a channel-attention-based deep neural network (DNN) model using the vectors learned by *flow2vec* to determine whether a given app is malicious. Specifically, a channel attention mechanism is used to dynamically adjust the importance of the vectors from different views, thereby enhancing the effectiveness of their fusion. Finally, a multi-layer perceptron(MLP)-based classifier is trained to produce a predicted label for the input app.

B. Data Collection

Based on the domain knowledge, we leverage existing static analysis tools to extract features from three views centered on control flows, data flows, ICCs. The flow-based features contain various entities of apps (*e.g.*, APIs, conditions, actions) and rich semantic relations among them, all of which are crucial to build the HIN in the following.

1) Feature Extraction from Control-flow View: Using control structures for sensitive operations is one of the common tricks by which malware can be camouflaged as benignware [11], [35]. To capture this kind of malicious intentions, we obtain trigger conditions, as well as the APIs guarded by the conditions within apps as follows.

We leverage the Soot framework to obtain the conditions that control the executions of sensitive APIs. Specifically, app code is first transformed into Jimple, the internal representation of Soot [34]. Sensitive APIs are then positioned in the Jimple code based on their API signatures. We extract the trigger conditions and their triggering semantics based on AppScalpel's strategy [33] implemented by Soot APIs as follows.

We position the target conditional statement based on a rule that the target statement is the predominator of an invocation statement of the guarded sensitive API, but the invocation statement is not its postdominator, and meanwhile, the join point of the conditional statement's branches is the postdominator of the invocation statement. We next extract the semantics of the obtained conditions by code instrumentation. Specifically, the functionality of a source API that has information flow to a conditional statement can be regarded as the semantic of that statement. However, conditional statements cannot be considered as sinks when using state-of-the-art static taint analyzers [4], [36]. To address this limitation, we instrument each conditional statement with a dummy method *ifStmt()*, which takes the variables involved in the conditional statement as parameters. The *ifStmt()* is static and declared in a dummy class *IfClass* that contains all the instrumented methods related to conditional statements. As depicted in Listing 1 for an app[1], we add code at Line 8 to wrap the parameters of the conditional statement at Line 9.

```
public final void b() {
    LocationProvider localLocationProvider;
    do {
        localLocationProvider = localLocationManager.
            getProvider((String)((Iterator)localObject2).
            next());
    } while(localLocationProvider.getAccuracy() != 2);
    ...
    // A dummy method call for a conditional statement
+   IfClass.ifStmt(localLocationProvider, localObject2);
    if((localLocationProvider == null) || (
         localLocationManager.getProvider((String)
         localObject2).getAccuracy() != 1)) {
        a(localLocationManager.getLastKnownLocation(...);
    }
}
```

Listing 1. A simplified code snippet of a real-world malware sample named *com.Gamezoor.Grand.CityRacing.game* for the instrumentation performed by MalFlows (a Line with "+" represents the instrumented code).

When the instrumentation is complete, the newly generated method calls are dynamically registered as sinks in FlowDroid. Based on that, we can get the data flows from source APIs and *ifStmt()*. We finally leverage the categories proposed by SuSi [37] to summarize the semantics of the condition-related sources. For example, the semantics of *getDeviceId()* and *getSubscriberId()* are summarized as *UNIQUE_IDENTIFIER*.

We build two matrixes to describe the relations above:

- $\mathbf{R_1}$**:** The *app-include-condition* matrix $\mathbf{C}$ where each element $c_{i,j} \in \{0,1\}$ means if app$_i$ includes condition$_j$.
- $\mathbf{R_2}$**:** The *condition-trigger-API* matrix $\mathbf{T}$ where each element $t_{i,j} \in \{0,1\}$ means if condition$_i$ controls the invocation of sensitive API$_j$.

2) Feature Extraction from Data-flow View: Usage of sensitive data within apps is one of the most important clues to identify hidden maliciousness [4], [6]. We run FlowDroid [4] to collect intra- and inter-component data-flow paths for each app. We then get a pair of APIs (*i.e.*, a source API and a sink API) from each path and record the apps using the APIs.

We build two matrixes to represent the obtained relations:

- $\mathbf{R_3}$**:** The *app-use-API* matrix $\mathbf{U}$ where each element $u_{i,j} \in \{0,1\}$ means if app$_i$ uses API$_j$.
- $\mathbf{R_4}$**:** The *API-flow-API* matrix $\mathbf{F}$ where each element $f_{i,j} \in \{0,1\}$ means if there exists a data flow from API$_i$ to API$_j$. API$_i$ is a source API and API$_j$ is a sink API.

3) Feature Extraction from ICC View: The ICC mechanism can be exploited by malware to launch stealthy attacks, *e.g.*, data leaks, code obfuscation, privilege escalation [5], [9]. To enhance the detectability of MalFlows for ICC-based malicious behaviors, we first run IccTA [5] to get ICC links from apps and extract actions of Intents and source components from each ICC link. The action is a core and widely used attribute of an implicit Intent, informing what operation is expected to be executed. The source component refers to the

[1]MD5: dc9b09466a0024f22989f71d9632a0d6

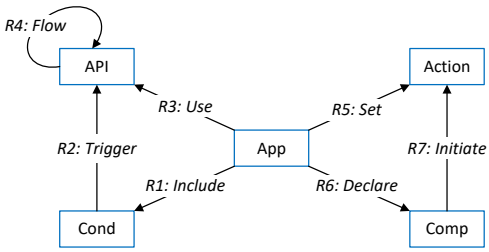

Fig. 2. Network schema for the HIN in MalFlows, where blue solid rectangles represent entity types, and black solid arrows indicate their relations.

sender of an explicit or impilict Intent. The two types of entities are essential to interpret the semantics of ICC links.

We build two matrixes to represent the obtained relations:

- $\mathbf{R}_5$**:** The *app-set-action* matrix **S** where each element $s_{i,j} \in \{0, 1\}$ means if action$_j$ is set in app$_i$.
- $\mathbf{R}_6$**:** The *app-declare-component* matrix **D** where each element $d_{i,j} \in \{0, 1\}$ means if app$_i$ declares component$_j$ in AndroidManifest.xml.
- $\mathbf{R}_7$**:** The *component-initiate-action* matrix **N** where each element $n_{i,j} \in \{0, 1\}$ means if component$_i$ initiates action$_j$ in an Intent.

C. HIN Construction

To depict the rich relations among the extracted Android entities, it is crucial to model them in an appropriate manner so that different relations can be better and easier analyzed and handled. Therefore, we employ a HIN [28] that is capable to incorporate different types of features (*i.e.*, entities and relations) extracted above. The HIN facilitates uncovering underlying intentions within Android apps by providing not only the network structure of the data associations but a high-level abstraction of categorical association. We first exhibit the definitions of the HIN and its network schema as follows.

Definition 1. *A heterogeneous information network [28] is defined as a graph G = (V, E) with an entity type mapping ϕ: V $\rightarrow$ T and a relation type mapping ψ: E $\rightarrow$ R, where V denotes the entity set and T denotes the entity type set, and E denotes the relation set and R denotes the relation type set, and the number of entity types $|T| > 1$ or the number of relation types $|R| > 1$. The network schema for G, denoted as S_G = (T, R), is a graph with nodes as entity types from T and edges as relation types from R.*

1) Network Schema Generation: We have 5 entity types (*e.g.*, APIs, conditions) and 7 types of relations among them (*i.e.*, R_1-R_7 in Section III-B). The network schema for the HIN of our work is shown in Figure 2, which enables apps to be comprehensively represented by simultaneously incorporating information from multiple views.

2) Meta-path Design: The different types of entities and relations motivate us to use a machine-readable representation for enriching the semantics of similarities among apps. The similarities are our basis to measure the maliciousness of apps. To handle this, the meta-path is designed to formulate the

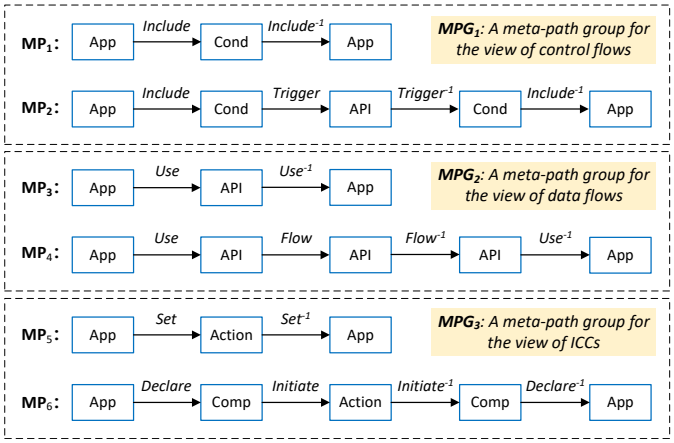

Fig. 3. Meta-paths built for flow-centric malware detection, where some entities (*e.g.*, *Cond*) is short for the contents in Figure 2 (*e.g.*, *Condition*). The definitions of nodes and edges are the same as those in Figure 2.

semantics of higher-order relations among entities in HIN. Its definition is shown as follows.

Definition 2. *A meta-path MP is a path defined on the graph of network schema S_G = (T, R), and is denoted in the form of $T_1 \xrightarrow{R_1} T_2 \xrightarrow{R_2} T_3 \xrightarrow{R_3} \ldots \xrightarrow{R_M} T_{M+1}$, which defines a composite relation $R = R_1 \circ R_2 \circ R_3 \circ \ldots \circ R_M$ between types T_1 and T_{M+1}, where $\circ$ denotes relation composition operator, and M is the length of MP.*

Given the network schema with different types of entities and relations, we enumerate many meta-paths. Based on the domain knowledge from human experts, we design 6 meaningful meta-paths shown in Figure 3 for characterizing relatedness over apps. To aggregate different types of semantic information from our HIN, we then group the meta-paths based into 3 distinct but related views, *i.e.*, control flows, data flows, and ICCs. Each group comprises one content-oriented meta-path (*e.g.*, MP_1) and one action-oriented meta-path (*e.g.*, MP_2), which work together to capture various relations within a view. Note that additional meta-paths can be included in each group, but we select the two most representative ones to demonstrate the effectiveness of MalFlows.

For example, the relatedness over apps by leveraging the control-flow features in a meta-path MP_1 is *App* $\xrightarrow{\text{Include}}$ *Condition* $\xrightarrow{\text{Include}^{-1}}$ *App*, which means that two apps can be connected as they use the trigger conditions with the identical semantics. Another meta-path MP_2 in the same group, *i.e.*, *App* $\xrightarrow{\text{Include}}$ *Condition* $\xrightarrow{\text{Trigger}}$ *API* $\xrightarrow{\text{Trigger}^{-1}}$ *Condition* $\xrightarrow{\text{Include}^{-1}}$ *App*, denotes that two apps are related as they use the trigger conditions controlling the invocations of the same API. The former depicts the contents of apps, but the latter describes the detailed actions of apps.

D. flow2vec

To measure the relatedness over entities of the constructed HIN, it is essential to adopt an effective representation learning method that effectively aggregates both structural and semantic relations about various flows.

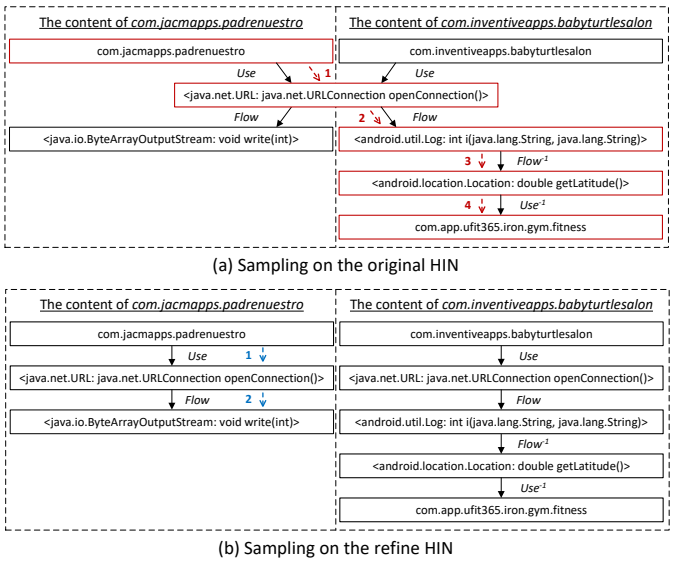

Fig. 4. Example for sampling on the original and refined HIN, where solid rectangles represent entities, black solid arrows indicate their relations, and the numbers on the dashed arrows show the order of access. Incorrect sampling paths are highlighted in red, and the correct paths are highlighted in blue.

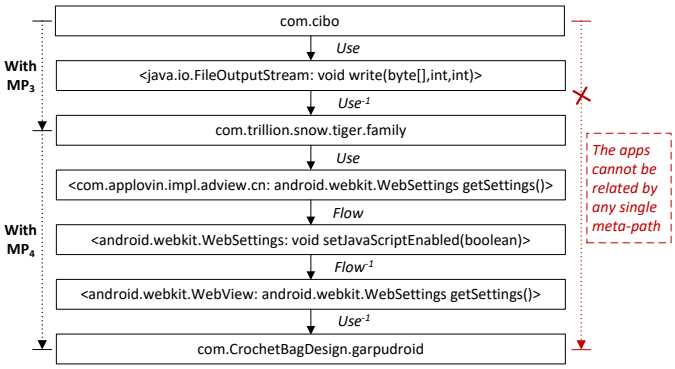

Fig. 5. Example of the HIN sampling guided by a meta-path group, where the black solid arrows represent the relations between the entities.

1) Flow-context-aware Graph Refinement: Modeling flow-related information using a standard HIN method [23] directly results in the description of incorrect relations. Specifically, flow-related information in the HIN consists of multiple entities, associated not only through the explicitly modeled relations but also by implicit contextual constraints (*e.g.*, the source API and the sink API in a data-flow path are used in the same app). Overlooking the constraints lead to incorrect extraction of structural and semantic relations of the HIN.

We use a real-world case to demonstrate the importance of modeling the flow-related information under contextual constraints. As drawn in Figure 4(a), the red sampling paths on the subgraph of the HIN violate a contextual constraint. Specifically, the app named *com.jacmapps.padrenuestro*[2] and the app named *com.app.ufit365.iron.gym.fitness*[3] appear to be related by MP_3 under the data-flow view. However, the sink API named ⟨*android.util.Log: int i(java.lang.String, java.lang.String)*⟩ is not used in the former app. Therefore, the data-flow-based relation does not exits in practice. The former app and the sink API are incorrectly associated over the HIN, resulting in inaccurate representations for the two apps. In comparison, as shown in Figure 4(b), by representing the API named ⟨*java.net.URL: java.net.URLConnection openConnection()*⟩ as two separate instances corresponding to different apps, the graph refinement enables the HIN to capture context-specific semantics and prevents unrelated apps from being incorrectly linked. All the paths (*e.g.*, the blue one) sampled from the refined HIN exist in practice.

Algorithm 1 Implementation of the graph refinement

1: **Input:** A HIN H, a meta-path mp_l of length 4, a set S_mp_s consisting of meta-paths of length 2
2: **Output:** A refined HIN H
3: $N \leftarrow$ getAnchorNodes(H, mp_l)
4: **for all** n in N **do**
5: $P \leftarrow$ getPredecessor(n, H, mp_l);
6: $S \leftarrow$ getSuccessor(n, H, mp_l)
7: **for all** p in P and s in S **do**
8: $nc \leftarrow$ clone(n)
9: **if** isConnected(p, s, S_mp_s) **then**
10: setPredecessor(nc, p, H); setSuccessor(nc, s, H)
11: **else if** haveNotConnected(p, H) **then**
12: setPredecessor(nc, p, H)
13: **end if**
14: **end for**
15: removeOriginalNode(n, H)
16: **end for**
17: **return** H

As depicted in Algorithm 1, we leverage joint semantics of meta-paths in each group to identify and correct the inaccurate parts in a HIN on demand. We first obtain all nodes (*i.e.*, anchor nodes) with the second entity type along mp_l from a HIN H (Line 3). We then find predecessors P and successors S of each anchor node (Lines 5-6). Next, we clone each anchor node as nc and rebuild its connection relations as follows. We connect a predecessor p to nc and nc to a successor s when p and s can be connected via any meta-path in S_mp_s (Lines 10-12). Otherwise, we connect p to nc only once (Lines 13-14). In other word, we use content-oriented relations of the meta-paths in S_mp_s to constrain action-oriented relations of mp_l in the HIN construction. Afterwards, we remove the anchor node to avoid information redundancy in H (Line 18). Finally, a refined HIN is generated.

2) Meta-path-group-guided Random Walk: To learn latent representations for the refined HIN, it is inapplicable to directly apply traditional single meta-path-based sampling schemes, *e.g.*, metapath2vec [30], HAN [31], MAGNN [38]. Specifically, single meta-path-based sampling considers only a limited subset of flow-based relations in the HIN, so it may fail to capture important structural and semantic dependencies. As depicted in Figure 5, malware samples named *com.cibo*[4]

[2]MD5: 0bffa8e7b29306f8bbec28dba6bb7150
[3]MD5: cbe89039caf326cff16070d50c0c0da8
[4]MD5: 5aa13c2dfe6668a6a170e82c70c1a56c

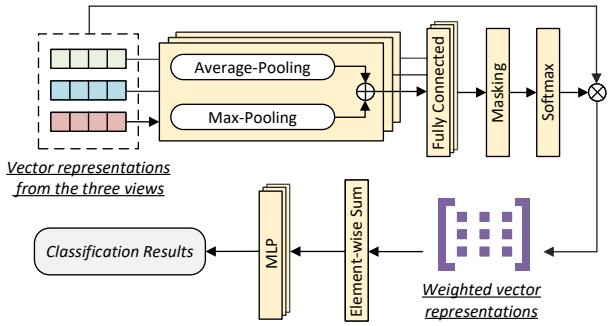

Fig. 6. Channel-attention-based deep neural network classifier.

and *com.CrochetBagDesign.garpudroid*[5] cannot be related by any single meta-path in Section III-C2. Due to the failure to capture their relations, the two apps may be misclassified in malware detection, In comparison, the apps can be connected by the meta-paths in MPG_2, where the app named *com.trillion.snow.tiger.family*[6] bridges them in the HIN. By capturing the relations, the apps can be correctly classified.

To produce precise node embeddings for our work, we extend metapath2vec based on the proposed work [39] for sampling based on each meta-path group. This facilitates the extraction of structural and semantic information from both contents and actions cooperatively. Given a meta-path group *MPG*, we put a random walker to traverse a HIN. The walker randomly chooses a meta-path MP_k from *MPG* at first and the transition probabilities at step i are defined as follows:

$$p\left(v^{i+1} \mid v_m^i, \mathit{MPG}\right) = \begin{cases} \frac{\mu}{|\mathit{MPG}|}\frac{1}{|N_{m+1}(v_m^i)|} & \text{if } \left(v_m^i, v^{i+1}\right) \in E, \phi\left(v_m^i\right) = T_{app}, \\ & \quad \phi\left(v^{i+1}\right) = T_{t+1} \\ \frac{1}{|N_{m+1}(v_m^i)|} & \text{if } \left(v_m^i, v^{i+1}\right) \in E, \phi\left(v_m^i\right) \neq T_{app}, \\ & \quad \phi\left(v^{i+1}\right) = T_{t+1}, (T_m, T_{m+1}) \in \text{MP}_k \\ 0 & \text{otherwise,} \end{cases}$$

where ϕ is the node type mapping function, v_m^i denotes the T_m type of the entity visited at step i, $N_{m+1}\left(v_m^i\right)$ denotes the T_{m+1} type of neighborhood of the entity v_m^i, T_{app} is the entity type of app, and μ is the number of meta-paths starting with $T_{app} \rightarrow T_{m+1}$ in the given meta-path group G. Note that MP_k will be updated by the meta-path drawn from G with $\frac{\mu}{|S|}$ probability when the walker meets the case at the first line of the above formula.

When the sampling length is long enough, the walker can traverse the HIN with various combinations of meta-paths within the given meta-path group. For example, if *MPG* = $\{\text{MP}_1, \text{MP}_2\}$, the walker may traverse based on the combined meta-paths, *e.g.*, [MP_1, MP_1, ...], [MP_2, MP_2, ...], [MP_1, MP_2, MP_1, MP_2, ...]. The sampled paths accurately preserve both structural and semantic relations between different types of nodes in the HIN, and thus facilitate the transformation of HIN structures into sequences. We finally use skip-gram [40] to generate vectors for entity nodes in the sampled paths.

[5]MD5: 37218220d384bf08aece74ab38479ce4
[6]MD5: 62925eabca26b8be2f6f7a2f280900b0

TABLE II
PARAMETERS OF OUR MLP-BASED ANDROID MALWARE CLASSIFIER

Designed Layer	Dimension	Activation Function	Using Dropout
Input Layer	128	None	No
Hidden Layer1	256	ReLU	Yes
Hidden Layer2	128	ReLU	No
Hidden Layer3	64	ReLU	Yes
Hidden Layer4	32	ReLU	No
Hidden Layer5	16	ReLU	Yes
Output Layer	1	Sigmoid	No

E. Channel-attention-based DNN Classifier

To leverage strengths of the semantics of the heterogeneous flows for profiling app behaviors, we design a channel-attention-based DNN classifier. As shown in Figure 6, for an app, we treat each feature vector extracted from a view of flows as a distinct channel and then employ channel attention to adaptively reweight each feature vector for the fusion of the semantics. This treatment enhances the most informative representations while suppressing less relevant ones, making a more comprehensive and discriminative fused representation for precise malware classification.

1) Channel-attention-based Semantic Fusion for Heterogeneous Flows: We first aggregate semantic information of the feature vectors $V \in \mathbb{R}^{c \times d}$ from the three views by both average-pooling and max-pooling operations, where c is the number of channels, and d is the dimension of each input feature. In our work, c = 3 and d = 128. Average-pooling preserves overall information and avoid feature loss, and then max-pooling captures the most dominant features. These operations generate two intermediate vectors: $V_{avg} \in \mathbb{R}^{c \times 1}$ and $V_{max} \in \mathbb{R}^{c \times 1}$. Both of the vectors are then added element-wise and forwarded to a network to produce our channel attention map. The network is composed of two fully-connected layers with the ReLU activation function.

$$M_c = \text{Softmax}(\theta(W_1^T(\text{ReLU}(W_0^T(V_{avg} + V_{max}))))) \quad (1)$$

, where $W_0 \in \mathbb{R}^{c \times 6}$ and $W_1 \in \mathbb{R}^{6 \times c}$ are project matrices. θ denotes a masking mechanism, which is used to identify and suppress zero channels, ensuring that they do not affect the subsequent Softmax computation, thereby improving numerical stability. $M_c \in \mathbb{R}^{c \times 1}$ is the channel attention map.

After that, we make element-wise multiplication of M_c and V to produce attention-weighted feature vectors. We next merge the vectors using element-wise summation:

$$y^{(0)} = \mathbf{1}^\top (M_c \cdot V) \quad (2)$$

2) Malware Classifier: The fused feature $y^{(0)} \in \mathbb{R}^{1 \times d}$ is fed into a 6-layer multi-layer perceptron (MLP) to get malware detection results. The parameters used in the model are listed in Table II. During the learning process, our model is tuned to minimize the value of the loss function, *i.e.*, the cross-entropy function. Moreover, to avoid the overfitting problem, Dropout regularization is applied for skipping some units randomly while the model is trained.

TABLE III
NUMBER OF APPS IN DIFFERENT CATEGORIES OF OUR DATASET

Category	2017	2018	2019	2020	2021	2022	2023	2024
Benignware	1132	3126	2631	2929	2314	3125	1204	206
Malware	1022	1994	2448	2096	3195	1808	1893	178

IV. EXPERIMENTAL EVALUATION

To evaluate the effectiveness of MalFlows, we seek to answer the following three questions:

- **RQ1:** Does MalFlows detect Android malware better than representative tools? Whether *flow2vec* facilitates precise malware detection compared to other typical techniques? How does MalFlows perform in terms of app evolution? What's the time cost of MalFlows?
- **RQ2:** Is it necessary to perform the flow-context-aware graph refinement? How effective is *flow2vec* in representing app behaviors with different usage of meta-paths?
- **RQ3:** How effective is the model for fusing the semantics of heterogeneous flows used in MalFlows for Android malware detection? What is the stability of the model?

A. Experimental Setup

1) Implementation: We implement a prototype of MalFlows. We use the off-the-shelf tools [4], [5], [33] to get flow-related analysis results from apps. We set the timeout for each tool in analyzing an app as 20 minutes. We leverage Deep Graph Library to build and refine the HIN, and achieve the meta-path-group-guided random walk. We finally implement the channel-attention-based DNN classifier by PyTorch. Our MLP-based classifier is modeled with the dropout rate of 0.5 and the learning rate of 0.001, both of which are commonly-used in typical DNN-based models. The experiments are conducted on a machine with AMD Ryzen 5 7600X 6-Core Processor 4.70 GHz CPU, 128GB memory, Ubuntu 22.04 LTS (64bit) and NVIDIA GeForce RTX 3080 Ti (12GB) GPU.

2) Dataset: To ensure authenticity and reliability of our statistics and the experimental results, as shown in Table III, we randomly collect 31,301 real-world samples spanning multiple years from AndroZoo [32]. The average size of each sample is 19.75 MB, where a maximum size of 333.04 MB and a minimum size of 4 KB. A sample is regarded as malicious if it is flagged by more than 2 antivirus engines, and a sample is regarded as benign if no engine reports it.

To extract various flow-based features from the apps above, we rent 12 cloud servers and run our self-developed extractor for over 3 months. Note that all the samples are analyzed by the aforementioned analysis tools without errors and interruptions. From malware samples, we obtain a total of 3,926,619 sensitive data-flow paths, 218,021 suspicious condition structures, and 157,153 ICC links; from benignware samples, we get a total of 13,905,513 sensitive data-flow paths, 108,472 suspicious condition structures, and 257,766 ICC links. All extractions are used to build the HIN.

3) Baselines: To demonstrate the effectiveness of our work, we select four representative and related Android malware detection tools with different key techniques:

TABLE IV
COMPARISON OF MALFLOWS AND OTHER SELECTED BASELINES ON OUR WHOLE DATASET, WHERE HINDROID-N2V = HINDROID WITH NODE2VEC, HINDROID-M2V = HINDROID WITH METAPATH2VEC, HINDROID-REC = HINDROID WITH API REDUCTION

Technique	Accuracy	Precision	Recall	F_1-score
Detection Tools				
HinDroid-n2v	93.42%	95.85%	90.98%	0.9336
HinDroid-m2v	87.33%	88.95%	85.74%	0.8731
HinDroid-rec	95.33%	95.26%	95.57%	0.9542
MaMaDroid	91.14%	94.96%	91.85%	0.9337
Drebin	93.97%	94.30%	92.40%	0.9333
AppPoet	95.81%	91.28%	94.99%	0.9310
Graph Embedding Models				
DeepWalk	83.91%	88.80%	75.04%	0.8134
LINE	77.85%	80.37%	73.33%	0.7669
node2vec	90.78%	91.17%	88.54%	0.8983
metapath2vec	93.81%	99.76%	91.33%	0.9536
HAN	94.51%	97.22%	92.01%	0.9455
MalFlows	**98.34%**	**98.98%**	**98.64%**	**0.9881**

- HinDroid [23] models apps, APIs, and their relations as a HIN, and computes app similarities via multi-kernel learning over semantic meta-paths. Based on the open-source project [41], we run HinDroid with three improvements: API reduction, node2vec [29], and metapath2vec.
- AppPoet [17] is a LLM-assisted Android malware detector. It firstly extracts string-type features from multiple views. Next, it generates comprehensive cross-view descriptions through multi-view prompt engineering. Finally, these descriptions are embedded and used to train a DNN model for malware detection.
- Drebin [13] is a typical framework that detects an app by collecting a wide range of features, *e.g.*, used hardware, API calls and permissions from *AndroidManifest.xml* and *.dex* code, and then trains a SVM-based classifier.
- MaMaDroid [42] abstracts each API on call graph to build a first-order Markov chain, and then uses transition probabilities as features. Following prior findings [19], we adopt the package mode for better performance, and apply Random Forests for malware classification.

Note that we attempt to find other HIN-based tools [25]–[27], [43] but they are either not publicly available or their code fails to run properly. Afterwards, to evaluate the performance of *flow2vec*, we select 5 generic models commonly used in some well-known malware detection systems as follows:

- HAN [31] is a heterogeneous graph representation learning model that leverages predefined meta-paths and hierarchical attention mechanisms for node embedding.
- metapath2vec is a heterogeneous graph representation learning model that leverages predefined meta-paths and skip-gram-based random walks for node embedding.
- LINE [44] is a network representation learning model that preserves both first-order and second-order proximities for node embedding.
- node2vec is a network representation learning model that utilizes biased random walks and the skip-gram model for node embedding.
- DeepWalk [45] is a network representation learning

TABLE V
COMPARISON OF MALFLOWS AND OTHER TOOLS IN THE APP EVOLUTION, WHERE HINDROID CORRESPONDS TO HINDROID-REC IN TABLE IV AS IT OBTAINS THE BEST RESULTS OF THE THREE IMPROVEMENTS

Tool	App Evolution			
	AUT(a)	AUT(p)	AUT(r)	AUT(f_1)
HinDroid	0.8871	0.8860	0.8841	0.8849
MaMaDroid	0.8515	0.8363	0.8370	0.8364
Drebin	0.8793	0.8715	0.8492	0.8599
AppPoet	0.9153	0.9199	0.9187	0.9185
MalFlows	**0.9369**	**0.9275**	**0.9446**	**0.9358**

model that utilizes truncated random walks and the skip-gram model for node embedding.

4) Evaluation Metrics: We define true positives as correctly classified malware, false positives as misclassified benignware, true negatives as correctly classified benignware, and false negatives as misclassified malware. We evaluate with four metrics, including Precision, Recall, Accuracy and F_1-score.

B. RQ1: Detection on Real-World Samples

1) Overall Effectiveness: To validate the effectiveness of MalFlows in malware detection, we randomly select 80% of apps from our dataset to form the training set, and test on the rest of the apps. The process above is run for 5 times, each time using a different subset for testing, and we calculate the average for each metric as the results. Furthermore, We train our model with a fixed 100 epochs.

The experimental results are shown at Lines 2-8 and Line 15 of Table IV. From the table, we can see that MalFlows outperforms all other selected detection tools in Precision, Recall, Accuracy and F_1-score. Specifically, for the three improvements of HinDroid, HinDroid-rec outperforms HinDroid-n2v and HinDroid-m2v, which indicates that retaining HinDroid's representation method while optimizing API selection yields better results than replacing its representation method. Furthermore, the F_1-score of HinDroid-rec is better than that of MaMaDroid, Drebin, and AppPoet. This demonstrates the value of using the HIN to detect Android malware. Compared to HinDroid-rec, MalFlows provides a comprehensive assessment of apps by complementarily fusing the semantics from the views of data flows, control flows, and ICCs, rather than relying solely on the relations of API calls. Similarly, MaMaDroid focuses on the sequences of API calls, which only reflects a portion of the semantics extracted from the three views used in MalFlows. Moreover, the features used by AppPoet and Drebin are organized as sets of strings, whereas MalFlows extracts high-order features from the HIN that contains rich relations based on different types of flows, enabling the expression of more structural and semantic information.

2) Effectiveness of flow2vec: To verify the effectiveness of the HIN embedding in MalFlows, we compare *flow2vec* with the representative baseline models described above. Specifically, we replace the graph embedding module in MalFlows, while keeping the dataset used, as well as the training and testing processes, identical to those described in Section IV-B1.

The experimental results are listed at Lines 9-14 and Line 15 of Table IV, showing that the HIN embedding method of MalFlows outperforms the other models in all metrics. Specifically, metapath2vec and HAN perform better than DeepWalk, LINE, and node2vec. This implies that, compared to modelling Android app behaviors with homogeneous graphs, using heterogeneous graphs can describe app behaviors more precisely and enhance the distinguishability between malware and benignware. In MalFlows, the meta-path-group-guided random walk of *flow2vec* jointly samples the HIN across multiple meta-paths under specified views, which helps capture structural and semantic relations in the HIN. Therefore, the vector representations can profile app behaviors comprehensively and precisely, leading to the best detection results performance among all compared schemes. Note that HAN and metapath2vec are executed based on our refined HIN. Their performance would likely deteriorate if evaluated on the standard HIN.

3) Adaptation for App Evolution: Machine learning-based Android malware detectors face a problem of aging as the app evolution due to the update of Android version [46]. To construct the required experimental environment, we use the samples from 2018 as the training set and the samples from each year between 2019 and 2024 to form six separate testing sets. Note that the samples from 2017 are not included due to their insufficient size. The appearance timestamps of the apps in the training set are earlier than the appearance timestamps of the apps in the testing set, so we experiment to figure out whether a malware classifier can identify evolved Android malware. We use Area Under Time (AUT) to measure a classifier's robustness to time decay [47]:

$$AUT(f, N) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{f(k+1) + f(k)}{2} \tag{3}$$

, where f is our evaluation metric, N is the number of test slots, and $f(k)$ is the evaluation metric generated at time k. N is set as 12 months in our experiment and thereby omitted from AUT expressions in subsequent sections. An AUT metric that is closer to 1 means better performance over time.

The experimental results are listed in Table V, where MalFlows outperforms all other tools in AUT(p), AUT(r), AUT(a) and AUT(f_1). This superior effectiveness is attributed to the complementary nature of the three flow-related views used by MalFlows, which facilitates capturing intrinsic factors of apps. Furthermore, it is interesting that AppPoet demonstrates the best overall performance among all other baseline tools, which validates that the features embedded from the content supplemented by LLMs are more stable over time than only API-related features and discrete string-based features in our experiment. However, the content in string format cannot intuitively represent the structural and semantic relations inherent in flow-related information, causing AppPoet to overlook key invariants in app evolution. This limitation ultimately results in AppPoet performing less effectively than MalFlows.

We then plot the change trend of the evaluation metrics for each selected tool over time in Figure 7. Overall, the metrics of all the tools drop to varying degrees over time. Among them, MalFlows exhibits the slowest decline in the four AUT metrics, indicating its robustness against the aging process. In

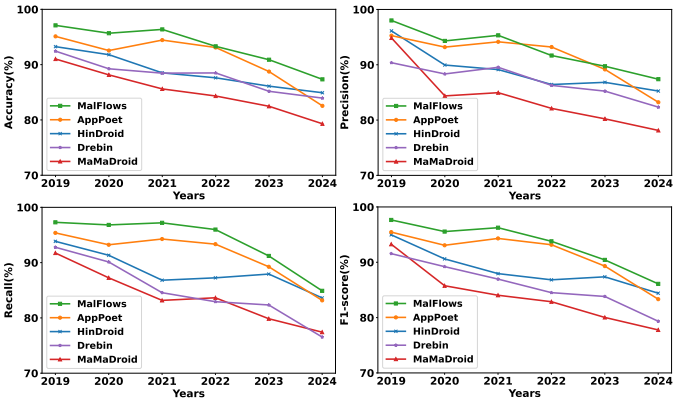

Fig. 7. Effectiveness metrics over 6 test slots of total 72 months.

contrast, the effectiveness of HinDroid, MaMaDroid, Drebin, and AppPoet deteriorates after 5 years, with all the metrics falling below 0.9. Therefore, we suggest that a detection model should ideally be retrained with newer app data after 4 years of use in practice. The suggestion is less strict than the one obtained by LensDroid [16]. Another interesting finding is that the metrics of AppPoet remain stable or even improve during 2020–2022, followed by a gradual decline. This phenomenon indicates that the feature descriptions generated by LLMs have the potential to resist the app evolution.

4) Time Cost: MalFlows consists of two stages. The HIN embedding stage is relatively time-consuming. The maximum runtime for a single view on our dataset is approximately 47 minutes. However, this stage is conducted offline and only once per dataset. Moreover, its efficiency can be improved through parallelization or enhanced hardware configurations. In contrast, the detection stage is highly efficient, making the system suitable for real-time analysis. MalFlows takes roughly 2.50 milliseconds on average to train a batch of 128, resulting in 1.04 seconds for an epoch over the whole dataset. The prediction time on average is 0.11 milliseconds.

5) Case Study: To exhibit the preformance of MalFlows, we present a malware sample named *com.runbey.byy*[7]. The app is detected by MalFlows correctly but misclassified by all the baselines above. To investigate the reason behind the app's classification, we extract the weights from the channel attention map during detection. The weights corresponding to the three channels are 0.4103, 0.2543, and 0.3354 respectively. This indicates that MalFlows adaptively emphasizes more informative views of the flows during the decision-making process, with particular attention given to the control-flow view, which contributes most significantly to the final prediction. Furthermore, we analyze the app from three views as follows: (a) In the view of control flows, the app frequently performs file operations when the network-related conditions are triggered. (b) In the view of data flows, the app repeatedly writes data to files during execution. (c) In the view of ICCs, the app makes use of various third-party components (*e.g.*, *cn.jpush*, *com.tencent*, *com.sina*) and custom actions of Intents (*e.g.*, *cn.jpush.android.intent.RECEIVE_MESSAGE*). Overall, the app is suspected to contain Trojan-like behavior and is capable of downloading unknown files from the network.

[7]MD5: 8ed90e74e8b41ecece4c35e86ebefff4

TABLE VI
DETECTION PERFORMANCE OF *flow2vec* WITH DIFFERENT SETTINGS

Setting	Accuracy	Precision	Recall	F_1-score
Usage of Graph Refinement				
Without	84.30%	96.54%	79.72%	87.33%
Usage of Meta-paths				
MP_1	91.46%	95.91%	91.63%	0.9372
MP_2	93.81%	99.76%	91.33%	0.9536
MP_3	91.58%	94.67%	89.43%	0.9197
MP_4	92.49%	94.23%	90.37%	0.9226
MP_5	88.58%	92.81%	78.09%	0.8481
MP_6	90.80%	93.40%	83.61%	0.8824
MPG_1	94.11%	99.31%	92.35%	0.9570
MPG_2	94.23%	95.72%	91.61%	0.9362
MPG_3	91.71%	92.51%	87.08%	0.8971
MPG_1,MPG_2	95.03%	96.11%	93.07%	0.9456
MPG_1,MPG_3	94.65%	97.57%	89.46%	0.9334
MPG_2,MPG_3	94.54%	94.74%	92.09%	0.9339
Ours	**98.34%**	**98.98%**	**98.64%**	**0.9881**

C. RQ2: Ablation Study of *flow2vec*

1) Necessity of Graph Refinement: We illustrate the necessity of the graph refinement based on the detection results. Except for not using the graph refinement, we perform the same experimental environments as described in Section IV-B1.

The results are listed at the third line of Table VI. We observe that the detection performance of MalFlows drops significantly across all metrics when the graph refinement is disabled, compared to when it is employed (*i.e.*, Ours). After further analysis, we infer that the variant of *flow2vec* without graph refinement tends to sample the paths that deviate from real-world scenarios from the built HIN. This leads to confusion in the relations between apps, which in turn negatively affects detection performance.

2) Usage of Different Meta-paths: To verify the effectiveness of our meta-path groups for *flow2vec*, we conduct ablation experiments using 13 different meta-path settings. Experimental environments are the same as depicted in Section IV-B2.

The experimental results are presented in Table VI, showing that using all groups of meta-paths (*i.e.*, Ours) yields the best performance, as the meta-paths under the three views of the flows effectively capture and represent app behaviors. We observe that: (a) MP_2 outperforms the other individual meta-paths, which indicates that trigger conditions can be effectively used to differentiate benignware and malware. (b) MP_2, MP_4, and MP_6 perform better than MP_1, MP_3, and MP_5 correspondingly. The reason is that important semantics within apps for detecting Android malware aggregated through action-oriented meta-paths are more beneficial than those aggregated through content-oriented meta-paths. (c) The overall effectiveness of meta-path groups with our semantic fusion method is superior to that of most individual meta-paths. Specifically, the Accuracy and F_1-score of MPG_1, MPG_2, and MPG_3 are higher than those of MP_2, MP_4, and, MP_6 respectively. This indicates that, under a single view, joint sampling based on both action-oriented and content-oriented

TABLE VII
COMPARISON OF THE MODEL WE USED AND THE ALTERNATIVE MODELS IN EFFECTIVENESS OF ANDROID MALWARE DETECTION

Fusion Method	Accuracy	Precision	Recall	F_1-score
Add-direct	92.00%	92.90%	89.31%	0.9105
Add-hybrid	95.26%	95.87%	96.07%	0.9597
Self-attn	87.22%	93.58%	74.57%	0.8300
Ours	**98.34%**	**98.98%**	**98.64%**	**0.9881**

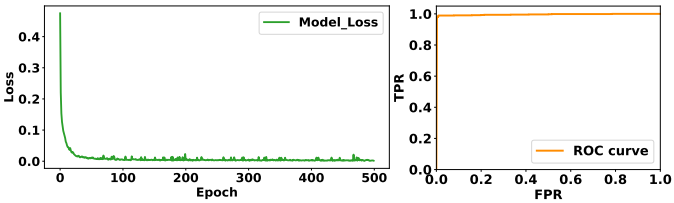

Fig. 8. Evaluation of model stability of MalFlows.

meta-paths is more effective in capturing key features of the HIN than sampling based on a single meta-path alone. (d) With the combined use of multiple groups of meta-paths, the detection performance of our tool is further improved, particularly in terms of Accuracy. For instance, with our semantic fusion method, the Accuracy of $\{MPG_1, MPG_2\}$ is 95.03%, which is higher than the Accuracy of MPG_1 (94.11%) and MPG_2 (94.23%). This further demonstrates the complementary effect among different groups of meta-paths.

D. RQ3: Performance of Our Fusion Model

1) Effectiveness on Fusing Heterogeneous Flow Semantics: To validate if the flow semantic fusion in MalFlows facilitates Android malware detection, we compare the effectiveness of MalFlows when using the original and alternative models. The experimental environments are the same as described in Section IV-B2. We construct three alternatives:

- It uses metapath2vec to generate vectors based on the 6 meta-paths respectively, and then leverages the element-wise vector addition to fuse the vectors (*i.e.*, Add-direct).
- It uses the meta-path-group-guided random walk to generate vectors for each view, and then leverages the element-wise vector addition to fuse the vectors (*i.e.*, Add-hybrid).
- It implements the multi-view fusion by replacing channel attention as self-attention (*i.e.*, Self-attn).

Table VII presents the experimental results, showing that the model we used surpasses the alternatives in the effectiveness of malware detection. Specifically, the methods of Add-direct and Add-hybrid perform better than the method of Self-attn. As further analysis, we speculate that the feature vectors obtained from the three views of flows contain unique semantic information, all of which make important contributions to Android malware detection. Vector addition does not alter the original feature representation, which makes it suitable for local feature fusion. However, self-attention may disrupt local semantics. Moreover, compared to Add-direct, Add-hybrid performs better, mainly because the meta-path-group-guided random walk can capture potential semantics within the HIN more effectively. Channel attention can adaptively enhance the weights of important views while suppressing irrelevant ones, thereby improving feature representation and achieving precision Android malware detection.

2) Model Stability: To demonstrates the training stability of our model, we plot Figure 8, where the experimental configurations and environments are described in Section IV-B1. Specifically, the left of Figure 8 plots the training losses of MalFlows. From the figure, we can find that the training loss decreases sharply during the initial epochs and gradually stabilizes after around epoch 50, indicating that the model converges effectively. The final loss remains consistently low, demonstrating that the model has successfully learned the underlying patterns of Android malware behaviors. The right of Figure 8 shows the ROC (receiver operating characteristic) curve of of our model, where the overall area under the curve (AUC) is 0.9946. This indicates that the model has a strong ability to distinguish between benignware and malware.

V. DISCUSSION

A. Threats for Disabling Our Work

Due to the inherent limitations of the widely-used static analysis tools, some flow-related information may be missed or inaccurately captured, potentially affecting the accuracy of our detection. For example, the tools are hard to handle anti-analysis mechanisms (*e.g.*, reflection and code obfuscation), which prevents the collection of involving information flows. In the future, we plan to introduce more sophisticated program analysis tools, *e.g.*, hybrid analysis tools [48], to improve the precision and coverage of the collection of flow-related data.

As a ML-based detection technique, MalFlows is also susceptible to adversarial Android malware attacks [49]. Specifically, MalFlows performs malware detection based on the flow-based features extracted from multiple views. Therefore, if the app code is modified from selected few of the views, MalFlows can still detect malware by leveraging the complementary information provided by other views. Moreover, one of the fundamental principles of adversarial example generation is to preserve the functionality of the target app [50]. The flow-based features used by MalFlows are highly correlated with the app's functionality, which contributes to ensuring the invariance of these features. In the future, we plan to select feature sources that are difficult to tamper with, *e.g.*, information flows within C-level code.

Furthermore, our current approach is primarily designed for offline detection scenarios. In future work, we plan to incorporate out-of-distribution learning techniques [26], [43] to establish a more comprehensive detection framework.

B. Selection of HIN Representation Techniques

MalFlows extends a random-walk-based HIN representation method to obtain feature vectors under each views. It is well known that DNN-based HIN representation methods are also commonly used. The rationale behind our technical choice lies in its accuracy and convenience in joint sampling based on multiple meta-paths. As explained in Section III-D2, multi-path joint sampling can capture complex associations

among different apps, which is beneficial for the accurate detection of Android malware. The meta-path-group-guided random walk approach is well suited to our needs and enables accurate capture of joint sampling results from diverse meta-path combinations. Furthermore, the approach is an extension of metapath2vec and is easy to implement.

As a typical DNN-based method, HGT [51] lacks semantic constraints imposed by predefined meta-paths, which may lead to the capture of irrelevant semantics and negatively affect detection performance. Since HAN [31] is inherently designed based on individual meta-paths, adapting it to support our multi-meta-path joint sampling task is non-trivial. It may require redesigning the attention mechanism to simultaneously handle multiple semantic contexts.

C. Usage of Meta-paths and Semantic Fusion Methods

We define 6 meta-paths and divide them into 3 groups based on the views of heterogeneous flows. Meta-paths within a group are used to describe contents and actions of apps under a given view respectively. Experimental results demonstrate that our simplified meta-path design achieves effective detection while maintaining computational efficiency. We will discover more suitable meta-paths automatically by LLMs [52].

MalFlows uses the channel attention to perform the semantic fusion of the heterogeneous flows. Given the semantic independence and interrelation among different meta-path groups, this fusion method is well-suited for our work. In the future, we plan to try more advanced alternatives [53]. Nevertheless, we believe that the selection of the fusion method and the core contributions of our work are orthogonal.

VI. Related Work

A. Flow-based App Analysis

1) Static Analysis Tools: Mainstream detection views in static analysis include data flows, control flows, ICC usages. Specifically, static data-flow tracking techniques [4], [36] detected if there exist sensitive data leaks in apps. MUDFLOW [6] further identified malware based on usage characteristics of sensitive data. Complex-Flows [8] leveraged app behavior along with information flows for classifying benign and malicious Android apps. Static control-flow analysis are commonly adopted to find suspicious trigger conditions in apps [11], [35]. The results of ICC detection can complement the tools above to enhance the overall detection coverage [5]. ICCDetector [9] detected malware based on the specified ICC patterns. Each of the schemes has its own advantages in disclosing a certain type of malicious app behaviors. In comparison, MalFlows unifies the strengths of given schemes for more comprehensive app examinations.

2) Dynamic Analysis Tools: TaintART [54] executed multi-level data-flow tracking for the ART environment. Compared with TaintART, Malton [55] monitored taint propagation instructions at more system layers. In the future, the dynamic information-flow features can be utilized in modeling the HIN and identifying malicious behaviors complementarily.

B. Heterogeneous Graph based Android Malware Detection

HinDroid [23] modeled the relations between APIs and apps with HIN and identifies malware using multi-kernel learning. Scorpion [24] proposed metagraph2vec, a new HIN embedding method, to represent and characterize sly malware. AiDroid [43] performed the HIN in-sample node embeddings and then represents each out-of-sample app with convolutional neural networks. Dr.HIN [25] integrated domain priors generated from different views to devise an adversarial disentangler based on the HIN embeddings. Hawk [26] modeled Android entities and behavioral relationships as a HIN and then identifies malware with graph attention networks. GHGDroid [27] proposed a graph-based approach that leverages a global heterogeneous graph built from sensitive APIs and graph neural embeddings to finish Android malware detection.

MalFlows and the previous HIN-based techniques profile Android apps from disparate perspectives. Specifically, the previous techniques typically build heterogeneous graphs using existence-based information (*e.g.*, APIs or files [23], [24], [26], [27], apk signatures or affiliations [25], [43]). In contrast, MalFlows constructs the HIN from fine-grained program analysis results (*e.g.*, data-flow paths, control dependencies), and then fuses the feature vectors based on their contributions to the malware classification. As a result, MalFlows is able to capture richer behavioral semantics of apps. Moreover, MalFlows introduces a flow-context-aware graph refinement strategy to better exploit flow-related semantics, thereby enhancing the effectiveness of Android malware detection. In the future, MalFlows and the existing techniques can complement each other to enrich the compositions of the HIN, thereby enabling the detection of a wider variety of malware samples.

C. Android Malware Detection based on Semantic Fusion

With the rapid development of deep learning techniques, many advanced Android malware detection approaches based on multi-view fusion are proposed. LensDroid [16] visualized app behaviors from three related but distinct views of behavioral sensitivities, operational contexts, and supported environments. It then fused the corresponding semantics to facilitate Android malware detection. AppPoet [17] designed the multi-view prompt engineering technique based on LLMs to integrate discrete static program features and identify malicious behaviors within apps. Li et al. [56] combined features from both source code and binary code modalities of apps to find their maliciousness. CorDroid [19] developed an obfuscation-resilient Android malware analysis based on enhanced sensitive function call graphs and opcode-based markov transition matrixes. AndroAnalyzer [20] fused microscopic features extracted from abstract syntax trees and the macroscopic features extracted from sensitive function call graphs for Android malware detection. Qiu et al. [18] extracted features from source code, API callgraphs, and Smali opcode to find Android malware. Kim et al. [15] extracted both existence-based and similarity-based features from Android apps, and then fed the resulting vectors into a multimodal deep neural network to identify malware.

Previous studies have primarily focus on fusing app behavior information from different modalities or views that lack explicit semantic correlations. For example, CorDroid utilizes callgraphs and opcodes, while LensDroid combines opcodes with the content of *.so* files. As a result, much of the effort has been devoted to feature extraction and semantic alignment across heterogeneous sources. In contrast, the heterogeneous flow information we incorporate, *e.g.*, function call sequences and control dependencies, naturally exhibits contextual semantic relations. This makes it more suitable for direct semantic fusion and enables more effective characterization and representation of semantics of app behaviors, thereby providing more accurate support for malicious behavior identification.

VII. Conclusion

We propose and implement MalFlows, a novel technique that detects Android malware by fusing heterogeneous flow semantics. Our experiments show that MalFlows achieves an Accuracy of 98.34% and a F_1-score of 0.9881 in the detection, outperforming the selected baseline techniques. We validate the effectiveness of a new context-aware HIN embedding technique named *flow2vec* and demonstrate that the channel-attention-based semantic fusion model for different types of program flows can enhance malware detection.

References

[1] "Mobile Operating System Market Share Worldwide," https://gs.statcounter.com/os-market-share/mobile/worldwide, 2024.

[2] L. Wang, H. Wang, R. He, R. Tao, G. Meng, X. Luo, and X. Liu, "Malradar: Demystifying android malware in the new era," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 2, pp. 1–27, 2022.

[3] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 89–99.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," vol. 49, no. 6. ACM New York, NY, USA, 2014, pp. 259–269.

[5] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *ICSE 2015*, vol. 1. IEEE, 2015, pp. 280–291.

[6] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *ICSE'15*, vol. 1. IEEE, 2015, pp. 426–436.

[7] O. Mirzaei, G. Suarez-Tangil, J. Tapiador, and J. M. de Fuentes, "Triflow: Triaging android applications using speculative information flows," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 640–651.

[8] F. Shen, J. Del Vecchio, A. Mohaisen, S. Y. Ko, and L. Ziarek, "Android malware detection using complexflows," *IEEE Transactions on Mobile Computing*, vol. 18, no. 6, pp. 1231–1245, 2018.

[9] K. Xu, Y. Li, and R. H. Deng, "Iccdetector: Icc-based malware detection on android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1252–1264, 2016.

[10] L. Shi, J. Ming, J. Fu, G. Peng, D. Xu, K. Gao, and X. Pan, "Vahunt: Warding off new repackaged android malware in app-virtualization's clothing," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 535–549.

[11] J. Samhi, L. Li, T. F. Bissyandé, and J. Klein, "Difuzer: Uncovering suspicious hidden sensitive operations in android apps," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 723–735.

[12] A. Arora, S. K. Peddoju, and M. Conti, "Permpair: Android malware detection using permission pairs," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1968–1982, 2019.

[13] 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 *Ndss*, vol. 14, 2014, pp. 23–26.

[14] Y. He, Y. Liu, L. Wu, Z. Yang, K. Ren, and Z. Qin, "Msdroid: Identifying malicious snippets for android malware detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 3, pp. 2025–2039, 2023.

[15] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.

[16] Z. Meng, J. Zhang, J. Guo, W. Wang, W. Huang, J. Cui, H. Zhong, and Y. Xiong, "Detecting android malware by visualizing app behaviors from multiple complementary views," *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 2915–2929, 2025.

[17] W. Zhao, J. Wu, and Z. Meng, "Apppoet: Large language model based android malware detection via multi-view prompt engineering," *Expert Systems with Applications*, vol. 262, p. 125546, 2025.

[18] J. Qiu, Q.-L. Han, W. Luo, L. Pan, S. Nepal, J. Zhang, and Y. Xiang, "Cyber code intelligence for android malware detection," *IEEE Transactions on Cybernetics*, 2022.

[19] C. Gao, M. Cai, S. Yin, G. Huang, H. Li, W. Yuan, and X. Luo, "Obfuscation-resilient android malware analysis based on complementary features," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 5056–5068, 2023.

[20] J. Gong, W. Niu, S. Li, M. Zhang, and X. Zhang, "Sensitive behavioral chain-focused android malware detection fused with ast semantics," *IEEE Transactions on Information Forensics and Security*, 2024.

[21] A. Narayanan, C. Soh, L. Chen, Y. Liu, and L. Wang, "apk2vec: Semi-supervised multi-view representation learning for profiling android applications," in *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2018, pp. 357–366.

[22] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "A multi-view context-aware approach to android malware detection and malicious code localization," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1222–1274, 2018.

[23] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "Hindroid: An intelligent android malware detection system based on structured heterogeneous information network," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 1507–1515.

[24] Y. Fan, S. Hou, Y. Zhang, Y. Ye, and M. Abdulhayoglu, "Gotcha-sly malware! scorpion a metagraph2vec based malware detection system," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 253–262.

[25] S. Hou, Y. Fan, M. Ju, Y. Ye, W. Wan, K. Wang, Y. Mei, Q. Xiong, and F. Shao, "Disentangled representation learning in heterogeneous information network for large-scale android malware detection in the covid-19 era and beyond," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 9, 2021, pp. 7754–7761.

[26] Y. Hei, R. Yang, H. Peng, L. Wang, X. Xu, J. Liu, H. Liu, J. Xu, and L. Sun, "Hawk: Rapid android malware detection through heterogeneous graph attention networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, pp. 4703–4717, 2024.

[27] L. Shen, M. Fang, and J. Xu, "Ghgdroid: Global heterogeneous graph-based android malware detection," *Computers & Security*, vol. 141, p. 103846, 2024.

[28] Y. Sun, *Mining heterogeneous information networks*. University of Illinois at Urbana-Champaign, 2012.

[29] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[30] Y. Dong, N. V. Chawla, and A. Swami, "metapath2vec: Scalable representation learning for heterogeneous networks," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 135–144.

[31] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in *The world wide web conference*, 2019, pp. 2022–2032.

[32] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *MSR'16*. IEEE, 2016, pp. 468–471.

[33] Z. Meng, Y. Xiong, W. Huang, L. Qin, X. Jin, and H. Yan, "Appscalpel: Combining static analysis and outlier detection to identify and prune undesirable usage of sensitive data in android applications," *Neurocomputing*, vol. 341, pp. 10–25, 2019.

[34] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *CETUS*, 2011.

[35] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: learning-based, large-scale discovery of hidden sensitive operations in android apps," in *Proc. of NDSS*, 2017.

[36] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, pp. 1–32, 2018.

[37] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *NDSS*, 2014.

[38] X. Fu, J. Zhang, Z. Meng, and I. King, "Magnn: Metapath aggregated graph neural network for heterogeneous graph embedding," in *Proceedings of the web conference 2020*, 2020, pp. 2331–2341.

[39] Y. Ye, S. Hou, L. Chen, X. Li, L. Zhao, S. Xu, J. Wang, and Q. Xiong, "Icsd: An automatic system for insecure code snippet detection in stack overflow over heterogeneous information network," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 542–552.

[40] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[41] "Hindroid-with-embeddings." https://github.com/davidzyx/HinDroid-with-Embeddings, 2025.

[42] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, pp. 1–34, 2019.

[43] Y. Ye, S. Hou, L. Chen, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao, "Out-of-sample node representation learning for heterogeneous graph in real-time android malware detection," in *28th International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

[44] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th international conference on world wide web*, 2015, pp. 1067–1077.

[45] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.

[46] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 757–770.

[47] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "{TESSERACT}: Eliminating experimental bias in malware classification across space and time," in *28th USENIX security symposium (USENIX Security 19)*, 2019, pp. 729–746.

[48] Y. Tsutano, S. Bachala, W. Srisa-an, G. Rothermel, and J. Dinh, "Jitana: A modern hybrid program analysis framework for android platforms," *Journal of Computer Languages*, vol. 52, pp. 55–71, 2019.

[49] P. He, Y. Xia, X. Zhang, and S. Ji, "Efficient query-based attack against ml-based android malware detection under zero knowledge setting," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 90–104.

[50] H. Li, Z. Cheng, B. Wu, L. Yuan, C. Gao, W. Yuan, and X. Luo, "Black-box adversarial example attack towards {FCG} based android malware detection under incomplete feature information," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1181–1198.

[51] Z. Hu, Y. Dong, K. Wang, and Y. Sun, "Heterogeneous graph transformer," in *Proceedings of the web conference 2020*, 2020, pp. 2704–2710.

[52] L. Chen, F. Xu, N. Li, Z. Han, M. Wang, Y. Li, and P. Hui, "Large language model-driven meta-structure discovery in heterogeneous information network," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 307–318.

[53] X. Huang, R. Zhang, Y. Li, F. Yang, Z. Zhu, and Z. Zhou, "Mfc-acl: Multi-view fusion clustering with attentive contrastive learning," *Neural Networks*, vol. 184, p. 107055, 2025.

[54] M. Sun, T. Wei, and J. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 331–342.

[55] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for art," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 289–306.

[56] X. Li, L. Liu, Y. Liu, Y. Zhao, P. Zhang, and H. Liu, "Multimodal fusion for android malware detection based on large pre-trained models," *IEEE Transactions on Software Engineering*, 2025.