DeUEDroid: Detecting Underground Economy Apps Based on UTG Similarity

Zhuo Chen
hypothesiser.hypo@zju.edu.cn
Zhejiang University
& Ant Group
China

Jie Liu
qingyang.lj@antgroup.com
Ant Group
China

Yubo Hu
yubohu@stu.xidian.edu.cn
Xidian University
China

Lei Wu*
lei_wu@zju.edu.cn
Zhejiang University
China

Yajin Zhou
yajin_zhou@zju.edu.cn
Zhejiang University
China

Yiling He
yilinghe@zju.edu.cn
Zhejiang University
China

Xianhao Liao
xianhao.lxh@antgroup.com
Ant Group
China

Ke Wang
kaywang.wk@antgroup.com
Ant Group
China

Jinku Li
jkli@xidian.edu.cn
Xidian University
China

Zhan Qin
qinzhan@zju.edu.can
Zhejiang University
China

ABSTRACT

In recent years, the underground economy is proliferating in the mobile system. These underground economy apps (**UEware** for short) make profits from *providing non-compliant services*, especially in sensitive areas (*e.g.*, gambling, porn, loan). Unlike traditional malware, most of them (over 80%) do not have malicious payloads. Due to their unique characteristics, existing detection approaches cannot effectively and efficiently mitigate this emerging threat.

To address this problem, we propose a novel approach to effectively and efficiently detect UEware by considering their UI transition graphs (UTGs). Based on the proposed approach, we design and implement a system, named DeUEDroid, to perform the detection. To evaluate DeUEDroid, we collect 25,717 apps and build up the first large-scale ground-truth dataset (1,700 apps) of UEware. The evaluation result based on the ground-truth dataset shows that DeUEDroid can cover new UI features and statically construct precise UTG. It achieves 98.22% detection F1-score and 98.97% classification accuracy, a significantly better performance than the traditional approaches. The evaluation result involving 24,017 apps demonstrates the effectiveness and efficiency of UEware detection in real-world scenarios. Furthermore, the result also reveals that UEware are prevalent, *i.e.*, 54% apps in the wild and 11% apps in the app stores are UEware. Our work sheds light on the future work of analyzing and detecting UEware. To engage the community, we have made our prototype system and the dataset available online.

*Corresponding author.

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598051

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**.

KEYWORDS

Underground economy, UI transition graph, Machine learning

ACM Reference Format:
Zhuo Chen, Jie Liu, Yubo Hu, Lei Wu, Yajin Zhou, Yiling He, Xianhao Liao, Ke Wang, Jinku Li, and Zhan Qin. 2023. DeUEDroid: Detecting Underground Economy Apps Based on UTG Similarity. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598051

1 INTRODUCTION

Mobile apps have been an indispensable part of our daily life, even in some sensitive areas, such as adult content, financial business [6], and online gambling. However, due to the huge economic benefits, apps that serve the underground economy are prevalent in these areas nowadays [2, 11, 33], and thereby lead to serious damages. For example, the underground porn apps caused more than $304 million losses in 2020 [1]. Meanwhile, underground gambling apps made more than $1 billion revenue in Malaysia in 2021 [13].

Unlike traditional malware, these underground apps make profits by *providing non-compliant services*, especially in sensitive areas (*e.g.*, gambling, porn, loan). For better understanding, Figure 1 shows a typical use scenario of a normal [1] loan app and an underground loan app. Unlike the normal app, the underground app does not provide UIs related to *terms and conditions*, including identity verification and user (or privacy policy) agreement. Such a phenomenon is probably because these apps want to avoid supervision and let the victims make a quick decision without thinking [9]. What's more, our investigation shows that most of these underground apps (over 80%) do not have malicious payloads (or serve advertisements).

[1]In this paper, apps that do not belong to UEware are named as *normal apps*.

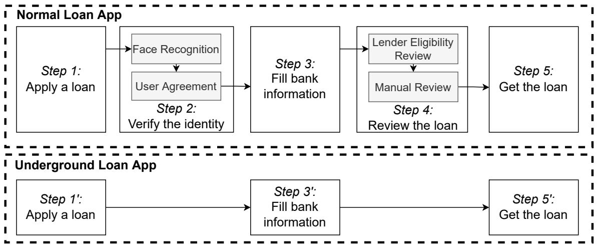

Figure 1: An example of a normal loan app vs. an underground loan app. The underground loan app only provides a single usury service (Steps 1',3',5') without verifying the identity (Step 2) and reviewing the loan (Step 4).

In this work, we call these apps *underground economy apps* (or **UEware** for short), and define them as *apps that serve the underground economy by providing non-compliant services* [2]. Obviously, the concept is orthogonal to that of malware.

The proliferation of UEware has become a widespread issue, particularly on platforms that lack adequate supervision, such as social media and third-party app markets. To mitigate this threat, authoritative agencies and app download platforms proactively seek effective solutions. However, to our best knowledge, only a few studies [23, 29, 32, 48, 49] focus on a special sort of UEware (*i.e.*, gambling scam, loan scam apps), and provide some recommendations to users and authorities. Nonetheless, none of these works propose feasible approaches which are capable of effectively detecting UEware. Apart from the effectiveness, efficiency is also significant due to the huge amount of apps [3] that need to be handled.

Unfortunately, existing dynamic and static approaches that are used to detect malware cannot be applied to detect UEware. On one hand, the dynamic approaches are inefficient to perform the large-scale detection due to the inherent scalability limitation, i.e., they have to launch the apps to examine their behaviors [59] and fetch the screenshots [26], thus consuming huge resources [40]. On the other hand, the static approaches are ineffective to detect UEware due to its unique characteristics. Specifically, the features used by those detection approaches can be dived into malicious payload [27, 42, 60, 68], GUI content [54, 69], and the Manifest information [28, 38, 53]. However, these features are not feasible to UEware: 1) *most UEware (over 80%) do not have malicious payloads*, which inevitably makes the payload-based approaches ineffective; and 2) *most UEware take countermeasures*, which also makes the content-based approaches ineffective; and 3) the Manifest information cannot be used as a unique detection feature, since it is the basic information and can be arbitrarily customized in Android.

Our approach. In this paper, we propose a novel approach to efficiently and effectively detect Android UEware by identifying the non-compliant services they provided. Specifically, an app service is presented to the users through multiple user interfaces (UIs) in a specific order, which constitutes the *UI transition graph* (**UTG**).

Our approach is based on the following two key observations. First, *the UTGs of UEware are different from those of the normal apps.* Even UEware imitate the UI of normal apps, their UTGs vary greatly. As discussed earlier, UEware do not provide UIs related to *terms and conditions*, which will always be provided by normal apps. Furthermore, UEware are generally implemented to serve only one purpose (*e.g.*, usury application), while normal apps are usually served for multiple services (*e.g.*, financial derivatives purchasing and asset evaluation). Second, *the UTGs of the same type of UEware are extremely similar, while the UTGs of different types of UEware vary from each other.* This is because different services share different UTGs. Based on these observations, we can use UTG as the key feature to perform the similarity-based detection.

Challenges. However, identifying UEware based on UTG is a challenging task. First, *it is difficult to directly leverage UTG to perform the UEware detection.* Although UTG has widely been adopted to assist the testing and security vetting in the previous studies [19, 55, 61], to our best knowledge, it has never been used as a detection feature. This is due to the fact that UTG is expressed via a mixture of graph topology and unstructured GUI widget attributes. As a result, their correlations cannot easily be applied as an effective representation. Second, *it is not easy to statically build a precise UTG.* Some new UI features (*e.g.*, WebView, fragment and navigation) are supported by Android and have been widely used in recent years. However, (even the latest) tools/systems proposed by the previous studies [37, 44, 50, 52, 52] cannot cover these new features. This inevitably leads to the imprecise UTG construction.

This work. In this study, we propose the first UTG based detection approach by addressing the above two challenges. For the first challenge, we propose a graph-embedding [31, 58] based method to represent UTG by properly correlating multi-dimensional attributes. Specifically, the graph topology and the encoded attributes [4] are aggregated and handled by applying the graph-embedding technique [31, 58] to generate the UTG representation. For the second challenge, we propose a taint-based construction method which can cover new features (*i.e.*, identifying GUI elements and determining UI transitions) to build up a more precise UTG.

We have implemented a prototype, DeUEDroid, to detect UEware. DeUEDroid consists of three modules: *(i) UTG builder*, which is used to construct the UTG. This module consists of two sub-modules, *i.e.*, *GUI widget identification* and *UI transition determination. (ii) UTG feature extractor*, which is responsible for extracting the UTG feature. *(iii) UEware detector.* The detector leverages the self-supervised learning to detect and classify UEware based on UTG similarity.

We build up three datasets to evaluate DeUEDroid. First, we use 15 source-available apps to evaluate the UTG construction accuracy. The evaluation result suggests that DeUEDroid is effective in UTG construction, which can accurately identify imprint tokens of WebView (over 85% F1-score) and transitions (92.3% F1-score in total). Specifically, the transitions identified by DeUEDroid achieves a significantly better F1-score than the State-Of-The-Art (SOTA) results (*i.e.*, 35.5% higher than IC3 [50], 67.7% higher than GATOR [52]). Then, we build up the ground-truth UEware dataset, which is the first large-scale ground-truth dataset (1,700 apps) of UEware, to evaluate the effectiveness of UEware detection and classification.

[2]For a given sensitive service, the regulations may vary in different countries/regions, *e.g.*, online gambling is prohibited in Saudi Arabia, while it is legal in some states of the United States [5]. The impact is discussed in Section 9.

[3]*E.g.*, there were over two million new apps released on GooglePlay in 2021 [12].

[4]Different attributes will be encoded with different methods, see Section 6.

The evaluation result shows that DeUEDroid is effective in UEware detection and classification (with 98.22% detection F1-score and 98.97% classification accuracy). Our system has a significantly better performance than the traditional approaches. Finally, we perform a large-scale experiment on 24, 017 apps collected from both app stores and the wild, to measure the effectiveness and efficiency of UEware detection in the real world. The evaluation result shows that UEware are prevalent, *i.e.*, 54% apps in the wild and 11% apps in the app stores are UEware.

Contributions. We make the following contributions:

- We proposed a novel approach to effectively and efficiently detect UEware based on UTG similarity, including a novel technique that can cover new UI features and statically build precise UTG, and a novel algorithm that can efficiently embed UTG with different dimension attributes.
- We implemented a prototype named DeUEDroid. The evaluation results demonstrate the effectiveness and efficiency of the system, with 98.22% UEware detection F1-score and 98.97% classification accuracy. And it is capable of performing large-scale detection to mitigate the real-world threats.
- We found that UEware are prevalent, *i.e.*, 54% apps in the wild and 11% apps in the app stores are UEware.
- We built up the first large-scale ground-truth UEware dataset with 1, 700 apps. We have released our system and the dataset on the github [7] to engage the community.

2 BACKGROUND

2.1 Android User Interface

In an Android app, the activity is the fundamental component for drawing the GUI with which users can interact with [4]. And since Android 3.0, fragments can also render the GUI. A GUI consists of GUI widgets (*e.g.*, button and textView) and layout models (*e.g.*, Linearlayout) that describe how to arrange GUI widgets. These widgets and layout models inherit from the *view* class, and are statically recorded in the XML file or added in code. What's more, web widgets can launch a local browser to load a web page.

An android app consists of multiple activity/fragment windows for handling complex business requirements. When some special conditions are triggered (*e.g.*, click event *onclick*), the app will transit from one window to another. Specifically, these GUI transitions are triggered in multiple ways: *(i)* triggered by explicitly invoking activity transition API, *e.g.*, the *StartActivity, StartActivityForResult*; *(ii)* triggered by implicitly invoking conversions, *e.g.*, Android Inter-Component Communication (*ICC*); *(iii)* triggered by invoking hardware event, *e.g.*, *BACK*; *(iv)* triggered by fragment navigation, *e.g.*, the *NavController* manages fragment navigation within a *NavHost*. Note that fragment navigation extremely improves the flexibility of transitions, and has been widely used in recent years [8].

2.2 Modeling of Android User Interface

An android app is presented to the users through multiple runtime-rendered user interfaces (UIs) in a specific order. However, runtime rendered visuals cannot be restored in static analysis. To this end, the UI modeling studies [21, 26, 44] focus on expressing UI through static analysis in a limited time. However, due to the rapid revolution of Android, there are some new features that cannot be handled well. In this study, to accommodate the new UI features (*e.g.*, fragment, navigation, web widget), we fine-tune the UI transition graphs (UTG) and propose our definition as follows.

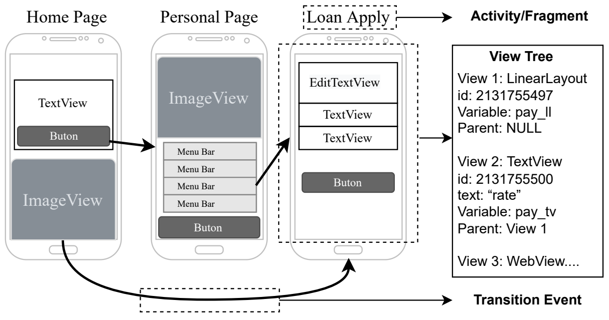

Figure 2: Simplified example of a loan app UTG.

Definition I: A UTG is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in a node attribute space Ω, where:

(1) $\mathcal{V}$ is a node set, we consider 4 categories of windows that users can interact with as a node in a UTG: activities, fragments, menus, and dialogs. Activities and fragments are often presented as full-screen windows, while menus and dialogs are short-lived windows that often require the user to take actions.
(2) Each node is assigned GUI view tree attributes in Ω, including native widget attributes (*e.g.*, layout hierarchy, widget type, text), and web widget attributes (*i.e.*, network imprint [22]);
(3) Directed edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of transitions between activity/fragment and ε is the single edge within the $\mathcal{E}$.

For better understanding, we show a simplified underground loan app UTG in Figure 2 as an example. It consists of three activity/fragment nodes with GUI attributes (view tree), and three transition events as the directed edges.

3 MOTIVATION

In this section, we first illustrate the limitation of the traditional detection approaches. As discussed in Section 1, the dynamic approaches are not feasible to perform large-scale detection, hence here we only focus on the static approaches. After that, we detail the feasibility of our approach by demonstrating the validity of the two key observations.

3.1 Limitation of the Traditional Approaches

Features used by those traditional detection approaches can be categorized into the following three categories: *Malicious Payload*, *GUI*, and *Manifest*, as shown in Table 1. However, all of these features are ineffective to detect UEware due to the following reasons:

- **Lack of malicious payload.** Most UEware do not have any malicious payloads (over 80% through our experiment in Section 8.2.2), which inevitably leads to the failure of malware detection approaches (*e.g.*, API flow detection).
- **Incomplete & Disguised GUI content.** Due to the strict censorship in sensitive areas, most UEware adopt countermeasures: *(i)* hide the sensitive GUI content (*e.g.*, pictures, text) on remote servers. *(ii)* disguised as normal apps, such as using similar icons.

Table 1: The feature selection in previous static detection studies.

Static Detection Approach	Studies	Feature Selection								
		Malicious payload				GUI			Manifest	
		Bytecode	API	Method	Activity	Native widget	Web widget	GUI content	Dveloper signature	Permission
Signature Based	ViewDroid [71], MassVert [20]	-	-	✔	-	✔	-	-	-	-
	DriodMoss [43]	✔	-	-	-	-	-	-	-	-
	Retriever [53], AndroSimilar [28]	-	-	-	-	-	-	-	✔	-
	Kirin [27], DroidMat [60]	-	✔	-	-	-	-	-	-	✔
Machine Learning Based	Kim et al. [36]	-	✔	✔	-	-	-	-	-	✔
	DroidSIFT [70], SIGPID [38], Droid-Sec [15]	-	✔	-	-	-	-	-	-	✔
	Tiger [22]	-	-	-	-	-	✔	-	-	-
	Malena [69], Sun et al. [54]	-	-	-	-	-	-	✔	-	-
	Mamadroid [46]	-	✔	-	-	-	-	-	-	-
	Drebin [16]	-	✔	-	✔	-	-	✔	-	✔

These countermeasures result in the failure of content-based detection (see Section 8.2.2).

- **Ineffective Manifest.** The Manifest information cannot be used as the unique features to detect UEware. Specifically, the permissions acquired by UEware are similar to those of normal apps [29], while the developer signatures can always be arbitrarily customized (in Android).

3.2 Feasibility of Our Approach

To perform an effective large-scale detection, we propose an UEware detection/classification approach based on the UTG similarity. Our approach is based on the following two key observations:

- **Observation-I:** UTGs of UEware are different from those of the normal apps.
- **Observation-II:** UTGs of the same UEware type are similar, while those of different types vary.

For the first observation, we have shown an example of a normal loan app vs. an underground loan app in Figure 1. What's more, we additionally select some other apps [5] for comparison. We point out that UEware always do not provide UIs related to terms and conditions and are only served for one purpose (*e.g.*, usury application), which leads to considerable differences in UTG structure. Through our experiment, we even find that the UTG of UEware is different from that of normal ones in statistic (*i.e.*, 11 vs. 22 transition events and 29 vs. 227 widgets, respectively), see details in Section 8.2.

To demonstrate the second observation, we randomly select another underground financial app (with a different Manifest). Although their runtime screenshots are different, they share a similar UTG (*e.g.*, home page, loan apply, bank card check). And then, we randomly select an underground gambling app for comparison. The gambling app provides many game windows rather than the loan applications. Due to the different services they provided, the UTGs of these two types of apps inevitably vary from each other.

4 DESIGN OVERVIEW

Based on the proposed approach, we design a system named DeUEDroid to detect and classify UEware. Specifically, DeUEDroid first statically builds precise UTG for UEware, and then applies a graph-embedding based method to represent UTG. After that, it leverages the self-supervised method to detect and classify UEware based on UTG similarity. Figure 3 shows the architecture of DeUEDroid.

[5]Due to the page limit, all the details of those examples are shown in https://github.com/HypoopyH/DeUEDroid.

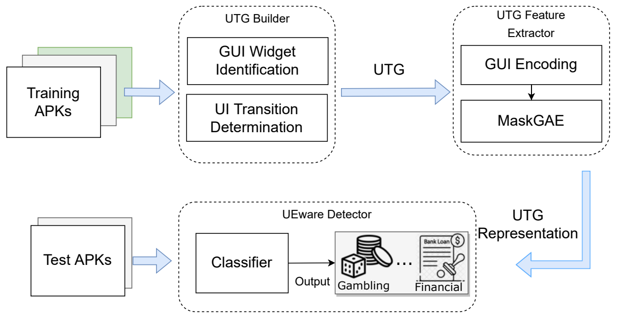

Figure 3: The design overview of DeUEDroid.

There are three modules, *i.e.*, *UTG Builder*, *UTG Feature Extractor* and *UEware Detector*, as follows:

- **UTG Builder.** This module accepts Android APK files as the input, and outputs the corresponding UTGs. It consists of two sub-modules: *GUI widget identification* and *UI transition determination*. Specifically, the first sub-module covers both the native widgets and the web widgets; while the second sub-module applies a novel UTG construction algorithm to accurately capture UI transitions.
- **UTG Feature Extractor.** Based on these UTGs, this module combines both UTG topology and GUI attributes together and outputs the UTG representation (feature). Specifically, it first leverages multiple encoding methods to handle GUI attributes of different dimensions. After that, it applies a novel algorithm to correlate the graph topology and the GUI attributes.
- **UEware Detector.** Based on the UTG feature, this module leverages the self-supervised learning to train an UEware classifier, which will be used to perform the detection. Note that the UTG based approach allows further classifying the apps into different categories of UEware. Currently, three underground categories, including *gambling*, *porn* and *financial*, are supported.

In the following, we will detail these three modules in Section 5, Section 6 and Section 7, respectively.

5 UTG BUILDER

In this section, we describe the design of UTG builder. This module has two sub-modules: *(i)* GUI widget identification, and *(ii)* UI transition determination.

All sub-modules take an Android APK file as the input. The output of GUI widget identification is a set of GUI widgets. Notably, we collect the layout hierarchy, text, and widget type of native widgets. And the web widget (*e.g.*, WebView) attributes are represented by their network imprints. The output of UI transition determination is a directed graph representing the transition relationship between activities/fragments. In the end, the UTG is well constructed as a directed graph with GUI attributes.

5.1 GUI Widget Identification

In Android, the GUI widgets can be dived into native widgets and web widgets. Specifically, DeUEDroid proposes different analysis methods for two types of widgets.

Native Widgets. As mentioned in Section 2, the native widgets can be statically recorded in the XML file or dynamically added in code. To the XML widgets, their widget types (*e.g.*, ImageView, Button), the text (*e.g.*, @string/ic_hello), and the layout hierarchy (*e.g.*, ConstrainLayout) are well organized in the corresponding XML files. In addition, the activities/fragments load their corresponding XML widgets based on the API calls *findViewbyID*. To the dynamically added widgets, they are dynamically added in code (*e.g.*, new TextView()), and their attributes are recorded in the code, which requires taint flow analysis to identify. These widgets are rendered within the "*OnCreate*" phase when a activity/fragment is calling up.

There have been several studies [34, 63] to perform native widgets analysis. Considering the efficiency, we build up our system based on a SOTA lightweight static UI analysis framework, FrontMatter [37], to figure out the native widgets.

Web Widgets. Nowadays, the hybrid development paradigm is commonly used [10], which acquires remote resources through web widgets. Specifically, the WebView starts a local browser and invokes the web APIs (*e.g.*,"*LoadURL*") to load a web page. These widgets are an important part of UTG, but are not properly analyzed by native widget analysis studies.

For these web widgets, the essential part is the URL parameter. However, web URLs are always dynamically concatenated, which is hard to restore in static analysis [22, 25]. But we have two observations about web widgets: *(i)* the URL parameter and its related tokens are all in string type. *(ii)* almost always, a parameter-related token originates from some constant values within the program, such as constants, and XML resources. To this end, we refer to the previous studies [22] and generate the network imprint to represent the web widget attributes.

Here we use an example (see Figure **??** in Appendix A) to present the imprint generation. First, the program loads activities/fragments, and searches special API calls (*e.g.*, *WebView.loadURL()*). Once the sink statement is found in Class Y method B, the sink variable is *url*. Then the program will perform backward taint analysis to check whether any variable affects *url*. After running, *v3* and *X.A()* is determined to affect *url* using *java.lang.StringBuilder.append()*. And the *v3* is considered unrelated to any invariant source within the app but instantiated from runtime variables, so we discard the *v3*. Turn to the *X.A()*, since it is an inter-procedure call, the program additionally loads Class X and sinks the *v1*. Going further, the program finds that *R.string.baseUrl* and *v2*. At this time, *v2* is determined to be an immediate variable, so the program instantiates *v2* with a concrete value and instantiated *R.string.baseUrl* from XML file. To sum up, *R.string.baseUrl,v2* are instantiated as constant variables, *v3* is instantiated as runtime variable and discarded. In the end, the program outputs a set of tokens (*e.g.*, [*R.string.baseUrl,v2*])

Algorithm 1: The UTG Construction Algorithm

```
Data: $apk$: the APK file.
Result: $UTG = (Nodes, Edges)$: the UI transition graph for $apk$.
Function UTG_Construction($apk$):
    $Nodes \leftarrow \{\}$;
    $Edges \leftarrow \{\}$;
    $cg \leftarrow$ get_CallGraph($apk$);
    $classes \leftarrow$ get_AllClasses($apk$);
    for $class$ in $classes$ do
        if is_Fragment($class$) or is_Activity($class$) then
            $Nodes \cup \{class\}$;
        end
        $methods \leftarrow$ get_AllMethods($class$);
        for $method$ in $methods$ do
            $units \leftarrow$ get_AllUnits($method, cg$);
            for $unit$ in $units$ do
                if is_Transition($unit$) then
                    $callees \cup$ get_Callees($unit$);
                    $callers \cup$ get_CallerActs($class, cg$);
                    $Edges \cup \{callers, callees\}$;
                end
            end
        end
        if has_Navigation($class$) then
            $caller \leftarrow$ get_CallerActs($class, cg$);
            $callees \leftarrow$ get_NavTargets($class$);
            $Edges \cup \{caller, callees\}$;
        end
    end
    return $UTG(Nodes, Edges)$;
Function get_CallerActs($class, cg$):
    if is_Fragment($class$) then
        $callers \leftarrow \{class,$ get_ActsWithFragment($class, cg$)$\}$;
    else if is_InnerClass($class$) then
        $callers \leftarrow$ get_OuterActs($class, cg$);
    else $callers \leftarrow class$ ;
    return $\{callers\}$;
```

and removes some common tokens (*e.g.*, *github*). These tokens are used as network imprints to identify web widget attributes.

5.2 UI Transition Determination

The GUI widgets and layout modules are rendered on activities/fragments. And when a new window is opened, it causes a transition. Existing studies [18, 21, 44, 50, 64] mainly identify transitions among activities, but overlook the *fragment-related transitions* (*e.g.*, the transitions between fragments and activities) and the *navigation-based transitions* (*e.g.*, navigation among fragments). As the latter two have been widely adopted to develop Android apps in recent years, the lack of support for them will inevitably lead to imprecise UTG. To accurately build up the transition events, we propose a new **UTG Construction Algorithm** 1. This algorithm can cover the fragment-related transitions and the navigation-based transitions, and build a precise UTG.

Specifically, we first initialize *Nodes* and *Edges* as empty sets, which store the activities/fragments and their transition relationships respectively. Then we generate the call graph of the given APK, and fetch out all classes defined in the Manifest. Note that, there are some implicit run and start pairs (*e.g.*, AsyncTask, OnClickListener, Runnable, and Message) in Android, which need to be bridged to make the call graph complete.

For each class in the APK, we first determine whether it inherits from activity or fragment or not. If so, it will be added to the *Nodes* set. Then for all methods in each class, we get all units according to the previous call graph. For these units, we will determine whether they have transition movements. Specifically, for the explicitly transition and implicitly transition, they have special APIs (*e.g.*, *StartActivity()*, or *Intent()*). By comparing the API signatures, we can locate their transition units. Besides, the navigation-based transitions are implemented by special structures (*e.g.*, *NavController*). We can locate the navigation unit by searching the variable structures. Finally, there are some hardware events (*e.g.*, *BACK* button), we monitor whether the activity has overloaded system-level calls, to implement the transition.

Furthermore, for each transition unit, we first get the targets (callees) by analyzing the unit arguments based on taint analysis. Due to the difficulty in handling conditional expressions, all possible target activities/fragments are included. Then we get the source (callers). Specifically, three different processing methods are adopted according to the class type.

- If the class is a fragment, the callers are the union of this fragment and all activities that own this fragment. Because activities can trigger transitions of fragments they own.
- If the class is an inner class, the callers are the union of the outer activities/fragments of this inner class.
- If the class is not an inner class nor a fragment, the caller is the current class itself.

Finally, for callers and callees found by a transition, we traverse all of them and add all pairs (caller $\longrightarrow$ callee) to the *Edges* set.

6 UTG FEATURE EXTRACTOR

In this section, we describe the design of UTG feature extractor. The overview of this module is shown in Figure 4. For an input UTG, we first encode the GUI attributes Ω into matrix X. And then, we propose a novel algorithm, MaskGAE, to correlate the graph topology $\mathcal{G}$ and GUI attributes X as the UTG representation Z.

6.1 GUI Encoding

To properly integrate different dimension attributes, we adopt multiple encoding approaches. Specifically, for an activity/fragment node, we first segment the text string and network imprint from GUI widgets. To utilize these string-type attributes, we use a pre-trained Word2Vec [47] model to generate their representations. After that, we encode the ViewTree, which consists of multiple GUI widgets. Note that, a GUI widget can be an instance of a system widget class (*e.g.*, button) or a customized widget class inherited from the view class. Thus, we regard the widget type as the GUI widget's "tag", and use the one-hot model to encode and serialize it. Further, to the layout hierarchy, we traverse the ViewTree and generate a widget sequence. Finally, we combine all representations into the vector x_0 of the node. And since a UTG may consist of multiple nodes, the final GUI matrix X is combined by every node representation. Meanwhile, the transition events and the activity/fragment nodes have been represented by the UTG topology $\mathcal{G}$.

6.2 MaskGAE

In the underground economy, developers always generate new apps by modifying parts of the existing apps. To counteract the impact of code modifications, it requires high robustness of the graph embedding model. To this end, we propose a novel algorithm, MaskGAE, which adopts the Mask strategy before the UTG embedding. Specifically, the Mask strategy can be viewed as an adversarial attack that provides a new graph as data augmentation. In this paper, we adopt Edge-wise random masking, which is defined as:

$$\varepsilon_{mask} \sim Bernoulli(p) \tag{1}$$

where ε denotes the edge within the $\mathcal{E}$.

After the mask, we follow the success of the graph autoencoder (GAE) [31], which is designed to reconstruct graph inputs, to correlate the graph topology and GUI attributes. In this study, our encoder f_θ is 2-layer graph convolutional networks (GCN) [24], a well-established GNN architecture.

Notably, due to the masking strategy, our loss calculation consists of two parts: local reconstruction loss L_{local} and global contrastive loss L_{global}. For the masked graph, we divide the edges as remaining edges and masked edges. The masked edges are selected as positive samples, while the disconnected node's edges are selected as negative samples. For the embeddings from the decoder, we calculate their inner product as the probability of the edge existing.

$$\begin{aligned}\mathcal{L}_{Local} = -(\frac{1}{|\varepsilon^+|} \sum_{(u,v)\in\varepsilon^+} \log h_w(z_u, z_v) \\ +\frac{1}{|\varepsilon^-|} \sum_{(u',v')\in\varepsilon^-} \log\left(1 - h_w(z_{u'}, z_{v'})\right))\end{aligned} \tag{2}$$

where $\mathcal{Z}$ is the graph embedding result from the encoder; ε^+ is a set of positive edges while ε^- is a set of negative edges sampled from the graph.

Further, we calculate the distance between the two representations $(Z, \widehat{Z})$ as the global contrastive loss. Specifically, we normalize the loss value into the range of $[0, 1]$ to facilitate optimization.

$$\mathcal{L}_{Global} = \frac{\sum_{i=1}^{N}(z_i - \widehat{z}_i)^2}{N} \tag{3}$$

Finally, we combine the global loss and local loss into the total loss and use gradient descent to minimize it.

$$\mathcal{L}_{Total} = \mathcal{L}_{Local} + \alpha\mathcal{L}_{Global} \tag{4}$$

Where α denotes a hyperparameter trading off two terms.

7 UEWARE DETECTOR

In this section, we describe the design of UEware detector. The UEware detector leverages the self-supervised learning, which is widely adopted in previous studies [30, 58]. In detail, it consists of two stages: self-supervision training task, and downstream training task, see Figure 5.

The self-supervised training task also refers to GAE, however, it is different from the UTG feature extractor. In detail, the self-supervision training task accepts the APK relation graph $\mathcal{H}$ and the APK feature X, and trains the encoder f_ψ. Referring to previous studies [73], the APK relation graph $\mathcal{H}$ is an undirected graph, while the node of the graph is APK, and the edge of the graph is the overlap Manifest information (*i.e.*, PackageName, AppName, developer signature) between APKs. Meanwhile, the APK feature X is the UTG representation produced by encoder f_θ. And the

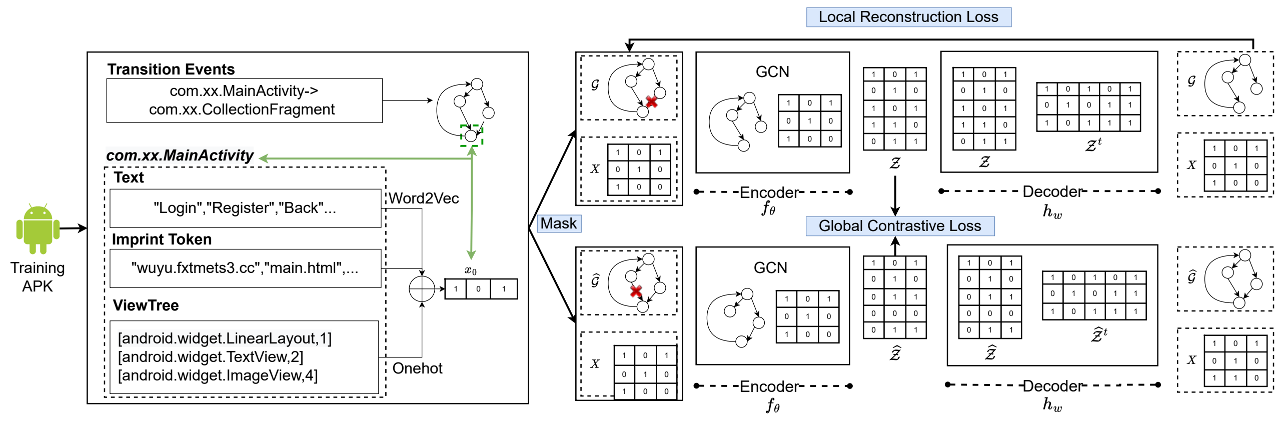
Local Reconstruction Loss
Transition Events
com.xx.MainActivity->com.xx.CollectionFragment
com.xx.MainActivity
Text
"Login","Register","Back"...
Word2Vec
Imprint Token
"wuyu.fxtmets3.cc","main.html",...
ViewTree
[android.widget.LinearLayout,1]
[android.widget.TextView,2]
[android.widget.ImageView,4]
Onehot
Training APK
Mask
GCN
Encoder f_θ
Decoder h_w
Global Contrastive Loss

Figure 4: The design of UTG feature extractor.

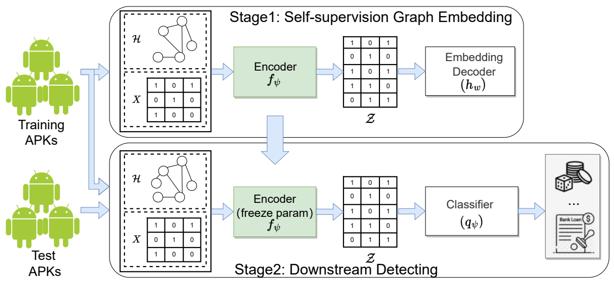
Stage1: Self-supervision Graph Embedding
Training APKs
Encoder f_ψ
Embedding Decoder (h_w)
Test APKs
Encoder (freeze param) f_ψ
Classifier (q_ψ)
Stage2: Downstream Detecting

Figure 5: The design of UEware detector.

downstream task uses the encoder f_ψ to train the classifier q_ψ for UEware detection.

Classifier: The APK encoder f_ψ learned from self-supervision training is set frozen and directly introduced into the downstream task. According to this frozen encoder, the downstream task then calculates the $\mathcal{Z}$. Finally, the classifier q_ψ (also called the downstream decoder) is trained based on the labeled dataset. In this study, we leverage a linear classifier (*i.e.*, a logistic regression model) and the labeled information is the app types (*i.e.*, underground gambling, underground porn, underground financial, and normal apps). Specifically, the detection task decoder parameter is calculated as:

$$\psi = argmin \mathcal{L}_{sup}(f_\theta, q_\psi, \mathcal{H}, y) \tag{5}$$

where $\mathcal{H}$ is the APK relation graph and y is the label of UEware.

In the end, when input test APKs, the DeUEDroid first combine the test APKs and training APKs together to construct a new graph $\mathcal{H}$. And then, the DeUEDroid labels the test APKs through the encoder f_ψ and decoder q_ψ.

8 IMPLEMENTATION AND EVALUATION

We have implemented a prototype of DeUEDroid. Specifically, we leverage Soot [56] and FlowDroid [17] for taint analysis and Frontmatter [37] for GUI analysis. In network imprint analysis, we additionally adopt Java string analysis techniques [25], and customize the conditions of instantiate. We deploy DeUEDroid on a server has 16 cores with 2.1GHz CPU, 256GB memory, and 8TB hard drives.

Furthermore, we will evaluate DeUEDroid by answering the following three research questions:

- **RQ1**: How effective is DeUEDroid in UTG construction?
- **RQ2**: How effective is DeUEDroid in UEware detection?
- **RQ3**: How effective and efficient is DeUEDroid in large-scale UEware detection?

8.1 Evaluation Setup

For RQ1, we investigate the capability of UTG construction. To evaluate the capability, we build up the **source-available dataset**, consisting of 5 self-developed apps and 10 real-world apps. Specifically, we developed 5 apps as our ground-truth benchmark, which covers different transitions (*i.e.*, the transitions between activity - activity, activity - fragment, fragment - fragment, and navigation-based transitions). This method is used to accurately understand the ground truth of transitions and widgets, and it has been widely adopted by previous studies [21, 39].

In addition, to make our experiments more convincing, we compare the SOTA works (*i.e.*, IC3 [50], Gator [52]) and analyze several famous open-sourced apps [6]. in f-droid [3]. Note that there are some studies [21, 22] that focus on static transition identification or imprint generation. However, we cannot compare them since they do not release the code or dataset.

For RQ2, we evaluate the capability in UEware detection and classification. As there does not exist any publicly available dataset. we have to make efforts to build the **ground-truth dataset** to perform the evaluation, as follows:

- First, five experienced experts in our team collect wild apps from social media, forums, and websites. They work independently to download, install, and launch these apps, including registering accounts and investigating their usage processes.
- Second, these experts select apps that provide sensitive services (*i.e.*, Porn, Gambling, Financial in this work) for further analysis. According to different regulations in special fields (such as asset review in loan, etc.), they independently check whether the app

[6]The package names are: *org.woheller69.wather, site.leos.setter, ademar.bitac, de.digisocken.antherrss, net.gsantner.dandelior, com.github.dfa.diaspora_android, com.chao.app, com.kylecorry.trail_sense, org.woheller69.arity, de...osmplugin* (this name was truncated due to its excessive length).

Table 2: The dataset for evaluation.

Dataset	App Type	Setup Time	Detail	Number
Source-available Dataset	Self-developed	July, 2022	Self-developed Apps	5
	Real-world	2021-2022	Real-world Apps	10
	Total			15
Ground-truth Dataset	UEware	June, 2022	Gambling	310
			Porn	441
			Financial	97
	Normal Apps	June, 2022	-	852
	Total			1, 700
Large-scale Dataset	Wild	June-July, 2022	Apps from websites	13, 460
	Appstore	June-July, 2022	Apps from AppStore	10, 557
	Total			24, 017

Table 3: The evaluation of imprint generation.

Dataset Component	Token (#)	Identified (F1-score)
Self-Developed App	36	84.7%
Real-World App		
⊢# site.leos.setter	21	95.5%
⊢# de.digisocken.anotherrss	40	90.9%
⊢# org.woheller69.weather	43	90.5%
⊢# net.gsantner.dandelior	10	88.9%
⊢# com.github.dfa.diaspora_android	12	80.0%
⊢# com.chao.app	50	83.3%
⊢# de.storchp.opentracks.osmplugin	47	75.0%
⊢# com.kylecorry.trail_sense	9	84.2%
⊢# org.woheller69.arity	19	90%
⊢# ademar.bitac	8	85.7%
Total	295	85.4%

Table 4: The evaluation of UI transition determination. The boldfaced score denotes the best result.

Dataset Component	Transition		Identified (F1-score)		
	Type	#	Gator	IC3	DeUEDroid
Self-Developed App	All	62	17.6%	57.4%	**97.6%**
⊢# App1	Act-Act[1]	13	26.7%	96.0%	**96.3%**
⊢# App2	Act-Frag.	13	26.7%	76.2%	**92.3%**
⊢# App3	Frag.-Frag.	13	0%	0%	**100%**
⊢# App4	Navigation	12	0%	0%	**100%**
⊢# App5	All	11	30.8%	62.4%	**100%**
Real-World App					
⊢# site.leos.setter	All	5	-[2]	0%	**72.7%**
⊢# de.digisocken.anotherrss	All	4	42.9%	50%	**80.0%**
⊢# org.woheller69.weather	All	12	-	-	**91.7%**
⊢# net.gsantner.dandelior	All	4	50.0%	57.2%	**75.0%**
⊢# com.github.dfa.diaspora_android	All	5	47.0%	50.0%	**88.9%**
⊢# com.chao.app	All	12	-	-	**96.0%**
⊢# de.storchp.opentracks.osmplugin	All	10	-	53.3%	**90%**
⊢# com.kylecorry.trail_sense	All	6	33.3%	-	**83.3%**
⊢# org.woheller69.arity	All	5	52.8%	88.9%	**88.9%**
⊢# ademar.bitac	All	9	-	66.7%	**90%**
Total	All	134	24.6%	57.0%	**92.3%**

[1] Act. means Activity, and Frag. means Fragment.
[2] timed out within the 30 minute time limit.

usage processes comply with the required behavior patterns, and mark the non-compliant apps as UEware.

- Finally, all experts work together to make the final decision by majority voting. The ground-truth dataset consists of 1, 700 apps, including 310 gambling apps, 441 porn apps, 97 financial apps, and 852 normal apps.

We randomly select 70% of them as the training set, 20% as the validation set, and 10% as the test set.

For RQ3, we conduct a large-scale experiment to evaluate the performance of DeUEDroid to mitigate real-world threats. For this goal, we collect a large number of real-world apps and setup the **large-scale dataset**. Based on the source of acquisition, we divide these apps into two categories: AppStore apps and wild apps. The AppStore apps are collected from formal app markets, such as Mi Store, 360 Software Manager. In contrast, wild apps are collected from informal channels, such as social platforms or websites. In total, the large-scale dataset consists of 24, 017 apps (10, 557 AppStore apps and 13, 460 wild apps). Based on this dataset, we evaluate the performance (including the detection rate and the time consumption), and we also analyze the detection results.

In total, we setup three datasets (*i.e.*, source-available dataset, ground-truth dataset, and large-scale dataset, as shown in Table 2) to perform the evaluation. Note that all hardened apps that cannot be analyzed have been removed from our datasets.

8.2 Evaluation Result

8.2.1 For RQ1. App sketch construction has two sub-modules: *(i)* GUI widget identification, and *(ii)* UI transition determination.

First, for the GUI widget identification, DeUEDroid applies Frontmatter [37], a SOTA GUI analysis tool to identify the native widgets. So here we only evaluate the effectiveness of web widget identification, which is implemented by ourselves. And we list our evaluation result in Table 3. For the self-developed apps, the token number is clearly identified by our developers. In total, the self-developed apps have 36 tokens, such as IP host, and domain name. In the end, DeUEDroid identifies 36 tokens from the self-developed apps, with 3 false positive and 8 false negative (F1-score is 84.7%). For real-world apps, we first manually identify the tokens used by their web widgets. Specifically, we insert logging code (*i.e.*, logging.log()) in the source code after the network API, which does not affect the app function. Then we run the app to generate the log, and also inspect the source code to get the final tokens. The F1-score of real-world apps are range from 75.0% to 95.5%, which shows that our imprint generation has sufficient coverage.

Second, for the UI transition determination, we list our evaluation result in Table 4. Since the three components (*i.e.*, dialog, menu, activity) have been well studied in previous studies [37, 44, 50], we mainly focus on the new android development features (*i.e.*, fragment and navigation) in this work. Specifically, these self-developed apps are designed to be developed by certain transition types. The **App1** to **App4** consist of a single transition type (*e.g.*, activity-fragment) to evaluate the accuracy of different transition types, and **App5** is implemented by all transition types. From the evaluation results, it can be seen that the accuracy of Gator is low. This is because Gator does not support advanced SDK and does not account for fragment-related transitions and navigation-based transitions. Besides, IC3 can accurately identify the transition between Activity, but also have no ability to identify the fragment-related transitions. What's more, to the real-world apps, the Gator and IC3 are also ineffective (even the highest F1-score lower than 88.9%) and even can not get the result within 30 minutes. In contrast, DeUEDroid can identify all transition types (especially fragment-related transitions). To real-world apps, DeUEDroid is able to produce results within the specified time, and also receives high accuracy and even the lowest

F1-score is more than 72.7%. In total, DeUEDroid achieves high F1-score in transition identification (92.3%), which has significantly improvement from 35.5% to 67.7% than previous studies.

We further explain the evaluation results. First, since we take into account new transition types (*i.e.*, fragment-activity transition, and navigation-based transitions) besides activity transition, DeUEDroid can label all kinds of sensitive APIs and perform a more accurate algorithm to identify the transition pairs. It results in the high accuracy of UI transition determination, especially the fragment-related transition pairs. Second, since we set a maximum taint depth to ensure time efficiency, it leads to some omissions of transition identification. For example, we show a missing edge in the real-world app, the missing edge consists of a long call chain (*i.e.*, *onClick() - func1() - func2() - ... - func9() - actionStart() - startActivity()*) that beyond our maximum recursion.

Answer RQ1: DeUEDroid is effective in UTG construction that covers new UI features. Specifically, DeUEDroid can accurately identify imprint tokens (over 85% F1-score) and transition events (achieves 92.3% F1-score, a far better performance than the SOTA systems).

8.2.2 *For RQ2.* To answer RQ2, we evaluate the effectiveness of our system in UEware detection and classification. Specifically, we first evaluate the traditional signature-based detection methods (*i.e.*, malware detection, GUI content detection) on UEware. And then, we evaluate DeUEDroid and select five SOTA algorithms (*i.e.*, LR, GAT [57], GraphSAGE, GCN [24], GAE) as the baseline, and further compare with other SOTA machine learning based detection methods (*i.e.*, Drebin [16], MamaDroid [46]).

First, we evaluate the effectiveness of the signature-based detection methods. For malware detection methods, we randomly select 200 UEware and upload them to VirusTotal, the largest online malware detection platform with 63 security vendors, containing the vast majority of malicious code signatures. However, we find that only 34 apps have been labeled as malware, while the rest are considered benign or caused only one vendor warning. This phenomenon proves that *the malware detection methods are inefficient for UEware detection and have a high false negative rate (over 80%).* And then, we evaluate the GUI content detection methods. After decoding the APKs, we parse their local figures/texts and perform the manual review. Apart from the general icons (*e.g.*, button icon, and payment bank icon), there is almost no sensitive content (*e.g.*, porn picture, gambling picture) in local APKs (over 95%). To this end, we can only leverage their icons to perform the detection. Specifically, we use the seresnext50 network, a widely used network in the previous image learning studies [45, 51], to encode the icon and perform the detection. However, the F1-score in UEware detection is only 45.7%. After manual review, the main reason is that many UEware imitate the icons of normal ones, resulting in a high false positive rate of content-based detection. *The incomplete & disguised GUI content results in the failure of the content-based detection methods.* The above evaluations justify our motivation in Section 3.

Second, we evaluate our system capability on the ground-truth dataset. Our evaluation is based on two metrics: the detection capability and the classification capability. The detection capability can be regarded as a binary classification problem. In our dataset, the

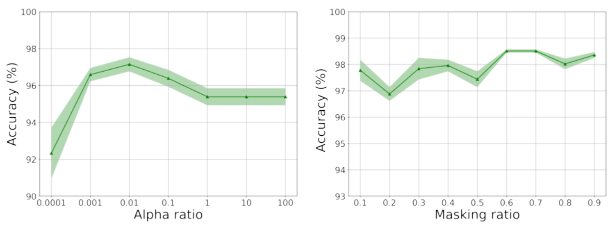

Figure 6: The effect of Alpha and mask ratio to DeUEDroid

number of normal apps is roughly equal to the number of UEware. We use F1-score to measure the detection performance. On the other hand, the classification capability is a multi-class classification problem. Since the distribution of the dataset may not be even, we use classification accuracy to measure the performance.

Specifically, we use the Manifest information (*i.e.*, permission, app size, package name, app components), and five SOTA algorithms as the baseline. We closely follow the linear evaluation scheme as previous studies [58] and report the detection F1-score and classification accuracy in Table 5.

- First, we compare the evaluation result between UTG (column#5-6) and baseline (column#3-4). The result shows that the UTG feature can improve the detection performance of all algorithms, with outstanding F1-score/accuracy improvement than Manifest across all algorithms (ranging from 5.68% to 14.19% in UEware detection and 9.24% to 17.62% in UEware classification). The outstanding improvement of the comparative experiments shows the UTG feature is universal for all algorithms and is effective in UEware detection and category classification.
- Second, we compare the evaluation result of the DeUEDroid algorithm, MaskGAE (line#7) with other SOTA algorithms (line#2-6). And our algorithm achieves leading performance among all algorithms, which exceeds all the supervised algorithms in UEware detection and classification, and even exceeds GAE 0.28% and 0.59% in F1-score and classification accuracy. This proves that our algorithm is effective, can better handle complex unstructured GUI attributes, and performs well on topological-attribute mixed information. Further, we show the Alpha and mask ratio effect of our detection results in Figure 6. From the alpha ratio effect in Figure 6, we can see that by increasing the alpha ratio from 0.0001 to 100, the accuracy first smoothly improves to 97.15% and declines then to 95.39%. The gap of ratio influence fluctuates very little, only around 4.82%. And the same as the masking ratio, the ratio influence gap is around 0.73%. The ratio effect indicates that our algorithm is very stable and insensitive to the influence of hyper-parameters, which can guarantee stable detection results.

In the end, DeUEDroid achieves 98.22% F1-score in UEware detection and 98.97% accuracy rate in UEware classification. To demonstrate the effectiveness, we also compare some SOTA machine learning based detection methods. Specifically, we compare our approach with two representative works, i.e., Drebin [16] and MamaDroid [46] [7]. Table 6 gives the result. Obviously, our approach

[7]Drebin and MamaDroid are implemented according to the previous studies [72], and we will also release the implementation on our github [7].

Table 5: The UEware detection F1-score (%) and UEware category classification accuracy (%) of different features and algorithms. In each column, the boldfaced score denotes the best result.

	Algorithm	Manifest		UTG	
		Detection F1-score	Classification Accuracy	Detection F1-score	Classification Accuracy
Supervised	LR	$76.81_{\pm0.33}$	$79.41_{\pm0.92}$	$91.00_{\pm0.45}$	$88.65_{\pm0.76}$
	GAT	$79.40_{\pm0.43}$	$82.56_{\pm0.33}$	$92.17_{\pm0.05}$	$93.64_{\pm1.75}$
	GraphSAGE	$91.00_{\pm0.00}$	$81.18_{\pm0.32}$	$97.83_{\pm0.12}$	$93.66_{\pm1.10}$
	GCN	$89.24_{\pm0.37}$	$81.89_{\pm0.37}$	$96.76_{\pm0.24}$	$93.73_{\pm1.36}$
Self-supervised	GAE	$92.29_{\pm0.18}$	$80.76_{\pm1.14}$	$97.94_{\pm0.07}$	$98.38_{\pm0.41}$
	MaskGAE (Ours)	$\mathbf{92.54}_{\pm0.21}$	$\mathbf{82.61}_{\pm1.52}$	$\mathbf{98.22}_{\pm0.06}$	$\mathbf{98.97}_{\pm0.22}$

Table 6: The comparison with SOTA AI-based malware detection methods. The results contain the detection F1-score and classification accuracy.

Method	Feature	Algorithm	Result	
			F1-score	Accuracy
Drebin	Manifest & API [1]	DNN	84.59%	79.00%
MamdDroid	API Markov Chain	RF [2]	96.94%	98.32%
DeUEDroid	UTG	MaskGAE	98.22%	98.97%

[1] Drebin uses the Manifest feature and some chosen APIs as composite features.
[2] Random Forest.

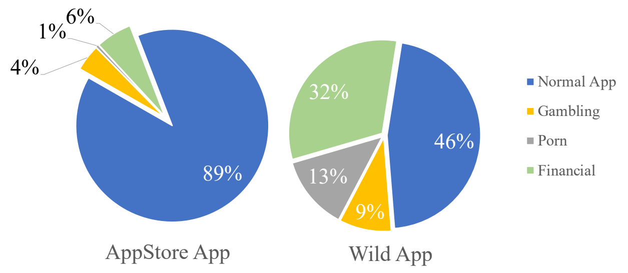

Figure 7: Detection results on the large-scale dataset.

achieves better performance in both binary detection and multi-class classification tasks.

Specifically, Drebin relies on features such as Manifest and sensitive API, which makes the UEware detection ineffective due to the essential limitation (see Section 3). While MamaDroid uses the global call graph to perform the detection. The number of API calls within the graph grows over time [46], which makes the entire API graphs of apps become a heavy-weight feature. As a result, MamaDroid's model would be extensively disturbed to affect the time consumption (more than half of the apps in our dataset take over 10 minutes).

Answer RQ2: DeUEDroid demonstrates effectiveness in UEware detection and classification, achieving a 98.22% detection F1-score and a 98.97% classification accuracy, outperforming traditional approaches. Besides, its algorithm surpasses SOTA algorithms and exhibits insensitivity to hyper-parameters.

8.2.3 For RQ3. We apply DeUEDroid on a large-scale dataset to evaluate its real-world performance, focusing on detection results, time efficiency, and UTG measurement.

Large-scale Detection Result. The detection results on the large-scale dataset are remarkable, as illustrated in Figure 7. First, we analyze the detection results of wild apps, finding that **more than half (54%) of the wild apps are UEware** (9% gambling, 13% porn, and 32% financial), while the remaining 46% are normals. Next, we examine the detection results for AppStore apps and surprisingly discover that **11% of AppStore apps are identified as UEware** (4% gambling, 1% porn, and 6% financial).

To further confirm the detection result, we additionally select detected UEware from the two dataset components for manual review. In total, we review a total of 400 apps, and 356 apps are consistent with the manual review results. Since the Out-of-distribution (OOD) of the dataset, the accuracy in the large-scale dataset is a bit lower than the ground-truth dataset. In conclusion, the detection result shows that DeUEDroid is effective for large-scale UEware detection tasks. And the UEware are extremely prevalent in the wild and Appstore. Even though the app stores have performed checks on their apps, we still find that there are some undetected UEware (11% of all apps) in AppStore.

Performance. In addition, we evaluate the time consumption of our system. Overall, each app costs about 213 seconds on average, ranging from 9 seconds to 2, 347 seconds. We show the average time consumption of different app sizes in Figure **??** (in Appendix B). The DeUEDroid time consumption is stable, where the fluctuation of time is only 67s in the range 15M-57M. The performance results show that DeUEDroid is large-scale resilient. And we further explain that our time consumption is much less than that of the dynamic detection method. Through our preliminary experiments, it takes more than 30 minutes for dynamic execution techniques to traverse an app. For large-scale UEware detection, our approach is more lightweight.

Measurement. We make a measurement of the UTG, including the number of transition pairs, the widgets, and the imprint tokens.

The investigation shows that the UTGs of UEware and normal apps are significantly different in statistics. Specifically, the UEware are the detected ones from the large-scale dataset, while the normal apps are the remaining ones. First, the average transition pairs in normal apps are twice as many as UEware (22 vs. 11). Second, the average number of widgets owned by normal apps is much higher than that of UEware (227 vs. 29). Finally, turning to the network imprint. The average number of tokens owned by normal apps is also higher than that of UEware (211 vs. 121). The statistical result can obviously proves our observation-I in Section 3, that UEware are always single-purpose and different from normal apps.

Answer RQ3: DeUEDroid is large-size resilient and effective in large-scale detection. The result suggests that 54% apps in the wild and 11% apps in the AppStore are UEware.

9 DISCUSSION

Regional Differences and Commonalities. The specific manifestations of UEware are closely linked to local laws and regulations. Although different regions/countries may have different regulations, our key observation is that there do exist differences between the non-compliant apps and the compliant ones (i.e., the normal apps) in a specific region [23, 32, 48]. This suggests that our detection approach can be applied to other regions, though our datasets are collected from Asia. Based on our approach, different regions can establish their own datasets to accommodate their regulations.
App Sketch Construction. As UTG is built using static analysis, apps may leverage app hardening techniques to disrupt static analysis. In addition, even though the static analysis considers the major factors introduced by Android (*i.e.*, multi-threading, lifecycle, and ICCs), apps may evade our analysis by invoking sensitive APIs via reflections and native libraries. On the other side, UTG uses network imprints to identify the network features. While the network imprints are highly malleable, they are still inaccurate compared to real remote resources. In the future, we plan to incorporate dynamic analysis to deal with the app hardening techniques and capture real network remote resources.
Anti Adversary. Our model is trained by the ground-truth UEware dataset. To avoid our detection, adversaries may download our dataset and find patterns that are not covered. For example, they can use specially designed app sketches to mislead our detection model. Since the dataset is collected manually by our team, the number of samples is limited which cannot prevent attacks against the dataset. However, we claim that enterprises can supplement the dataset by themselves to improve the coverage, thereby improving the accuracy and avoiding attacks on the model coverage. In addition, UEware classification categories can also be added according to the needs of enterprises.

10 RELATED WORK

Android UI Analysis. There are many studies focusing on Android UI analysis. Azim et al. [18] presented the activity transition graph to model the Android UI transition. Liu et al. [44] combine machine learning and static analysis to complete the transition pairs. Chen et al. [21] took into account the inner class to complete the transition pairs. Yang et al. [65, 67] leveraged context-sensitive analysis of callback methods and developed a client analysis that builds a static GUI model and window transition graph. PERUIM [41] focused on the permissions used behind the widgets and connected the widgets with their handlers. Gator [52, 64, 66] modeled the WTG and the widget attributes by windows stack. Frontmatter [37] implemented a lightweight static widget analysis tool, and linked the callback with buttons. Our app sketch is built upon these static UI analysis techniques, and additionally adds new features (*i.e.*, WebView, transition about the fragments, and navigation).
Machine-Learning Based Detection Machine learning is widely used for anomaly detection. Previous studies [15, 36, 38, 70] leveraged the feature of API control flow and permissions to perform malware detection, since the malicious code is different from the benign code. Besides, some studies leveraged anomaly pictures/texts and code to perform detection. AppIntent [68] used the taint analysis from the UIs to analyze the location that may cause sensitive data leakage. AsDroid [35] compares the GUI attribution and code behavior to find out the stealthy behavior against the normal GUI attribution. IconIntent [63] leverages machine learning to automatically identify sensitive icon behaviors. DeepIntent [62] leverages machine learning to compare the difference between UI and code behavior to analyze the misusing icon. SUPOR [34] identified the privacy input based on the text, figures, and layout by machine learning. In this study, we refer to the self-supervised algorithm and propose a novel algorithm to extract the UTG features and identify the UEware.

11 CONCLUSION

In recent years, the proliferation of UEware has become an emerging threat. This has seriously impacted the security of the mobile ecosystem and resulted in significant financial losses. In this paper, we propose a novel UTG based approach to effectively and efficiently detect UEware. Specifically, it first statically builds precise UTG for UEware, and then applies a graph-embedding based method to represent UTG by properly correlating multi-dimensional attributes. After that, it leverages the self-supervised learning method to detect and classify UEware based on the UTG feature similarity. We have implemented a prototype system named DeUEDroid to perform the evaluation. The evaluation results show that DeUEDroid is efficient and effective. It can detect UEware more efficiently than the traditional approaches (with 98.22% detection F1-score and 98.97% classification accuracy). Furthermore, DeUEDroid is large-size resilient and efficient for large-scale detection in the real world. By using DeUEDroid, we found that UEware are prevalent, *i.e.*, 54% apps in the wild and 11% apps in the app stores are UEware. Our system could be used by authoritative agencies and app download platforms to effectively mitigate this threat while alleviating the human efforts required.

DATA AVAILABILITY STATEMENT

The prototype of the proposed DeUEDroid system [14] is available from the corresponding author upon reasonable request.

ACKNOWLEDGMENTS

We would like to thank all anonymous reviewers for their helpful suggestions and comments to improve the paper. This work was supported by the National Key R&D Program of China (No. 2022YFE0113200), the National Natural Science Foundation of China (No. U21A20464, 62172360, U21A20467, 62072395, and U20A20178), and the National Key Research and Development Program of China (No. 2020AAA0107705). The findings herein reflect the work and are solely the responsibility of the authors.

REFERENCES

[1] 2021. BBC Report about romance scammer. https://edition.cnn.com/2021/02/21/us/losses-to-romance-scams-trnd/index.html. Accessed January 1, 2021.
[2] 2021. China Gambling Report. http://english.www.gov.cn/statecouncil/ministries/202101/06/content_WS5ff570dac6d0f7257694358b.html. Accessed January 1, 2021.

[3] 2021. F-droid. https://f-droid.org/. Accessed November, 2021.
[4] 2021. Google Activity. https://developer.android.com/guide/components/activities/intro-activities. Accessed November 23, 2021.
[5] 2021. USA Gambling. https://www.baltictimes.com/usa_online_gambling_laws___legal_states_in_usa_2022/. Accessed November, 2021.
[6] 2022. Apps for economy. https://42matters.com/blog/?p=the-state-of-the-app-economy-and-app-markets. Accessed June 4, 2022.
[7] 2022. DeUEDroid project website. https://github.com/HypoopyH/DeUEDroid. Accessed November 20, 2022.
[8] 2022. Fragment Navigation. https://developer.android.com/guide/navigation/navigation-getting-started. Accessed June 4, 2022.
[9] 2022. Google Play. https://support.google.com/googlepay/answer/10223857?hl=en. Accessed November 08, 2022.
[10] 2022. Hybrid App Percentage in Appstore. https://venturebeat.com/2020/11/23/why-74-of-the-top-50-retail-apps-are-hybrid-apps-not-native-apps/. Accessed August 8, 2022.
[11] 2022. India loan scam. https://www.bbc.com/news/business-61564038. Accessed June 4, 2022.
[12] 2022. State of Mobile in 2022. https://www.data.ai/en/go/state-of-mobile-2022. Accessed June 8, 2022.
[13] 2022. US Scam App Report. https://www.straitstimes.com/asia/se-asia/malaysia-govt-loses-s1bil-revenue-a-year-from-illegal-gaming-syndicates. Accessed June 6, 2022.
[14] 2023. DeUEDroid System. https://zenodo.org/record/7962231. https://doi.org/10.5281/zenodo.7962231 Accessed May 08, 2023.
[15] Faraz Ahmed, Haider Hameed, M Zubair Shafiq, and Muddassar Farooq. 2009. Using spatio-temporal information in API calls with machine learning algorithms for malware detection. In *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*. 55–62.
[16] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *Ndss*, Vol. 14. 23–26.
[17] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.
[18] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 641–660.
[19] Farnaz Behrang and Alessandro Orso. 2019. Test migration between mobile apps with similar functionality. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 54–65.
[20] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the {Google-Play} Scale. In *24th USENIX Security Symposium (USENIX Security 15)*. 659–674.
[21] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. Storydroid: Automated generation of storyboard for Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 596–607.
[22] Yi Chen, Wei You, Yeonjoon Lee, Kai Chen, XiaoFeng Wang, and Wei Zou. 2017. Mass discovery of android traffic imprints through instantiated partial execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 815–828.
[23] Zhuo Chen, Lei Wu, Jing Cheng, Yubo Hu, Yajin Zhou, Zhushou Tang, Yexuan Chen, Jinku Li, and Kui Ren. 2021. Lifting The Grey Curtain: A First Look at the Ecosystem of CULPRITWARE. *arXiv preprint arXiv:2106.05756* (2021).
[24] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 257–266.
[25] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Proc. 10th International Static Analysis Symposium (SAS) (LNCS, Vol. 2694)*. Springer-Verlag, 1–18. Available from `http://www.brics.dk/JSA/`.
[26] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. 2018. Frauddroid: Automated ad fraud detection for android apps. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 257–268.
[27] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*. 235–245.
[28] Parvez Faruki, Vijay Laxmi, Ammar Bharmal, Manoj Singh Gaur, and Vijay Ganmoor. 2015. AndroSimilar: Robust signature for detecting variants of Android malware. *Journal of Information Security and Applications* 22 (2015), 66–80.
[29] Yuhao Gao, Haoyu Wang, Li Li, Xiapu Luo, Guoai Xu, and Xuanzhe Liu. 2021. Demystifying illegal mobile gambling apps. In *Proceedings of the Web Conference 2021*. 1447–1458.
[30] Kaveh Hassani and Amir Hosein Khasahmadi. 2020. Contrastive multi-view representation learning on graphs. In *International Conference on Machine Learning*. PMLR, 4116–4126.
[31] Geoffrey E Hinton and Richard Zemel. 1993. Autoencoders, minimum description length and Helmholtz free energy. *Advances in neural information processing systems* 6 (1993).
[32] Geng Hong, Zhemin Yang, Sen Yang, Xiaojing Liaoy, Xiaolin Du, Min Yang, and Haixin Duan. 2022. Analyzing Ground-Truth Data of Mobile Gambling Scams. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2176–2193. https://doi.org/10.1109/SP46214.2022.9833665
[33] Yangyu Hu, Haoyu Wang, Yajin Zhou, Yao Guo, Li Li, Bingxuan Luo, and Fangren Xu. 2019. Dating with scambots: Understanding the ecosystem of fraudulent dating applications. *IEEE Transactions on Dependable and Secure Computing* (2019).
[34] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. {SUPOR}: Precise and scalable sensitive user input detection for android apps. In *24th USENIX Security Symposium (USENIX Security 15)*. 977–992.
[35] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*. 1036–1046.
[36] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im. 2018. A multimodal deep learning method for android malware detection using various features. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018), 773–788.
[37] Konstantin Kuznetsov, Chen Fu, Song Gao, David N Jansen, Lijun Zhang, and Andreas Zeller. 2021. What do all these Buttons do? Statically Mining Android User Interfaces at Scale. *arXiv preprint arXiv:2105.03144* (2021).
[38] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-An, and Heng Ye. 2018. Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics* 14, 7 (2018), 3216–3225.
[39] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2014. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. *arXiv preprint arXiv:1404.7431* (2014).
[40] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2017. Simidroid: Identifying and explaining similarities in android apps. In *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 136–143.
[41] Yuanchun Li, Yao Guo, and Xiangqun Chen. 2016. Peruim: Understanding mobile application privacy with permission-ui mapping. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 682–693.
[42] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. 2014. {DECAF}: Detecting and characterizing ad fraud in mobile apps. In *11th USENIX symposium on networked systems design and implementation (NSDI 14)*. 57–70.
[43] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 872–881.
[44] Changlin Liu, Hanlin Wang, Tianming Liu, Diandian Gu, Yun Ma, Haoyu Wang, and Xusheng Xiao. 2022. ProMal: precise window transition graphs for android via synergy of program analysis and machine learning. In *Proceedings of the 44th International Conference on Software Engineering*. 1755–1767.
[45] Amirreza Mahbod, Gerald Schaefer, Chunliang Wang, Georg Dorffner, Rupert Ecker, and Isabella Ellinger. 2020. Transfer learning using a multi-scale and multi-network ensemble for skin lesion classification. *Computer methods and programs in biomedicine* 193 (2020), 105475.
[46] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2016. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433* (2016).
[47] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[48] Collins W Munyendo, Yasemin Acar, and Adam J Aviv. 2022. "Desperate Times Call for Desperate Measures": User Concerns with Mobile Loan Apps in Kenya. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1521–1521.
[49] Nicole L Muscanell, Rosanna E Guadagno, and Shannon Murphy. 2014. Weapons of influence misused: A social influence analysis of why people fall prey to internet scams. *Social and Personality Psychology Compass* 8, 7 (2014), 388–396.
[50] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 77–88.

[51] Sanghyuk Park, Minchul Shin, Sungho Ham, Seungkwon Choe, and Yoohoon Kang. 2019. Study on fashion image retrieval methods for efficient fashion visual search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 0–0.

[52] Atanas Rountev and Dacong Yan. 2014. Static reference analysis for GUI objects in Android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 143–153.

[53] Silvia Sebastian and Juan Caballero. 2020. Towards attribution in mobile markets: Identifying developer account polymorphism. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 771–785.

[54] Mingshen Sun, Mengmeng Li, and John CS Lui. 2015. DroidEagle: Seamless detection of visually similar Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. 1–12.

[55] Yutian Tang, Yulei Sui, Haoyu Wang, Xiapu Luo, Hao Zhou, and Zhou Xu. 2020. All your app links are belong to us: understanding the threats of instant apps based attacks. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 914–926.

[56] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java bytecode using the Soot framework: Is it feasible?. In *International conference on compiler construction*. Springer, 18–34.

[57] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[58] Petar Velickovic, William Fedus, William L Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. 2019. Deep Graph Infomax. *ICLR (Poster)* 2, 3 (2019), 4.

[59] Michelle Y Wong and David Lie. 2016. Intellidroid: a targeted input generator for the dynamic analysis of android malware.. In *NDSS*, Vol. 16. 21–24.

[60] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. *2012 Seventh Asia Joint Conference on Information Security* (2012), 62–69.

[61] Haowei Wu, Yan Wang, and Atanas Rountev. 2018. Sentinel: generating GUI tests for Android sensor leaks. In *2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST)*. IEEE, 27–33.

[62] Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, et al. 2019. DeepIntent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2436.

[63] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. 2019. Iconintent: automatic identification of sensitive ui widgets based on icon classification for android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 257–268.

[64] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25, 4 (2018), 833–873.

[65] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *International Conference on Software Engineering*. 89–99.

[66] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. 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, 89–99.

[67] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android. In *IEEE/ACM International Conference on Automated Software Engineering*. 658–668.

[68] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. 2013. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1043–1054.

[69] Kan Yuan, Di Tang, Xiaojing Liao, XiaoFeng Wang, Xuan Feng, Yi Chen, Menghan Sun, Haoran Lu, and Kehuan Zhang. 2019. Stealthy porn: Understanding real-world adversarial images for illicit online promotion. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 952–966.

[70] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-sec: deep learning in android malware detection. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 371–372.

[71] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. 25–36.

[72] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. 2020. 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*. 757–770.

[73] Yutao Zhang, Fanjin Zhang, Peiran Yao, and Jie Tang. 2018. Name Disambiguation in AMiner: Clustering, Maintenance, and Human in the Loop.. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 1002–1011.

Received 2023-02-16; accepted 2023-05-03