TACDROID: Detection of Illicit Apps through Hybrid Analysis of UI-based Transition Graphs

Yanchen Lu*, Hongyu Lin*, Zehua He*, Haitao Xu*, Zhao Li†, Shuai Hao‡, Liu Wang§, Haoyu Wang¶, Kui Ren*
*Zhejiang University, Hangzhou, China
†Hangzhou Yugu Technology, Hangzhou, China
‡Old Dominion University, VA, USA
§Beijing University of Posts and Telecommunications, Beijing, China
¶Huazhong University of Science and Technology, Wuhan, China
*{22221066,22321068,3200103719,haitaoxu,kuiren}@zju.edu.cn,†lzjoey@gmail.com,‡shao@odu.edu,§w_liu@bupt.edu.cn,¶haoyuwang@hust.edu.cn

***Abstract*—Illicit apps have emerged as a thriving underground industry, driven by their substantial profitability. These apps either offer users restricted services (*e.g.*, porn and gambling) or engage in fraudulent activities like scams. Despite the widespread presence of illicit apps, scant attention has been directed towards this issue, with several existing detection methods predominantly relying on static analysis alone. However, given the burgeoning trend wherein an increasing number of mobile apps achieve their core functionality through dynamic resource loading, depending solely on static analysis proves inadequate.**

To address this challenge, in this paper, we introduce TACDROID, a novel approach that integrates dynamic analysis for dynamic content retrieval with static analysis to mitigate the limitations inherent in both methods, *i.e.*, the low coverage of dynamic analysis and the low accuracy of static analysis. Specifically, TACDROID conducts both dynamic and static analyses on an Android app to construct dynamic and static User Interface Transition Graphs (UTGs), respectively. These two UTGs are then correlated to create an intermediate UTG. Subsequently, TACDROID embeds graph structure and utilizes an enhanced Graph Autoencoder (GAE) model to predict transitions between nodes. Through link prediction, TACDROID effectively eliminates false positive transition edges stemming from misjudgments in static analysis and supplements false negative transition edges overlooked in the intermediate UTG, thereby generating a comprehensive and accurate UTG. Finally, TACDROID determines the legitimacy of an app and identifies its category based on the app's UTG. Our evaluation results highlight the outstanding accuracy of TACDROID in detecting illicit apps. It significantly surpasses the state-of-the-art work, achieving an F1-score of 96.73%. This work represents a notable advancement in the identification and categorization of illicit apps.

***Index Terms*—Illicit app detection, UI transition graph**

I. INTRODUCTION

Mobile apps have become integral to various aspects of our lives, encompassing activities like shopping, communication, and travel. Unfortunately, the underground economic ecosystem that exploits mobile apps to provide illicit services has been rapidly expanding. Notably, in China, over 60% of telecommunication fraud incidents are carried out through apps [1]. Similarly, in Singapore, fraud cases involving gambling apps surged by over 18 times from 2019 to 2020 [2]. Mobile applications engaged in activities that breach ethical or legal standards are commonly referred to as *illicit apps*. Recognizing the diverse interpretations of what constitutes illicit across different countries or regions, this study concentrates on three primary categories: porn, gambling, and scam.

Most countries and app stores have strict restrictions on the development and distribution of illicit apps. Taking major app stores like Google Play Store and Apple App Store as examples, they do not accept porn and scam apps, and gambling apps can only be accepted after rigorous approval [3, 4]. However, driven by economic interests, a significant number of unregulated illicit apps persist in spreading widely through channels such as SMS and instant messaging software [5, 6]. Consequently, there is an urgent demand for app stores and regulatory bodies to efficiently detect and prevent the proliferation of illicit apps.

Existing techniques for detecting Android malware face challenges in seamlessly transitioning to the identification of illicit apps. Unlike typical malware, illicit apps often operate without malicious payloads, primarily relying on content-based deception to achieve their goals. Current research on illicit apps predominantly focuses on characterizing their ecosystem and app features [5–9]. To our knowledge, the work by Chen *et al.* [10] stands out for proposing DeUEDroid, the first system designed specifically for identifying illicit apps. DeUEDroid utilizes static User Interface Transition Graphs (UTGs) to detect these apps by capturing differences in content presentation, effectively distinguishing between illicit and legitimate apps based on semantic variations in UI. However, the challenge arises when handling illicit apps that leverage WebView [11], a popular Android system component enabling apps to display dynamic web content. The static UTG-based approach falls short in such cases.

WebView is frequently utilized in illicit apps compared to legitimate ones. A recent study [12] shows that 57% of apps on Google Play use WebView, while our research suggests that this figure increases to over 87% among illicit apps. Furthermore, WebView is a critical component for many illicit apps to fulfill their core functions, and one extreme case is that a feature-rich illicit app contains 6 native classes only,

with all other functionalities implemented through WebView. Consequently, constructing the UTG through static analysis alone is inadequate when an app employs WebView.

Dynamic analysis can alleviate this problem by actually running the app, but it also comes with inherent limitations. One significant constraint is the potential generation of incomplete UTGs due to insufficient coverage. Moreover, its effectiveness can be compromised by factors like unstable internet connections.

To tackle these challenges, we introduce TACDROID, a novel approach that integrates dynamic analysis for WebView content retrieval with static analysis to address the limitations inherent in both methods, *i.e.*, dynamic analysis's low coverage and static analysis's low accuracy. Specifically, TACDROID performs both dynamic and static analyses on an Android app to construct dynamic and static UTGs, respectively. These two UTGs are then correlated to create a Seed UTG. Subsequently, TACDROID embeds graph structure and utilizes an enhanced Graph Autoencoder (GAE) model to predict transitions between nodes. Through link prediction, TACDROID effectively eliminates false positive transition edges resulting from misjudgments in static analysis and supplements false negative transition edges overlooked in the Seed UTG, thereby generating a comprehensive and accurate UTG. Finally, TACDROID determines the legitimacy of an app and identifies its category based on the app's UTG and metadata.

We implemented TACDROID, evaluated its effectiveness in UTG construction, and compared it to 6 other baselines. On a dataset consisting of 698 nodes and 1,975 edges, TACDROID achieved an F1-score of 83.48%, outperforming the best baseline by 8.45%. To evaluate the effectiveness of illicit app detection, we conducted a large-scale experiment using the most extensive ground-truth dataset of illicit apps available to date. TACDROID achieved F1-scores of 96.73% for illicit app detection and 92.09% for classification, significantly outperforming the state-of-the-art (SOTA) illicit detection method, DeUEDroid, as well as traditional methods.

The paper makes the following key contributions:

- Our investigation uncovers the widespread utilization of WebView in illicit apps and underscores the constraints inherent in exclusively depending on static analysis for UTG construction.
- We introduce an effective method for UTG construction that harnesses both dynamic and static analyses alongside link prediction techniques, demonstrating that this integrated approach yields a more comprehensive UTG compared to individual methods.
- We implement an illicit app detection system, TACDROID, which outperforms existing SOTA solutions for illicit app detection.
- We compile the most extensive ground-truth dataset for illicit apps, comprising 8,618 apps. Our system and the dataset are available on GitHub [13].

II. BACKGROUND

Activity is an essential component of the Android system responsible for rendering and displaying the user interface (UI). An Android UI is typically represented as a hierarchical tree structure, *i.e.*, a UI tree. In this tree, each node represents a UI widget (*e.g.*, Button, ImageView, LinearLayout). UI widgets can register event listeners with the Android framework, each equipped with callback methods to respond to user actions. For example, when a user clicks a button, the Android framework invokes the associated callback method, *e.g.*, `onClick`, for that component. Developers can use functions like `startActivity` within `onClick` to initiate another activity, triggering a UI transition. Additionally, hardware events like pressing the HOME button can also lead to UI transitions. As existing work [14–16] did, we define a UTG as

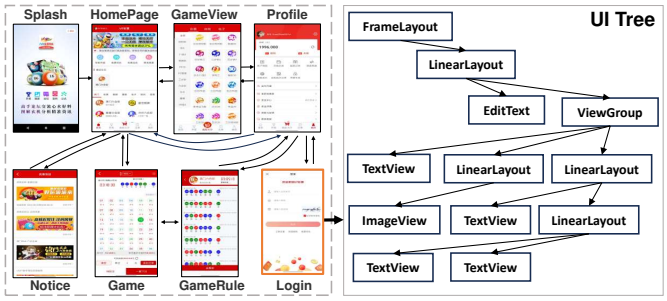

Fig. 1: Partial UTG of a gambling app

a directed graph $U < N, E >$, where a node $n \in N$ represents a UI element, and an edge $e \in E$ represents a UI transition. Figure 1 provides examples of a UTG and a UI tree.

Node. We refer to the notion of UI state in PermDroid [17] and GoalExplorer [18] and define a UI node as being composed of an activity and three categories of potential components: *fragment*, *dialog* and *menu*. Fragment is an independent UI component that can be embedded in an activity and has its own lifecycle. Menus and dialogs are short-lived windows that can trigger corresponding events based on different user selections or inputs.

Transition. A transition represents a link from a source node N_s to a target node N_t, denoted as $E < N_s, N_t >$. Each transition is associated with a label e that signifies the event that triggers the transition. Events are typically categorized into UI events (*e.g.*, `click`, `touch`, and `scroll` events) and hardware events (*e.g.*, pressing the `BACK` or `POWER` button).

III. DATASET

We established a collaboration with a prominent Internet Service Provider (ISP) in China, obtaining a ground truth dataset of illicit apps compiled between March 2023 and May 2023. This dataset was generated as the ISP identified potential illicit app download links from SMS based on user feedback and inputs from authoritative agencies. The process involved downloading the associated apps and curating the illicit app dataset through manual assessment. Furthermore,

we conducted an independent collection and categorization of a separate dataset containing both legitimate and illicit apps, employing the following two methods:

Illicit App Collection. Illicit apps are commonly promoted through various channels, including Telegram groups [19] and online social media platforms. Therefore, we conducted searches on several popular platforms, including Telegram, Google, and Twitter, for websites containing content related to porn, gambling, and scams. A significant number of these websites also host more than one illicit app. Given the considerable variability in website layouts and the necessity for logging in to download apps, we opted to manually visit each of these websites and download potential illicit apps, as opposed to adopting an automated collection method, which would likely result in missing a substantial number of apps. For websites, particularly those related to porn and gambling, that link to other similar category websites, we performed recursive browsing of these webpages and downloaded the apps hosted on them to expand our dataset as much as possible.

We dedicated considerable effort to the aforementioned process from September 2022 to June 2023, ultimately obtaining 2,927 potential illicit applications from 2,239 websites. Subsequently, we referred to Apple's App Store Review Guidelines [4] to formulate codebooks for three categories of illicit apps. Specifically, our definitions of these categories are as follows:

- **Porn App**: Applications containing explicit descriptions or displays of sexual organs or activities, aimed at provoking erotic rather than aesthetic or emotional responses, including "hookup" apps, those containing pornography, or those potentially involved in facilitating prostitution, human trafficking, or exploitation.
- **Gambling App**: Applications that enable or mimic gambling involving monetary transactions, including casino games, sports betting, and lottery competitions.
- **Scam App**: Applications designed to deceive users through false statements, hidden charges, exaggerated claims, or manipulation to divulge personal information. This encompasses phishing attempts, fraudulent transactions, counterfeit services, or products.

Three experienced researchers conducted independent examinations and categorizations of all the apps, endeavoring to reconcile deviation. Ultimately, we marked a total of 2,566 illicit apps, alongside 361 apps for which the types could not be definitively determined. Inspired by prior works [5, 20], to determine the inter-rater reliability of this process, we computed Krippendorff's alpha coefficient [21], a statistical measure of the extent of agreement among raters, which is regularly used by researchers in the field of content analysis. The computed value of 0.87 demonstrates the high reliability of our labeling of illicit apps.

Legitimate App Collection. We conduct bulk downloads of apps from two primary sources: `Google Play` and `AppChina` (a prominent application store in China) [22]. These apps are selected based on their high rankings and positive user reviews, indicating their high probability of being legitimate. For Google Play, we select those apps with more than 100 reviews with four and five stars. As for AppChina, we focus on apps that have a favorable rating higher than 60%, with more than 200 reviewers. Furthermore, We refine this dataset by selecting 60% of legitimate apps from specific categories such as gaming, dating, and finance, which offer services akin to those provided by illicit apps. We verify the legitimacy of these apps by subjecting them to scans on VirusTotal [23] to ensure they are free from malicious code[1]. The apps through the above process are labeled as legitimate.

TABLE I: Statistics of our ground truth dataset

App Source	Type of Apps				Total
	Porn	Gambling	Scam	Legitimate	
ISP-provided	80	527	2,014	–	2,621
Self-collected	1,456	963	147	3,431	5,997
Total	1,536	1,490	2,161	3,431	8,618

Dataset Statistics. We define the app's ID as the concatenation of its app name and package name. Following the removal of duplicate apps, we finalize our dataset. Table I presents the statistics of our ground truth dataset. The dataset comprises a total of 8,618 apps, including 1,536 porn apps, 1,490 gambling apps, 2,161 scam apps, and 3,431 legitimate apps. Notably, in scenarios where there are no financial transactions associated with the apps, determining an application as a scam app can pose challenges. To ensure the credibility of the dataset, we specifically categorize only a limited number of apps as scam apps, such as counterfeit banking apps. The dataset provided by the ISP proves to be valuable in supplementing this deficiency. As far as we know, this ground truth dataset is the largest available, several times larger than the dataset collected in a recent work, DeUEDroid [10].

IV. MOTIVATION

Conventional Android malware detection methods typically select features based on malware behavior (*e.g.*, Permissions, API, Control Flow Graph (CFG)). However, these features often underperform in detecting illicit apps, as the latter generally do not directly harm the device. For instance, considering the widely chosen feature, Permissions, in malware detection, we surprisingly find that illicit apps, on average, request a lower number of dangerous permissions compared to legitimate apps. While the recent work DeUEDroid utilized static UTGs to detect illicit apps and achieved notable results by leveraging content differences between apps, it overlooked the significant prevalence of hybrid apps[2] within the illicit application domain. In this section, we first present a preliminary study of the prevalence of WebView within illicit apps. Furthermore, we provide a motivation example to illustrate

[1]We randomly sampled 200 instances from the dataset. Following a manual content review, all samples were confirmed to be legitimate apps. This underscores the dependable quality of the legitimate dataset.

[2]Hybrid apps are mobile applications that use WebView to display web content within a native app framework.

the insufficiency of exclusively relying on static analysis for detection and the efficacy of our approach, TACDROID.

A. Prevalence of WebView Usage among Illicit Apps

WebView is a component that enables the display of web content within an activity's layout. We examined the prevalence of WebView utilization in illicit apps.

We utilized FlowDroid [24] to construct a call graph for the app using its default construction algorithm, SPARK. Subsequently, we traversed the call graph to identify whether any callees are among the methods of the `android.webkit.WebView` class. If such a method is found, we conclude that the app employs WebView.

Furthermore, we investigate the number of activities that utilize WebView in an app to measure the extent of WebView usage. Specifically, we first identify the activities actually displayed in the app by examining the class names and parent classes of callers in the call graph, as an app may declare many activities, some of which may never be displayed to a user. Then, we determine which activities utilize WebView based on the classes that invoke WebView-related methods. The higher the proportion of activities utilizing WebView among the activities actually displayed, the more extensively WebView is utilized in the app.

TABLE II: The prevalence of WebView usage both at the app level and the activity level by app category

App Type	Apps*	%Apps w/ Webview	%Activities w/ Webview (Avg.)
Porn	1,202	914 (**76.04%**)	65.73%
Gambling	1,136	605 (**53.26%**)	44.72%
Scam	2,021	1,701 (**84.15%**)	64.51%
Legitimate	2,653	1,480 (**55.79%**)	20.96%

**Apps* denotes # apps that we can construct call graph successfully.

Table II displays the percentage of apps utilizing WebView, along with the percentage of activities within those apps that incorporate WebView. The results indicate that WebView is extensively utilized in illicit applications. Specifically, at the app level, WebView is employed in 76.04% of porn apps, 53.26% of gambling apps, and 84.15% of scam apps. Moreover, at the activity level, among the apps using WebView, approximately 65.73% of activities in porn apps, 44.72% in gambling apps, and 64.51% in scam apps incorporate WebView elements or content. In contrast, although a higher percentage of legitimate apps use WebView (55.79%) compared to gambling apps, only 20.96% of activities in legitimate apps involve WebView. This is significantly lower than in illicit apps.

In summary, WebView demonstrates a notably higher prevalence within illicit apps. This prevalence may be attributed to that illicit apps face the imminent risk of shutdown by regulatory authorities. Leveraging WebView for illicit app development enables the reuse of existing web services and thereby mitigates development costs [25].

B. Motivation Example

Corresponding to the widespread utilization of WebView in illicit apps is the phenomenon whereby numerous such apps showcase their core content through WebView, while the app itself comprises only limited code.

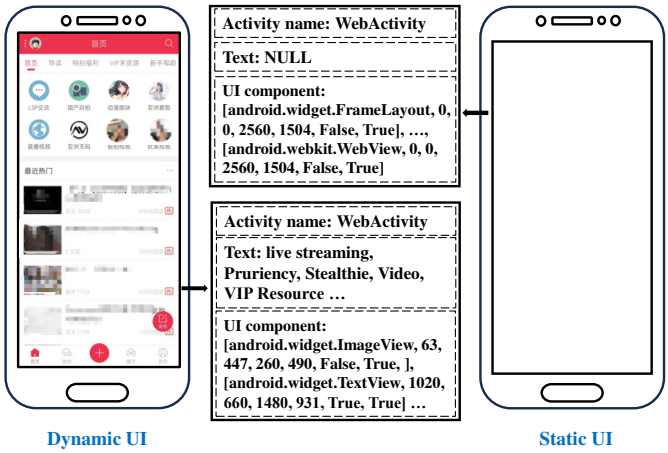

Fig. 2: A porn app's UI and the semantic information

Figure 2 illustrates the UI of a porn app, where a WebView spans the entire screen. Due to the absence of actual web content loading, static analysis can hardly extract any semantic information that is critical for detection from this UI (visually inspecting its layout only reveals a blank page) and is also unable to trigger any UI transitions. On the other hand, dynamically loading the actual UI can reveal rich semantic content and facilitate the transition to other UIs. In summary, the static UTG fails to adequately capture the real UI transitions and semantic information, thereby diminishing the efficacy of the detection process. TACDROID endeavors to address this challenge by introducing a more precise UTG that integrates both dynamic and static UTGs.

Figure 3 illustrates TACDROID's methodology for constructing the UTG for this porn app by correlating static and dynamic UTGs. Specifically, TACDROID first applies both static and dynamic analysis to construct the static UTG and the dynamic UTG, respectively. Given that the primary functionality of this app is achieved through WebView, the dynamic UTG provides a more comprehensive representation than the static UTG.

Subsequently, TACDROID then integrates two UTGs into a Seed UTG by matching nodes and edges. Finally, TACDROID calculates the edge scoring adjacency matrix based on UTG topology and GUI attributes, using it to update the Seed UTG to form the final UTG. In Figure 3, TACDROID eliminates edges introduced by the over-approximation of the static UTG (from Node *Config* to Node *Register*) and successfully identifies transitions not originally present in the Seed UTG (transition between Node *Video 2* and Node *Video 3*).

Leveraging the more precise transitions and richer semantic content in the final UTG, TACDROID accurately identifies the app as illicit and appropriately categorizes it. In contrast, DeUEDroid relies solely on the static UTG and erroneously classifies this app as legitimate.

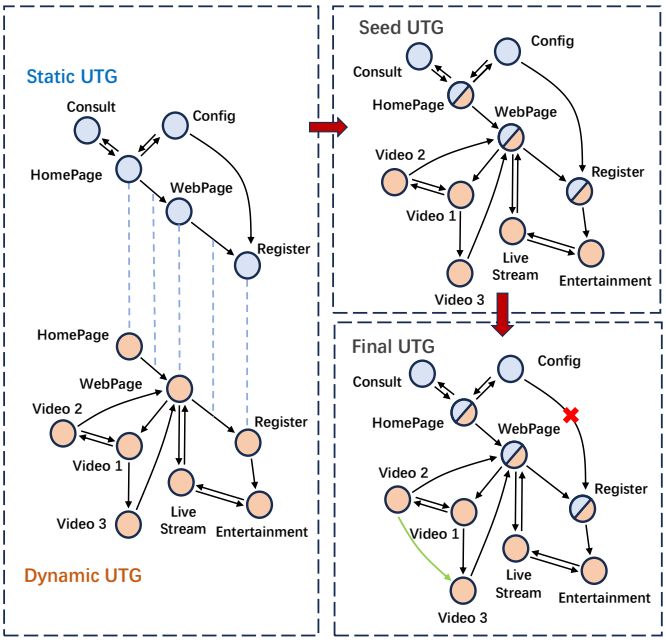

Fig. 3: An example of the UTG created by TACDROID for a porn app, correlating static and dynamic UTGs

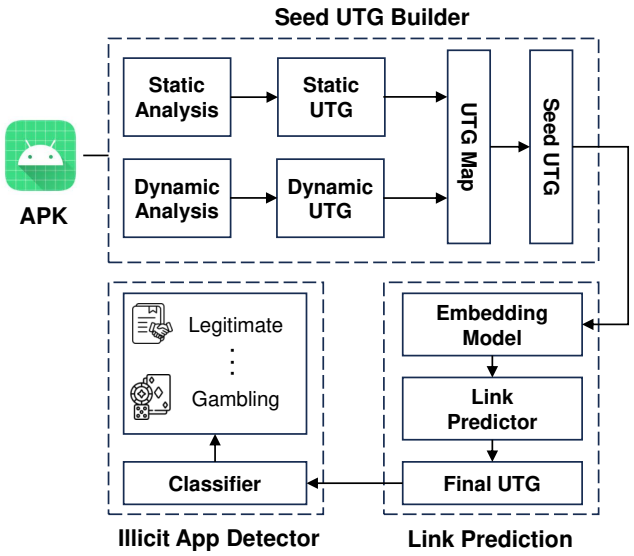

Fig. 4: Workflow of TACDROID

V. DESIGN OF TACDROID

Figure 4 provides an overview of TACDROID, which comprises three main modules: *Seed UTG Builder*, *Link Prediction*, and *Illicit App Detector*.

- **Seed UTG Builder**: This module accepts an Android app APK file as input and applies both static and dynamic analysis to generate the static and dynamic UTGs, respectively. TACDROID fuses these UTGs to create the Seed UTG.
- **Link Prediction**: The link prediction module focuses on predicting and adjusting the connections within the Seed UTGs. TACDROID employs embedding models to transform graph characteristics into vectors. Subsequently, it utilizes an enhanced GAE model that leverages both graph features and node features to reconstruct the graph's adjacency matrix. This procedure yields the final UTGs suitable for classification purposes.
- **Illicit App Detector**: The illicit app detector utilizes UTG characteristics and employs a graph convolutional network (GCN) to identify illicit apps and classify them into specific categories (*i.e.*, *porn*, *gambling*, *scam*, or *legitimate*), based on data from the current ground truth dataset.

A. Seed UTG Builder

Given an app, the Seed UTG builder module first constructs a static UTG and a dynamic UTG respectively, and finally fuses the two UTGs as the Seed UTG.

Static UTG Construction. Our initial step involves utilizing static analysis to generate the UTG. Prior studies [17, 18, 26, 27] commonly use open-source tools for static UTG construction. We adopt a similar approach by selecting the most suitable static UTG construction tool for our study from several latest tools, *i.e.*, PermDroid [17], GoalExplorer [18], and Frontmatter [26].

TABLE III: Evaluation of Static UTG Construction Tools

	PermDroid	GoalExplorer	Frontmatter
# UTGs constructed	68	34	68
# UTGs with Edges	59	12	29

To determine the best tool, we randomly selected 70 apps from our ground truth dataset and employed PermDroid, GoalExplorer, and Frontmatter to build static UTGs for each app. As shown in Table III, PermDroid successfully constructed UTGs for 68 apps, with 59 of those 68 UTGs containing both nodes and edges. In contrast, GoalExplorer constructed UTGs for 34 apps, with only 12 of those containing both nodes and edges. Frontmatter constructed UTGs for 68 apps, like PermDroid, but only 29 of them included edges. In summary, PermDroid demonstrated superior stability compared to GoalExplorer and Frontmatter, and hence we employ PermDroid for constructing our static UTG.

PermDroid retrieves widget declarations from XML layout files and identifies transition targets by analyzing the callback methods of event listeners. Considering that widgets can also be dynamically added in code, we enhance PermDroid to incorporate the detection of dynamically added widgets.

We utilize the call graph constructed with FlowDroid to identify the activities actually displayed, as detailed in §IV-A. Subsequently, we employ Soot [28] to construct the CFG for each method within these activities and search for invocations of event listener registration methods such as `setOnClickListener`. We begin with backward dataflow analysis on the caller of the listener registration method to locate the widget's declaration (*e.g.*, `new Button(this)`). After obtaining the widget's reference, we retrieve the widget's properties from setter methods (*e.g.*, `setText`) on the CFG. Additionally, the parameter of the event listener registration

method is an instance of an implementation of the corresponding event listener interface, which contains the event handling logic in its callback method (*e.g.*, `onClick`). We can determine the transitions associated with the widget by conducting data flow analysis on the relevant callback method.

Specifically, PermDroid initially retrieves the values of Intent arguments for inter-component communication (ICC) methods (*e.g.*, `startActivity`) that trigger UI transitions. PermDroid employs backward data flow analysis to trace the flow of the Intent, examining preceding statements to determine where the Intent was assigned or modified. This process involves identifying the original initialization of the Intent and tracking any subsequent changes to its properties. Given that both the Intent object and its field data can be passed among function calls, PermDroid utilizes FlowDroid for interprocedural analysis to track various Intent transmissions, including assignments from method returns, parameters passed into methods and class fields. For explicit intents, PermDroid identifies methods such as `setClass` and constructor calls that directly specify the target activity, recognizing the target activity from their parameters. For implicit intents, PermDroid traces method calls and assignments that modify the Intent to gather its attributes (*e.g.*, action, category), then matches these attributes against predefined Intent filters to determine the target activity. It is worth noting that due to the challenges in analyzing ICC methods [29], the UTG constructed by PermDroid is imprecise. To address this issue, we introduce dynamic analysis.

Dynamic UTG Construction. To complement static UTGs, which usually suffer from low-accuracy representation of an app's functionalities, our design also incorporates dynamic UTG. We employ the breadth-first search (BFS) strategy for dynamic exploration and construct the dynamic UTG using DroidBot [30], a renowned dynamic analysis tool.

Mapping of Static UTG and Dynamic UTG. In this phase, we fuse static UTG with dynamic UTG of an app, by building mapping between the two UTGs in terms of nodes and edges.

We introduce a UTG mapping algorithm (Algorithm 1) to achieve this objective. Initially, we establish correlations between nodes from the static UTG and those from the dynamic UTG. For each node N_s in the static UTG, we compute its UI similarity U_{sim} with every node N_d in the dynamic UTG. Specifically, we first define the attribute string of a widget as the concatenation of its `resource-id`, `class`, and `text` attributes. Then we calculate the Levenshtein similarity between the attribute strings of widgets from N_s and N_d that have the same XPath path. The Levenshtein similarity is defined as $1 - LevenDis/\max(L_A, L_B)$, where L_A and L_B represent the lengths of the two strings, and $LevenDis$ denotes the Levenshtein distance between the two strings. The average Levenshtein similarity, U_{sim}, is computed as the mean of all these similarity values.

We then map N_s to the N_d with the maximum U_{sim}. Furthermore, to ensure accuracy, we only consider two nodes to be identical if their U_{sim} surpasses the specified threshold. We evaluate this approach across 300 apps collected from F-Droid [31] and observe that setting the threshold above 0.87 ensured accurate mapping for all correlated node pairs.

After UI mapping, based on the principle that "When the same UI node triggers the same transition, the destination UI node is uniquely determined", we traverse all edges, if two edges have the same triggering widget and event, and share the identical source node, we consider these two edges to be equivalent (Lines 8-12). Besides, we label the target nodes of the transition accordingly as matching nodes (Line 13). After returning the U_i consisting of matched nodes and edges. Finally, we can easily utilize U_i to align the dynamic UTG with the static UTG, resulting in the Seed UTG.

Algorithm 1 UTG mapping algorithm

Input: UTG_s, UTG_d: the static and dynamic UTG for apk
Output: UTG_i: the Intersection of UTG_s and UTG_d

```
1:  function UTG_Alignment(UTG_s, UTG_d)
2:      pairs ← Get_similar_nodepair(UTG_s, UTG_d)
3:      nodes ← {}
4:      edges ← {}
5:      for (node_s, node_d) in pairs do
6:          nodes.add(node_s)
7:      end for
8:      for e_s in UTG_s.edges do
9:          for e_d in UTG_d.edges do
10:             if (e_s.src, e_d.src) in pairs then
11:                 if e_s.feature equals e_d.feature then
12:                     edges.add(e_s)
13:                     nodes.add(e_s.dst)
14:                 end if
15:             end if
16:         end for
17:     end for
18:     UTG_i ← New_Graph(nodes, edges)
19:     return UTG_i
20: end function
```

B. Link Prediction

With the Seed UTGs constructed, we utilize neural networks to learn the embedding of information collected from Seed UTGs and achieve link prediction. Figure 5 illustrates the design of the link prediction module. It consists of two main components: embedding models and a link predictor. This module takes a Seed UTG as input and produces a Final UTG. Next, we elaborate on the components of link prediction.

1) Embedding Models: The embedding models are used to transform the feature information obtained by the Seed UTGs into feature vectors. Specifically, TACDROID leverages the following features.

- **Activity name**: Previous work [32] demonstrated that using activity name to form vectors as node features can yield positive results in link prediction task. We employ a pre-trained BERT model for fine-tuning to encode an activity name into a 64-dimensional vector.
- **Text**: In the process of fusing static and dynamic UTGs, we concatenate a node's texts in both static UTGs and dynamic UTGs to obtain texts with no more than 256 tokens. Then

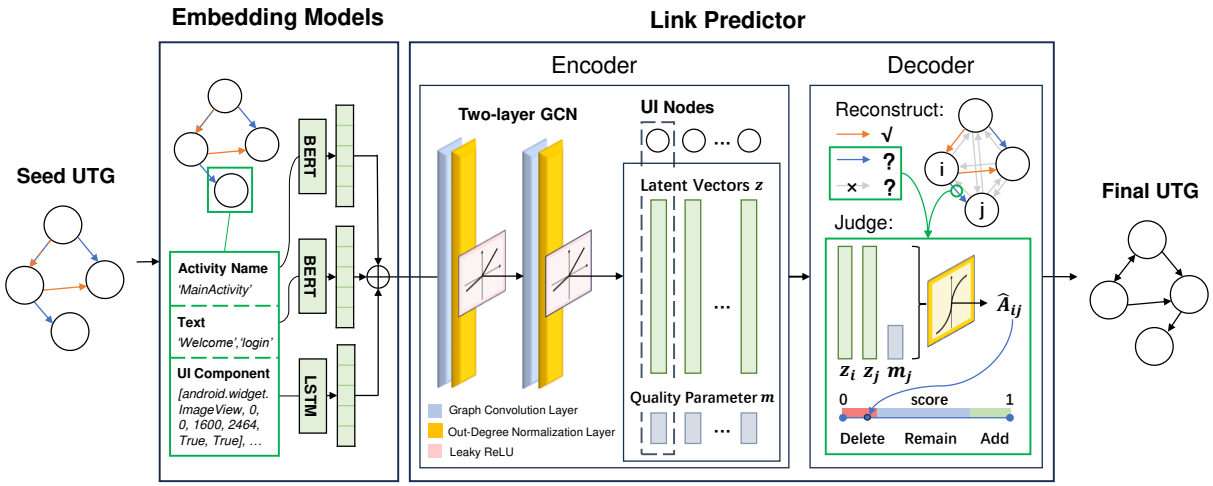

Fig. 5: Design of Link Prediction Module

another pre-trained BERT model fine-tuned through Masked Language Model (MLM) tasks is applied to obtain a 64-dimensional text feature vector.

- **UI component**: We include UI components by traversing each node in the UI tree and extracting attributes like node type, size, position, and specific properties related to transitions, such as `clickable` and `enabled`. Subsequently, we employ an LSTM model to encode this sequence.

Eventually, TACDROID obtains multiple vectors through the embedding process and concatenates them together to form a graph containing feature vectors for subsequent prediction and classification operations.

2) Link Predictor: Given the intricate information within the UTGs of apps and the limitations of static and dynamic analyses, we employ a Graph Autoencoder (GAE) to discern connections between nodes. However, the decoder structure of GAE struggles with directed graphs. To overcome these limitations, we adopt the Gravity-Inspired Graph Autoencoders [33]. This approach retains directional information and simplifies the reconstruction of adjacency matrices by employing straightforward matrix operations on node features from the convolutional layer. This eliminates the necessity for additional concatenation and network structures to compute scores between nodes. The model specifically comprises the following two components, *i.e.*, encoder, and decoder.

Encoder. Compared to conventional encoder in GAE, the normalization method applied in graph convolution formulas is out-degree normalization instead of symmetric normalization, which is more effective in handling the directionality of directed graphs. Additionally, Graph Convolutional Networks (GCN) assign a feature vector with the size of $d+1$, in which the first d dimensions correspond to the latent vector representation of the node, and the last value of the vector is deemed as *quality parameter* [33], which allows two nodes to give different *acceleration* to each other. TACDROID utilizes two layers of GCN to implement the encoding process.

Decoder. After obtaining the latent vectors z and the quality parameters m, the possibility for the existence of a directed edge from N_i to N_j is defined as follows [33]:

$$\hat{A}_{ij} = \sigma(m_j - log||z_i - z_j||_2^2) \tag{1}$$

where z_i and z_j stand for the latent vector of nodes N_i and N_j, m_j denotes the quality parameter of node N_j, and σ is the sigmoid activation. Therefore, TACDROID can calculate the reconstruction matrix $\hat{A}$ of the adjacency matrix A directly from the feature matrix obtained through convolution. Due to the existence of m_j, generally we have $\hat{A}_{ij} \neq \hat{A}_{ji}$.

Obtaining accurate and comprehensive UTGs through both dynamic and static analyses is challenging, and manually calibrating a large number of precise UTGs is impractical due to the extensive workload. To overcome this challenge, we employ a two-phase training process. In the initial phase, we directly utilize edges and non-existing connections sampled randomly from Seed UTGs to pretrain the model. To address accuracy issues with dynamic and static UTGs, we also utilize a small set of manually calibrated UTGs in a fine-tuning phase to adjust the parameters of the link predictor model, enabling it to adapt to a broader range of transition patterns and improve edge accuracy in the output UTGs.

Following training, the model can predict the likelihood of edge presence in the input graph. Based on the computed score indicating the probability of a directed edge which ranges from 0 to 1, we predict and add undiscovered directed edges with high scores, which should have existed but were overlooked by both static and dynamic UTG construction tools. Conversely, we remove directed edges with low scores that only exist in the static UTG. Consequently, establishing upper and lower bounds becomes necessary. The inclusion of two thresholds aims to mitigate the influence of prediction outcomes on the UTG, ensuring that edge information derived from both dynamic and static UTGs is preserved in the new UTG, rather than being entirely disregarded post-convolution.

To determine the thresholds, we input all UTGs from the training set into the link prediction model after fine-tuning and obtained the predicted scores for all edges in the graph. We

then sorted these scores in ascending order and used the values at the 1/4 and 3/4 positions as the lower and upper thresholds, specifically 0.202 and 0.408, respectively.

C. Illicit App Detector

After constructing a refined UTG through link prediction for an app, our subsequent step is utilizing the UTG to determine whether an app is illicit and constructing a classifier to categorize the apps. Considering that metadata partially reflects an app's characteristics (*e.g.*, app names featuring terms like "casino"), we integrate metadata elements (*e.g.*, package name, icon, certificate) into each node of the UTG.

Specifically, we apply a BiGRU model to the strings of the app name, package name, and certificate to derive a 128-dimensional vector. For icons, recognizing that the text within icons may contain app information—such as illicit apps often featuring website domains on their icons, unlike legitimate apps—we employ EasyOCR [34] to extract the text from the icons. We then set the pixels containing text to 1 and those without text to 0. Subsequently, we utilize the R, G, and B channels of the icon, along with the brightness, as four 2D features. These 2D arrays are concatenated and inputted into a CNN model, resulting in a 96-dimensional feature vector. Finally, we concatenate the metadata with the feature vectors of each node to obtain the final vector.

To determine the most suitable classification model, we evaluated three classical graph neural network models—GCN, GAE, and GraphSAGE—on a four-class classification task using the ground truth dataset. After conducting 5-fold cross-validation, the results indicated that GCN achieved an F1-score of 92.09%, GAE 90.10%, and GraphSAGE 89.38%. Consequently, we selected the GCN model for subsequent classification tasks due to its superior performance. Our GCN architecture comprises two convolutional layers with LeakyReLU activation and average pooling to consolidate node features into a graph feature vector, culminating in a fully connected layer that outputs four dimensions.

VI. Evaluation

We test all tools on a 64-bit Windows 10 physical machine with an Intel Core i5-12600 CPU and 32GB of RAM. All apps are tested in the Genymotion emulator with an 8-core CPU and 4GB of RAM. Machine learning models are trained in an NVIDIA GeForce RTX 3090 GPU (24GB memory).

To evaluate the effectiveness of TacDroid, we aim to answer the following research questions:

- **RQ1:** How effective is TacDroid in constructing UTGs?
- **RQ2:** To what extent do various features of TacDroid contribute to the detection of illicit apps?
- **RQ3:** How effective is TacDroid in detecting and classifying illicit apps?

A. RQ1: UTG Construction Effectiveness

Setup. To evaluate the effectiveness of the UTG construction algorithm, we first need to pretrain and fine tune our link prediction model. Given that we evaluated TacDroid's performance in detecting illicit apps using a ground truth dataset, and considering that the link prediction model's outcomes directly influence TacDroid's final detection results, training the link prediction model on a subset of the ground truth dataset might introduce train-test contamination. Therefore, we obtained an additional small set of apps specifically for training and testing the link prediction model in this evaluation.

(i) Pretrain. First, to encompass a variety of UTG types, we selected 2,713 apps from 10 different categories on Google Play based on the criteria of having more than 5 activities. We constructed Seed UTGs for these apps and used them for training. The parameters of the embedding model are determined during training using these dynamic UTGs and are subsequently frozen.

(ii) Fine-tune. In the fine-tuning phase, we use manually calibrated UTGs. Specifically, we collect an additional set of 120 apps, with 30 apps in each of four categories: porn, gambling, scam, and legitimate. We manually explore these apps and build ground truth UTGs with the assistance of PermDroid. Additionally, for potential transitions that are unreachable during exploration, we decompile the app and analyze the feasible transitions with the help of two experts who have extensive Android development experience. Table IV presents the ground truth UTGs. We use half of this data for fine-tuning and reserve the other half as a test dataset. During the fine-tuning process, the link predictor learns the transition relationships of the UI nodes. The model is trained with 70 epochs, lasting approximately 5 hours. Note that to avoid data leakage, the apps used in our pre-training and fine-tuning processes are not part of the ground truth data.

(iii) Metrics. To evaluate the effectiveness of TacDroid in constructing UTGs, we compared TacDroid and baseline methods by building UTGs and comparing them to ground truth UTGs. Our evaluation was based on precision, recall, and F1-score as key metrics. Specifically, we defined an edge in the UTG as a true positive (TP) if it satisfied the following criteria: both its source and target nodes exist in the ground truth UTG, and an edge connecting the same source and target nodes was also present in the ground truth UTG. Any edge that did not meet these conditions was labeled as a false positive (FP). Conversely, edges that were expected in the ground truth UTG but were absent in the constructed UTG were categorized as false negatives (FN). Edges correctly absent in both UTGs were classified as true negatives (TN). We calculated precision as $prec = \frac{TP}{FP+TP}$, recall as $rec = \frac{TP}{TP+FN}$, and the F1-score as $F_1 = \frac{2*prec*rec}{prec+rec}$. This comprehensive evaluation enabled us to thoroughly assess the UTGs constructed by TacDroid alongside the baseline methods. Each dynamic analysis tool underwent a consistent 15-minute duration of dynamic exploration.

To demonstrate the effectiveness of our approach, we compared it with SOTA baselines, including 3 static analysis tools (*i.e.*, Gator [27], ICCbot [35], and PermDroid), 2 dynamic analysis tools (*i.e.*, DroidBot and Stoat [36]), and 2 hybrid analysis methods. For hybrid analysis, we individually eval-

TABLE IV: Statistics of UTGs we constructed for 120 apps

Type of Apps	Fine-tuning dataset			Test dataset		
	# Apps	# Nodes	# Edges	# Apps	# Nodes	# Edges
Porn	15	242	637	15	147	428
Gambling	15	183	642	15	193	534
Scam	15	101	326	15	119	211
Legitimate	15	237	909	15	239	802
Total	60	763	2,514	60	698	1,975

TABLE V: Evaluation results of UTG construction

	Method	Precision	Recall	F1-score
Static Analysis	Gator	40.65%	49.67%	47.71%
	ICCbot	48.37%	54.18%	51.11%
	PermDroid[1]	58.39%	47.59%	52.44%
Dynamic Analysis	DroidBot	**100%**	49.72%	66.42%
	Stoat	**100%**	43.18%	60.32%
Hybrid Analysis	ProMal	79.95%	70.68%	75.03%
	Seed UTG	70.24%	**80.05%**	74.82%
	TACDROID	89.21%	78.43%	**83.48%**

[1] PermDroid is our improved version.

uated our Seed UTG, TACDROID (including both the Seed UTG module and the link prediction module), and ProMal [15]. Our Seed UTG is a fusion of static UTG and dynamic UTG (see §V-A); similarly, ProMal combines static analysis and dynamic analysis to predict transition pairs.

Results. Table V shows the performance of our approach and baselines. Since a large number of illicit apps use WebView to build apps and static analysis cannot analyze web content, static analysis cannot identify the transitions related to the dynamically-generated web content, which leads to poor performance of static analysis. While dynamic analysis can achieve 100% precision and identify web content accurately, it may have a significantly lower average recall rate. One main reason is that it is quite difficult for dynamic analysis to fully simulate user behavior or make transitions to specific activities that require login. For hybrid analysis, Seed UTG directly fuses static UTG with dynamic UTG. Despite the highest recall rate of 80.05% among all solutions, Seed UTG could blindly take in many UI transitions that should not have existed but are determined as reachable in static UTG, resulting in a precision rate of only 70.24%. Through link prediction, TACDROID eliminates 706 edges that are only present in static UTGs, out of which 517 are false positives. Additionally, it adds 198 predicted edges, of which 152 are true positives. This indicates that TACDROID can effectively remove actually non-existent edges and identify edges missing in both dynamic and static UTGs, significantly improving precision.

The enhanced performance of TACDROID over ProMal can likely be ascribed to two main factors. First, ProMal relies solely on the information between two nodes to determine the presence of a transition. In contrast, TACDROID leverages the intrinsic structural information found within the UTG to improve the prediction of transitions. Second, whereas ProMal merely validates statically identified edges absent in the dynamic UTG, TACDROID has the ability to predict new edges by learning from established transition patterns.

TACDROID identifies 1,743 edges, of which 982 are edges existing in the dynamic UTG that do not require verification. Of the 761 edges predicted through link prediction, 75.2% are true positives. In summary, TACDROID strikes a balance between precision and recall, achieving a high F1-score of 83.48% in transition identification. Compared to using PermDroid and DroidBot alone, TACDROID's F1-score increased by 31.04% and 17.06% respectively, indicating that TACDROID can effectively combine the advantages of dynamic and static analysis to construct a more accurate UTG.

B. RQ2: Ablation Experiment On Different Features

Setup. To evaluate the effectiveness of different features in TACDROID, we performed an ablation experiment using the ground truth dataset outlined in §III. Specifically, we investigated the following features:

- **Metadata.** The metadata of an app mainly includes its package name, app name, icon, and certificate (§V-C).
- **Seed UTG.** The Seed UTG is obtained by a fusion of the static UTG with dynamic UTG, possibly encompassing numerous unverified UI nodes and transition edges (§V-A).
- **Final UTG.** The Final UTG refers to the resulting UTG by applying the link prediction procedure to Seed UTG, which is a more precise representation.

We assessed the performance of these three features individually and in combination across five classification tasks. The four binary classification tasks entail identifying porn, gambling, scam, and illicit apps, while the fifth task involves multi-class classification of these four app categories. We conducted a 5-fold cross-validation using a batch size of 64 and utilized the F1-score as our performance metric.

Results. The results of the ablation experiment are presented in Table VI. It is apparent that the utilization of metadata alone leads to a higher F1-score for the detection of porn and gambling apps, as shown in column 2. This outcome is principally attributed to the distinctive features inherent in these app categories, such as app name and icon characteristics.

Moreover, it is evident that the incorporation of two types of UTG (Seed UTG and Final UTG) as features outperforms the usage of metadata alone. Discernible when comparing column 2 with columns 3-4. The notable disparity is particularly pronounced in scam app detection, with an F1-score increase exceeding 10%. This observation underscores the efficacy of UTG in capturing the subtleties among various apps.

Furthermore, among the two types of UTG, *Final UTG* demonstrates superior performance compared to *Seed UTG* (refer to columns 3 and 4). This discrepancy arises from the presence of numerous unreachable edges within *Seed UTG*, resulting from static analysis. Such unreachable edges diminish the representational effectiveness of *Seed UTG*, whereas *Final UTG* mitigates this issue through link prediction.

Additionally, *Final UTG* demonstrates a slight enhancement in performance when integrated with metadata (see columns

TABLE VI: Ablation experiment results, with F1-score as the performance metric

Task	Metadata	Seed UTG	Final UTG	Seed UTG+Metadata	Final UTG+Metadata (TACDROID)
Porn App Detection	85.71%	89.91%	94.07%	93.98%	**96.26%**
Gambling App Detection	79.63%	86.65%	94.12%	91.66%	**95.98%**
Scam App Detection	71.56%	82.23%	88.74%	88.33%	**91.31%**
Illicit App Detection	76.28%	90.83%	93.24%	94.28%	**96.73%**
App Classification	74.12%	85.98%	90.73%	88.75%	**92.09%**

TABLE VII: Illicit app detection and classification results

Method	Feature	Task	
		Binary Detection	Multiple Classification
Drebin	**Static Feature**	73.82%	69.34%
DroidAPIMiner	**API**	68.47%	62.69%
DeUEDroid	**Static UTG**	85.22%	80.82%
TACDROID	**UTG**	96.73%	92.09%

4 and 6), thereby reinforcing the proposition that metadata enhances the representational capabilities for app detection.

In fact, the amalgamation of *Final UTG* and metadata (column 6), which are also the features incorporated by TACDROID, achieves an 11% to 21% higher F1-score across all tasks compared to the other feature combinations listed in columns 2-5. This enhancement observed in the ablation experiment underscores the efficacy of TACDROID's feature selection in illicit app detection and app category classification.

C. RQ3: Illicit App Detection Effectiveness

Setup. To assess TACDROID's effectiveness, we conducted a comparative analysis with several classic works, including DeUEDroid, Drebin [37], and DroidAPIMiner [38]. DeUEDroid is regarded as a SOTA technique for identifying illicit apps, which employs the construction of a static UTG for illicit app detection. In contrast, Drebin and DroidAPIMiner rely on static features (*e.g.*, APIs, permissions, components, and URLs) along with the frequency of API calls, for malware detection. We conducted a binary illicit detection task and a multi-class app classification task on a ground truth dataset to evaluate TACDROID's performance.

Results. Table VII presents the F1-score of evaluation results. Drebin and DroidAPIMiner exhibited relatively lower performance in detecting illicit apps, which might be partially attributed to a less comprehensive consideration of apps' content attributes. DeUEDroid outperformed the other two approaches by employing a static UTG as a feature. However, DeUEDroid relies on static UTGs only and does not load web content, resulting in unsatisfactory performance in detecting scam apps and porn apps that heavily utilize WebView technology.

The integration of dynamic analysis in the design presents a potential drawback: the construction of a comprehensive dynamic UTG might necessitate a significant time investment for dynamic exploration. We separately evaluate the performance of employing solely static UTG and dynamic UTG as inputs

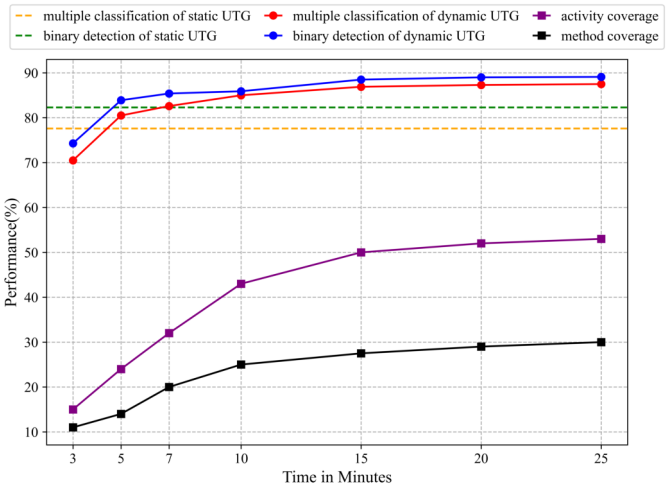

Fig. 6: The classification performance of static UTG and dynamic UTG as well as the activity coverage and the method coverage across the time

for two tasks. Figure 6 illustrates the coverages of dynamic analysis (the lower two curves) as well as the F1-score of dynamic and static analysis under different running times[3]. We measured two different types of coverage: activity coverage and method coverage. The activity coverage was reported by DroidBot, while the method coverage was measured by ACVTool [39]. Due to the inherent limitations of dynamic analysis, neither coverage metric can achieve full coverage of all activities and statements within the app. During the first 15 minutes, both coverage metrics increased rapidly. After 15 minutes of execution, the activity coverage and method coverage reached 49.8% and 27.5%, respectively. Beyond this point, the growth in coverage metrics slowed. By the 25-minute mark, the activity coverage and method coverage were 53.1% and 30.7%, respectively.

Unlike coverage metrics, which are significantly affected by running time, dynamic UTG demonstrates relatively stable performance regardless of the runtime duration. Dynamic UTG only lags behind static UTG at the minimum running time of 3 minutes but surpasses it when the duration extends to 5 minutes or longer. Notably, at the 7-minute mark (the average runtime for constructing static UTG), dynamic UTG

[3]Unlike dynamic analysis, static analysis cannot be halted at any time during execution. Consequently, we did not impose any restrictions on the running time for static analysis. The average and median durations for constructing the static UTG are 7 minutes and 5 minutes, respectively.

achieves F1-scores over 85% for both tasks. This resilience may be attributed to developers typically highlighting apps' characteristics and functionalities on the homepage, thereby furnishing abundant semantic information for dynamic UTG.

In conclusion, TACDROID's combination of static and dynamic analysis equips it with a substantial advantage in building detailed and informative UTGs, significantly enhancing its capabilities in illicit app detection. Moreover, TACDROID exhibits exceptional performance consistently across various durations of dynamic analysis, underscoring its potential for broad-scale application.

VII. DISCUSSION

Dataset. While we employed a majority voting approach to maximize the accuracy of annotating illicit apps during our data collection, it's important to acknowledge that categorizing illicit apps can sometimes be challenging. For instance, many porn apps may promote gambling content, and numerous gambling apps may use explicit material to attract users. This ambiguity can, to some extent, impact our system's performance. However, it's worth noting that organizations overseeing app stores and regulatory agencies primarily concentrate on the binary classification problem of whether an app is illicit or not, and our TACDROID could deliver excellent classification performance of 96.73% (see Table VI).

External validity. Due to the limited number of ground truth UTGs, our constructed link prediction model may deteriorate when applied to a larger scale of apps. Furthermore, given the rapid evolution of illicit apps, the model's effectiveness in detecting the most current illicit apps may diminish if not regularly updated. Additionally, since different countries and regions have varying definitions of illicit apps, users may build their own datasets and train models to suit their specific needs.

Ethical Considerations. Similar to previous studies [7, 40], the dataset used in this study also originates from law enforcement procedures conducted by ISPs. Throughout our study, we rigorously managed the data and its processing in accordance with common practices and guidelines as recommended in the Menlo reports [41]. We note that our data neither contains any personally identifiable information (PII) nor involves the activities of end-users. Moreover, for the purpose of analyzing and characterizing illicit apps, we accessed and downloaded these apps from various websites and marketplaces. While this activity may indirectly generate financial profits for the developers of these illicit apps, we consider such profits to be marginal, especially when compared to the broader benefits of this research, which ultimately contributes to improving the detection of illicit content and reducing criminal activities.

Limitations. We employ both PermDroid and DroidBot for static and dynamic analysis, respectively. Consequently, TACDROID inherits the limitations of these tools, such as difficulties in handling reflection and callback order resolution in PermDroid, and the loop exploration problem in DroidBot.

VIII. RELATED WORK

Illicit App Analysis. Previous research has delved into the analysis of illicit apps in various domains [42–46]. For instance, Gao *et al.* [7] conducted a comprehensive characterization of gambling apps and their ecosystem. Hu *et al.* [8] focused on detecting and understanding the characteristics of fraudulent dating apps. Hong *et al.* [5] investigated mobile gambling scams, revealing different transaction patterns within various gambling apps using ground truth data. Lee *et al.* [42] explored a new type of malicious crowdsourcing clients, shedding light on the mobile-based crowdturfing ecosystem and the strategies employed by underground developers. In our research, we contribute to this body of work by emphasizing the metadata of illicit apps. Furthermore, we aim to develop an illicit app detection system based on these characteristics.

Android GUI Analysis. Android GUI analysis has been a well-explored domain, involving both dynamic and static analysis techniques. Several notable works have contributed to this field. Gator's work [27, 47, 48] focuses on modeling Window Transition Graphs (WTG) through static analysis. These WTGs describe relationships between UI widgets, offering valuable insights into an app's user interface. Dynamic analysis approaches, as demonstrated in studies [49, 50], provide higher accuracy and create more detailed state transition graphs. These graphs illustrate transitions between various states of an app during runtime execution. Several studies, including [18, 51–53], integrate static UTGs into the process of guiding dynamic exploration. This combination enhances the overall understanding of an app's UI transitions and behaviors. Machine learning techniques have been employed to analyze Android app UIs. DeepIntent [54] leverages machine learning to understand the behavior of icons within technical analysis apps. This approach aids in identifying the functionality and purpose of different UI elements. ArchiDroid [32] leverages semantic data extracted from activity names to augment the accuracy of its activity transition graph (ATG). Unfortunately, the unavailability of ArchiDroid's source code repository renders direct comparison with our work unfeasible.

Malicious/Illicit App Detection. The detection of malicious/illicit apps has been well-explored in previous research [55–58]. For example, Arp *et al.* [37] proposed Drebin, which relies on static features such as permissions and APIs to identify malware. Also, other studies employ API analysis or Markov chains of APIs [59–61] to identify malicious software. Chen *et al.* [10] introduced DeUEDroid, a method tailored for the detection of illicit apps, which constructs a static UTG as a feature for identifying illicit apps. In another approach, Zhang *et al.* [62] proposed integrating domain knowledge into machine learning models to enhance the detection of evolving malware variants. In our study, we adopt a combination of static and dynamic analysis to create a more accurate and representative UTG, using it to detect illicit apps.

IX. Conclusion

Illicit apps have caused substantial economic losses in recent years, yet there remains a notable scarcity of relevant research in this domain. Our investigation first uncovers the widespread adoption of WebView in illicit apps, highlighting the deficiencies of current detection approaches primarily reliant on static analysis alone. We then proposed TacDroid, which harnesses both dynamic and static analyses alongside link prediction techniques, significantly enhancing its capabilities in illicit app detection. TacDroid surpasses existing SOTA solutions in both UTG construction and illicit app detection, and also demonstrates its potential for widespread adoption. We have made TacDroid and the largest ground-truth dataset for illicit apps publicly accessible on GitHub, aiming to stimulate further research in this domain.

Acknowledgment

This work was supported in part by National Key R&D Program of China (2023YFB3106800) and by National Natural Science Foundation of China (62272410). Haitao Xu is the corresponding author.

References

[1] "Xinhuanet: Mobile apps are extensively exploited in telecomm fraud." http://www.news.cn/2022-04/14/c_1128560504.htm, 2022, accessed November, 2022.

[2] "Fake gambling platform scam cases up by 18 times, investment scams more than double." https://www.channelnewsasia.com/singapore/investment-fake-gambling-platform-scams-police-banks-444476, 2021, accessed November, 2022.

[3] "Google Play Policy Center," https://support.google.com/googleplay/android-developer/topic/9858052, 2022, accessed December, 2022.

[4] "App Store Review Guidelines," https://developer.apple.com/app-store/review/guidelines/, 2022, accessed December, 2022.

[5] G. Hong, Z. Yang, S. Yang, X. Liaoy, X. Du, M. Yang, and H. Duan, "Analyzing Ground-truth Data of Mobile Gambling Scams," in *IEEE S&P*, 2022.

[6] Z. Chen, L. Wu, Y. Hu, J. Cheng, Y. Hu, Y. Zhou, Z. Tang, Y. Chen, J. Li, and K. Ren, "Lifting the grey curtain: Analyzing the ecosystem of android scam apps," *IEEE TDSC*, pp. 1–16, 2023.

[7] Y. Gao, H. Wang, L. Li, X. Luo, G. Xu, and X. Liu, "Demystifying Ilegal Mobile Gambling Apps," in *Proceedings of the Web Conference*, 2021, pp. 1447–1458.

[8] Y. Hu, H. Wang, Y. Zhou, Y. Guo, L. Li, B. Luo, and F. Xu, "Dating with Scambots: Understanding the Ecosystem of Fraudulent Dating Applications," *IEEE TDSC*, 2018.

[9] C. W. Munyendo, Y. Acar, and A. J. Aviv, ""Desperate Times Call for Desperate Measures"': User Concerns with Mobile Loan Apps in Kenya," in *IEEE S&P*, 2022.

[10] Z. Chen, J. Liu, Y. Hu, L. Wu, Y. Zhou, Y. He, X. Liao, K. Wang, J. Li, and Z. Qin, "DeUEDroid: Detecting Underground Economy Apps Based on UTG Similarity," in *ACM ISSTA*, 2023.

[11] "Android Developers – Webview API." https://developer.android.com/reference/android/webkit/WebView, 2023, accessed February, 2023.

[12] D. Tian, Y. Ma, A. Balasubramanian, Y. Liu, G. Huang, and X. Liu, "Characterizing Embedded Web Browsing in Mobile Apps," *IEEE TMC*, vol. 21, no. 11, 2022.

[13] "Tacdroid," https://github.com/initfunction/TacDroid.

[14] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, and A. Rountev, "Static window transition graphs for Android," *Automated Software Engineering*, vol. 25, 2018.

[15] C. Liu, H. Wang, T. Liu, D. Gu, Y. Ma, H. Wang, and X. Xiao, "ProMal: Precise Window Transition Graphs for Android via Synergy of Program Analysis and Machine Learning," in *IEEE ICSE*, 2022.

[16] X. Ge, S. Yu, C. Fang, Q. Zhu, and Z. Zhao, "Leveraging Android Automated Testing to Assist Crowdsourced Testing," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, 2022.

[17] S. Yang, Z. Zeng, and W. Song, "Permdroid: automatically testing permission-related behaviour of android applications," in *ACM ISSTA*, ser. 2022.

[18] D. Lai and J. Rubin, "Goal-driven Exploration for Android Applications," in *IEEE/ACM ASE*, 2019.

[19] "Group in Telegram." https://telegram.org/tour/groups, 2022, accessed June, 2022.

[20] Z. Chen, L. Wu, Y. Hu, J. Cheng, Y. Hu, Y. Zhou, Z. Tang, Y. Chen, J. Li, and K. Ren, "Lifting the grey curtain: Analyzing the ecosystem of android scam apps," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 3406–3421, 2024.

[21] A. F. Hayes and K. Krippendorff, "Answering the call for a standard reliability measure for coding data," *Communication methods and measures*, vol. 1, no. 1, pp. 77–89, 2007.

[22] "AppChina," http://m.appchina.com/, 2022, accessed October, 2022.

[23] "VirusTotal," https://www.virustotal.com/gui/, 2022, accessed November, 2022.

[24] "FlowDroid," https://github.com/secure-software-engineering/FlowDroid, 2024, accessed July, 2024.

[25] C. M. Pinto and C. Coutinho, "From native to cross-platform hybrid development," in *2018 International Conference on Intelligent Systems (IS)*, 2018, pp. 669–676.

[26] K. Kuznetsov, C. Fu, S. Gao, D. N. Jansen, L. Zhang, and A. Zeller, "Frontmatter: mining android user interfaces at scale," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1580–1584. [Online]. Available: https://doi.org/10.1145/3468264.3473125

[27] A. Rountev and D. Yan, "Static reference analysis for gui objects in android software," in *CGO*, ser. 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 143—-153. [Online]. Available: https://doi.org/10.1145/2581122.2544159

[28] "Soot," https://github.com/soot-oss/soot, 2022, accessed October, 2022.

[29] J. Yan, S. Zhang, Y. Liu, X. Deng, J. Yan, and J. Zhang, "A comprehensive evaluation of android icc resolution techniques," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3560420

[30] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: a Lightweight UI-Guided Test Input Generator for Android," in *IEEE ICSE-C*, 2017.

[31] "fdroid," https://f-droid.org/, 2022, accessed August, 2022.

[32] Z. Liu, C. Chen, J. Wang, Y. Su, Y. Huang, J. Hu, and Q. Wang, "Ex pede Herculem: Augmenting Activity Transition Graph for Apps via Graph Convolution Network," in *IEEE ICSE*, 2023.

[33] G. Salha, S. Limnios, R. Hennequin, V.-A. Tran, and M. Vazirgiannis, "Gravity-inspired Graph Autoencoders for Directed Link Prediction," in *ACM CIKM*, 2019.

[34] "EasyOCR," https://github.com/JaidedAI/EasyOCR, 2024, accessed August, 2024.

[35] J. Yan, S. Zhang, Y. Liu, J. Yan, and J. Zhang, "Iccbot: fragment-aware and context-sensitive icc resolution for android applications," in *IEEE ICSE-C*, 2022, pp. 105–109.

[36] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[37] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket." in *NDSS*, 2014.

[38] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *SecureComm*. Springer, 2013, pp. 86–103.

[39] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw, "Fine-grained code coverage measurement in automated black-box android testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, jul 2020. [Online]. Available: https://doi.org/10.1145/3395042

[40] A. Noroozian, J. Koenders, E. van Veldhuizen, C. H. Ganan, S. Alrwais, D. McCoy, and M. van Eeten, "Platforms in Everything: Analyzing Ground-Truth Data on the Anatomy and Economics of Bullet-Proof Hosting," in *USENIX Security*, 2019.

[41] M. Bailey, D. Dittrich, E. Kenneally, and D. Maughan, "The Menlo Report," *IEEE Security & Privacy*, vol. 10, no. 2, 2012.

[42] Y. Lee, X. Wang, K. Lee, X. Liao, X. Wang, T. Li, and X. Mi, "Understanding iOS-based crowdturfing through hidden UI analysis," in *USENIX Security*, 2019.

[43] E. Blancaflor, R. L.-P. J. Pastrana, M. J. C. Sheng, J. R. D. Tamayo, and J. A. M. Umali, "A security and vulnerability assessment on android gambling applications," in *CCET*. Springer, 2023.

[44] F. Dong, H. Wang, L. Li, Y. Guo, T. F. Bissyandé, T. Liu, G. Xu, and J. Klein, "Frauddroid: Automated ad fraud detection for android apps," in *ESEC/FSE*, ser. 2018.

[45] Y. Hu, H. Wang, Y. Zhou, Y. Guo, L. Li, B. Luo, and F. Xu, "Dating with scambots: Understanding the ecosystem of fraudulent dating applications," *IEEE TDSC*, vol. 18, no. 3, pp. 1033–1050, 2019.

[46] Y. Hu, H. Wang, R. He, L. Li, G. Tyson, I. Castro, Y. Guo, L. Wu, and G. Xu, "Mobile app squatting," in *Proceedings of The Web Conference 2020*, 2020, pp. 1727–1738.

[47] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *IEEE ICSE*, vol. 1, 2015, pp. 89–99.

[48] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, and A. Rountev, "Static window transition graphs for android," *Automated Software Engineering*, vol. 25, pp. 833–873, 2018.

[49] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *IEEE ICSE*. IEEE, 2019, pp. 269–280.

[50] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 94–105.

[51] S. Yang, Z. Zeng, and W. Song, "PermDroid: Automatically Testing Permission-related Behaviour of Android Applications," in *ACM ISSTA*, 2022.

[52] W. Guo, L. Shen, T. Su, X. Peng, and W. Xie, "Improving automated gui exploration of android apps via static dependency analysis," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 557–568.

[53] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *IEEE/ACM ASE*. IEEE, 2019, pp. 1070–1073.

[54] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu *et al.*, "Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps," in *ACM CCS*, 2019, pp. 2421–2436.

[55] 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.

[56] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," *arXiv preprint arXiv:1612.04433*, 2016.

[57] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye, "Significant permission identification for machine-learning-based android malware detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[58] J. Kim, Y. Ban, E. Ko, H. Cho, and J. H. Yi, "MAPAS: a practical deep learning-based android malware detection system," *International Journal of Information Security*, vol. 21, no. 4, 2022.

[59] 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, 2019.

[60] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Asia JCIS 2012*. IEEE, 2012, pp. 62–69.

[61] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *ACM CCS*, 2014, pp. 1105–1116.

[62] 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 *ACM CCS*, 2020, pp. 757–770.