UI-CTX: Understanding UI Behaviors with Code Contexts for Mobile Applications

Jiawei Li[†‡*] Jiahao Liu[‡*] Jian Mao[†§¶] Jun Zeng[‡] Zhenkai Liang[‡]
[†]Beihang University [‡]National University of Singapore
[§]Tianmushan Laboratory [¶]Hangzhou Innovation Institute, Beihang University
{daweix, maojian}@buaa.edu.cn {jiahao99, junzeng, liangzk}@comp.nus.edu.sg

***Abstract*—Many mobile apps utilize UI widgets to interact with users and trigger specific operational logic, such as clicking a button to send a message. While UI widgets are designed to be intuitive and user-friendly, they can also be misused to perform harmful behaviors that violate user expectations. To address these potential threats, recent studies strive to understand the intentions of UI widgets in mobile apps. However, existing methods either concentrate on the surface-level features of UI widgets, failing to capture their underlying intentions, or involve tedious and faulty information, making it challenging to distill the core intentions. In this paper, we present UI-CTX, which demystifies UI behaviors with a concise and effective representation. For each UI widget, UI-CTX first represents its intentions with a *UI Handler Graph (UHG)*, incorporating the code context behind the widget while eliminating irrelevant information (*e.g.,* unreachable code blocks). Then, UI-CTX performs graph summarization and explores both the structural and semantic information in UHGs to model the core intentions of UI widgets. To systematically evaluate UI-CTX, we extract a series of UI widget behaviors, such as login and search, from a large-scale dataset and conduct extensive experiments. Experimental results show that UI-CTX can effectively represent the intentions of UI widgets and significantly outperforms existing solutions in modeling UI widget behaviors. For example, in the task of classifying UI widget intentions, UHG achieves the highest average F1-score compared to other widget representations (+95.2% and +8.2% compared with permission set and call sequence, respectively) used in state-of-the-art approaches. Additionally, by accurately pinpointing the code contexts of widgets, UI-CTX achieves a 3.6× improvement in widget intention clustering performance.**

I. Introduction

Mobile applications (*a.k.a.,* apps) have become an indispensable part of our daily lives, providing powerful support for various activities such as travel, education, and finance [1], [2]. To enhance user experience, most apps are equipped with UI widgets, such as buttons and text boxes, to facilitate smooth user interactions. Specifically, the functionalities and behaviors of mobile apps are often driven by these UI widgets [3], such as executing a search query when a user presses a *search* button. However, not all UI widgets perform as intended by their design, which can raise security and privacy concerns [4]–[6]. For instance, a *login* operation in a phishing app may send the user's credentials to an attacker instead of initiating a legitimate login request. To detect potentially harmful behaviors that violate user expectations, it is crucial to accurately identify and understand the true intended behaviors behind UI widgets in mobile apps. Existing solutions for representing UI widgets' intended behaviors generally fall into three categories: appearance-based, permission-based, and code-based methods.

Appearance-based methods use the appearance (*e.g.,* texts and images) of UI widgets as the primary source to describe their intentions [7]–[9]. However, due to the lack of underlying execution semantics, these solutions fail to recognize the real intended behaviors of UI widgets, especially when counterfeit apps mimic the appearance of legitimate apps [10]. To resolve this issue, *permission-based methods* represent UI widgets' intended behaviors by analyzing their associated permissions [4]. Nevertheless, with the increasing complexity of app behaviors, similar permissions may be shared, leading to a significant mix-up and increased manual efforts for security analysts. For example, both a *login* and a *help* operation may require the same `INTERNET` permission. As a remedy, *code-based methods* analyze the associated code contexts of UI widgets to uncover their deeper latent functionalities [3], [5]. One common approach is to leverage the invocation relationships among API calls (*e.g.,* call graph). However, these techniques often introduce unnecessary code parts (*e.g.,* code blocks that are not reachable from the UI widgets), making it challenging to identify the true functionality of UI widgets. For instance, existing methods over-approximate code contexts connected to UI widgets and bind more code than necessary [4], [11]–[13].

To resolve this issue, we propose UI-CTX, an approach that specifies and understands UI widgets' intended behaviors with code contexts triggered by UI events in mobile apps. We focus on Android in this study since it is the most popular mobile platform with a large number of apps [8], but the general idea applies to graph user interface (GUI) applications on other platforms. Specifically, UI-CTX is based on three key insights. First, UI widgets usually trigger certain app behaviors when their status changes by UI events (*e.g.,* being clicked, selected, or focused). The corresponding event callbacks are the starting points of the UI's background logic. This inspires us to locate the specific callback and its subsequent invocations for each UI widget to reveal its functionality. Second, developers often use encapsulation-based external library APIs to acquire

[*] Co-primary authors. Jian Mao is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240238
www.ndss-symposium.org

certain functionalities, such as HTTP connections. While the implementation details of these libraries are usually tedious and complex compared to the execution logic constructed by the app developer, understanding how the APIs interact is essential to discerning the UI widgets' intended behaviors. This motivates us to prune and summarize the meaning of these external libraries, avoiding their dominance over the entire code context and reducing the analysis burden. For example, if a UI widget invokes a library API for data encryption, we should focus more on the context of the encryption and the high-level data operation (*e.g.,* encrypt and save data) as the UI widget's functionality, instead of the detailed implementation of the data encryption algorithm. Third, both the semantics within a function and its structural information play crucial roles in understanding the intended behavior of UI widgets. This guides us to explore the code context from both semantic and structural perspectives to describe the intended functionality of a UI widget.

We represent UI widget functionality with *UI Handler graph (UHG)*, a graph representation where nodes denote function calls and edges represent the invocation relationships between them. To construct the UHG, we employ a backward analysis starting from the event handler rather than a forward analysis from the UI widget. This strategy tracks the data flow from the event handler to the UI widget, constructing the minimal event handler set for the UI widget and reducing false connections. We further perform a fine-grained analysis of the conditional code branches and thread blocks in event handlers to identify the true triggered callback functions for different UI widgets. Additionally, we summarize the key semantics of the external libraries to reduce noisy graph parts and focus on essential information. Finally, we leverage the instruction-level information of the functions in the UHG to enhance the semantics of the graph. Having built the UHG to describe UI widget functionality, we incorporate both structural-level (*e.g.,* inter-procedural control flow) and node-level (*e.g.,* instruction semantics of functions and APIs) information to embed the UI widget's intention. We then conduct an unsupervised clustering analysis to categorize UI widgets according to their intended functionalities.

To investigate UI-CTX's effectiveness in describing UI widget functionality, we compile a comprehensive dataset of 40,000 Android apps from public repositories. These apps are sourced from various platforms, including Google Play, AppChina, Mi and Anzhi, and span ten year periods. For the widget categories, we focus on common and popular UI widgets, particularly those related to user privacy and data operations, such as *login*, *search*, and *location*. Evaluation results demonstrate that UHG more accurately depicts and distinguishes UI widget functionality compared to widely used approaches based on permission sets and call sequences. Additionally, we also conduct a series of ablation studies to highlight the importance of addressing the challenges faced by existing solutions. Our findings also indicate that a deeper understanding of UI widget functionality can significantly assist real-world security applications, such as in-depth widget analysis and anomaly detection.

In summary, we make the following contributions:

- We highlight the importance and identify the challenges of UI widget behavior representation, and design *UI Handler graph (UHG)*, a concise and accurate representation to describe the intended behavior of a UI widget.
- We present UI-CTX, an end-to-end approach to build UHG, summarize and embed the semantics in UHG to investigate UI widget functionality[1].
- Experimental results show that UI-CTX can effectively identify UI widgets with similar functionalities, enhancing Android app analysis through a better understanding of UI widget behavior.

II. Background and Motivation

In this section, we introduce background knowledge, utilize an example to underscore the need for a concise and accurate representation to describe UI widgets' intentions, and outline the challenges associated with formulating UI widget functionalities of existing methods.

A. Background

An Android app typically consists of foreground UIs for user interaction and backend code to implement corresponding app behaviors. A UI window within an app is termed an *activity*, where UI widgets are initialized using attributes like ID, text, and image source from tree-structured layout resources. To interact programmatically with widgets, developers associate a UI layout with a code class or instance using APIs like `setContentView` and `inflate`, and retrieve widgets via `findViewById` to find the widgets [14]. Once a specific widget is retrieved from the UI, its attributes can be modified or overridden. In addition, user-triggered events (*e.g.,* `onClick`, `onTouch`, and `onKey`) are assigned to UI widgets for the binding of the widget's and its execution logic. Since Android operates on an event-driven model, a widget's code context stems from its event handlers. Additionally, developers may reuse a widget (with a fixed ID) across different UI pages, making it crucial to uniquely bind widget IDs, layout IDs, and event handlers during widget analysis to prevent ambiguity.

B. Motivation Example

Figure 1 presents two visually similar *login* buttons from two different apps: the left one from a popular instant messaging app and the right one from a phishing version. Both widgets display the text "*LOGIN*" and trigger the same permission, `android.permission.INTERNET`, suggesting data exchange with the Internet. However, these surface-level information fail to depict their true functionalities in detail. A closer analysis of their code contexts shows clear differences: the legitimate app uploads user credentials to an HTTP server, while the phishing app sends them to an external email address using the JavaMail API [15]. These function call operations, including how data is fetched and

[1]To facilitate further analysis and research, we open source the implementation of UI-CTX on https://github.com/DaweiX/UI-CTX.

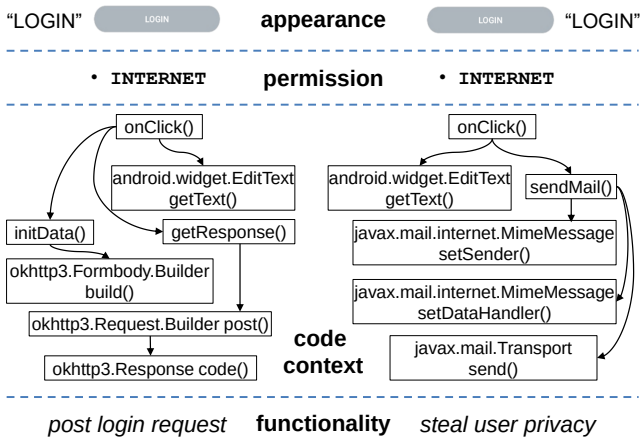


Fig. 1: Two *login* buttons with different functionalities.

transmitted, serve as fundamental components of the UI widget functionality and key to understanding widget behavior. This insight drives us to utilize detailed code contexts to accurately infer and analyze widget intentions.

C. Challenges in Existing Solutions

Code contexts encapsulate essential information and can serve as a basis for understanding UI widget functionalities. However, we identify several inherent challenges faced by existing approaches in describing the intended functionalities of UI widgets based on code-level semantics.

Challenge A: Finding the correct code context for the intended widget functionality. Existing solutions typically employ a context-insensitive reference strategy to determine the corresponding code contexts for UI widgets [4], [12], [13]. These methods decompile app bytecode into an intermediate language (IL) and trace the reachability of UI widget variables within the app's codebase by performing forward analysis on the callsites where these variables are used. However, this forward analysis often overestimates widget-event pairs due to the repeated use of some temporary variables, mistakenly linking multiple unrelated event handlers to a single widget. Listing 2a illustrates an IL code snippet featuring three UI widgets and their respective event handlers. By analyzing variable reachability, we observe that the return value of retrieving the widget with ID 2131230766 flows to $\$r6$, which then "accesses" all of $\$r11$, $\$r12$, and $\$r13$, corresponding to three different event handlers on Lines 11, 14, and 17. However, this UI widget is actually linked only to one event listener class at Line 9 (`Gallery$Adapter$7`).

Additionally, it's important to note that not all code contexts originating from an event handler are necessarily triggered at app runtime. This is because a single event handler can encompass multiple event callbacks through conditional branches [16]. For example, Listing 2b illustrates an event handler containing various branches for different widgets. Existing solutions often operate at the level of event handlers and overlook the fine-grained code contexts within these handlers, which results in a lack of precision when identifying the

```
1  // app: com.mediaaz.bryanadamtopfree
2  $r6 = invoke $r1.<findViewById(int)>(2131230766)
3  $r11 = (ImageView) $r6
4  $r6 = invoke $r1.<findViewById(int)>(2131230767)
5  $r12 = (ImageView) $r6
6  $r6 = invoke $r1.<findViewById(int)>(2131230768)
7  $r13 = (ImageView) $r6
8  ...
9  $r25 = new Gallery$Adapter$7
10 invoke $r25.<Gallery$Adapter$7: void <init>(...)
11 invoke $r11.<void setOnClickListener(...)>($r25)

12 $r26 = new Gallery$Adapter$8
13 invoke $r26.<Gallery$Adapter$8: void <init>(...)
14 invoke $r12.<void setOnClickListener(...)>($r26)

15 $r27 = new Gallery$Adapter$9
16 invoke $r27.<Gallery$Adapter$9: void <init>(...)
17 invoke $r13.<void setOnClickListener(...)>($r27)
```

(a) Setting event handlers by anonymous classes.

```
1  // app: com.bionicpanda.aquapets
2  r0 := @this: TankWallActivity
3  $r1 := @parameter0: View
4  $i0 = invoke $r1.<View: int getId()>()
5  lookupswitch($i0) {
6    case 2131427367: goto s1;
7    case 2131427707: goto s2;
8    case 2131427708: goto s3;
9    case 2131427713: goto s4;
10   default: goto return;
11 }
12 return
13 s2: invoke r0.<TankWallActivity: void finish()>()
14 return
15 s1: invoke <game.b: void a(int)>(0)
16 invoke r0.<TankWallActivity: void a(int)>(99)
17 return
18 s3: $r2 = new Intent
19 $r3 = invoke r0.<TankWallActivity: Context
       getApplicationContext()>()
20 ...
21 return
22 s4: $r4 = r0.<TankWallActivity: EditText e>
23 $r5 = invoke $r4.<EditText: Editable getText()>()
```

(b) Setting code logic to be triggered by branches.

Fig. 2: Assigning code contexts for different UI widgets (we make some simplifications on the IL for illustration).

intended functionality of a specific UI widget. This observation is substantiated by our empirical study using 20,000 apps sampled from our collected dataset (see Section IV). We investigate cases where multiple function callbacks for different UI widgets are consolidated into a single event handler. The cumulative distribution function (CDF) curves in Figure 3 demonstrate that 21.99% of apps integrate multiple callbacks within one handler. Moreover, while most handlers contain no more than 20 branches, some handlers manage callbacks for up to 58 UI widgets. On average, one event handler contains 5.39 branches for different UI widgets.

Challenge B: Focusing on widget functionality amidst extensive code contexts. Modern app development frequently relies on mature third-party library (TPL) and software development kit (SDK) APIs to leverage rich functionalities such as data encryption and HTTP requests. However, integrating these encapsulated APIs introduces significant challenges,

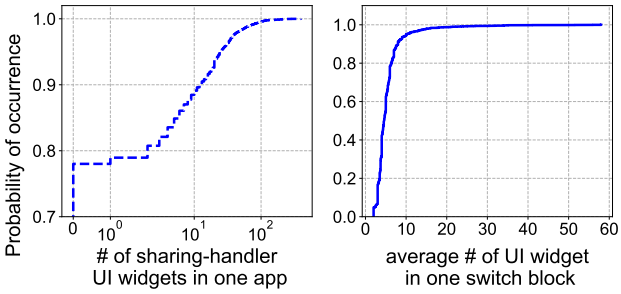


Fig. 3: CDF curves of widget event callbacks of apps.

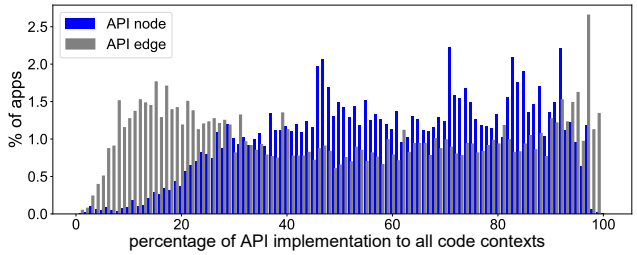


Fig. 4: Distribution of the percentage of API code in the complete app call graph.

including managing large codebases during code analysis. We further investigate the usage of TPL and SDK in 20,000 apps, and illustrate the distribution of code contexts invoked by external TPL and SDK APIs in Figure 4. Our analysis reveals that in 88.4% of apps, more than 30% of function calls originate from external APIs rather than from app-specific code (*i.e.,* code written by app developers). Furthermore, for over half of these apps, TPL and SDK implementations constitute more than 60% of the entire app's code contexts. The prevalence of external API implementations in app codebases results in UI widget intentions becoming obscured within extraneous information. This makes it challenging to concisely and accurately represent the core functionality of UI widgets.

Challenge C: Representing meaningful function call semantics. To describe code semantics, previous approaches utilize class or method names to represent the semantic information of function calls [17]–[19]. For instance, given the function call *picasso.OkHttpDownloader: HttpURLConnection openConnection(Uri uri)*, *OkHttpDownloader* or *openConnection* can be used to represent its semantics since they reasonably reflect the function's purpose. However, app developers widely employ obfuscation techniques such as ProGuard [20] and packers [21] to safeguard their code against reverse-engineering [22]. These techniques often rename code identifiers (such as class names and method names) to meaningless strings (*e.g., a.a.z* for a class name, *z* for a method name). Moreover, apps may use method overloading to map unrelated identifiers to the same name [23]. Consequently, literal names extracted from decompiled apps may be insufficient for comprehensive code functionality analysis. Addressing this challenge requires aggregating fine-grained code semantics to accurately represent function calls within a code graph.

III. Design

A. Overview

Figure 5 illustrates the workflow of UI-CTX, which takes Android application packages (APKs) as input and analyzes the functionalities of UI widgets. The end-to-end framework of UI-CTX consists of three distinct phases: (1) *Multi-layer Knowledge Extraction* (Section III-B), which parses app resources to collect the UI layer corpus, including layouts, widget attributes, and string values. Additionally, it performs static code analysis to extract code layer information, such as inter- and intra-process control flow, data flow, and detailed implementations of different functions. (2) *UHG Construction* (Section III-C), which identifies the precise backend code contexts associated with each UI widget and represents them using graphs. UI-CTX further refines these graphs by eliminating and summarizing graph parts that could potentially obscure or mislead the analysis, such as detailed implementations of external libraries. (3) *Behavior Investigation* (Section III-D). This phase embeds the UHG representation and applies clustering analysis to investigate the UI widget functionalities.

B. Multi-layer Knowledge Extraction

Given an Android app, UI-CTX first extracts knowledge from both the UI and code layers, serving as the basis for UHG construction and UI widget behavior investigation.

UI Layer. UI-CTX analyzes the static resources bundled in `resources.arsc`, a binary file containing a table of app resources. Among different resource types, the *layout* resources establish relationships between UI layouts and UI widgets, and *string* resources map string references to literal values, such as *@string/changeLangTitle* → *"Select language"*. Some *layout* resources can also serve as components to be embedded and reused in complex UI pages. For example, the *activity_main* layout in Figure 6 introduces a button panel using the `include` tag (Line 3). UI-CTX identifies and unfolds such sub-layouts. In this case, the `ImageView` widget with ID *btn_login* will be regarded as a child widget of the layout *activity_main*. Such associating widgets and layouts is essential as UI widgets may have different functionalities when they are attached to different layouts. In the end, the UI resource parsing phase yields separate UI pages, each holding several UI widgets with attributes like `id`, `text`, and `src`, which define their appearance and semantics at the UI layer.

Code Layer. UI-CTX conducts a flow-sensitive and context-sensitive static analysis to construct a code graph for an Android app. This approach preserves the statement order and contextual information of function calls, facilitating the subsequent identification and summarization of the code context of UI widgets. Notably, UI-CTX does not differentiate between self-defined methods by app developers and API calls (*i.e.,* internal and external methods as in existing tools [24]), aiming to integrate as much information as possible when constructing the code graph. If the implementation of a method call does not appear in the disassembly output, UI-CTX searches for it within Android SDKs [25] compatible with the app (*e.g.,*

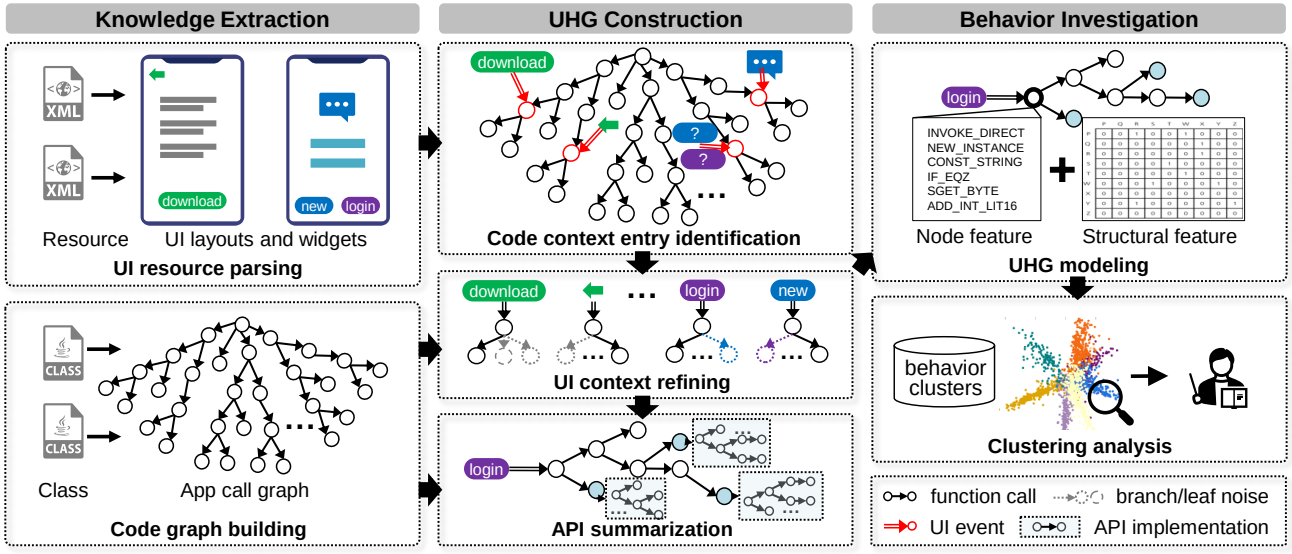


Fig. 5: The overview of UI-CTX, which consists of Knowledge Extraction, UHG Construction, and Behavior Investigation.

Symbol	Note
c, ξ_c	class instance and its heap in the address space, $c \in Class, \xi_c \in Addr$
m, f	method and field of class. $m \in Method, f \in Field$
x, y, z	local variables, $x, y, z \in Var$
s	an assignment statement, $s ::= \{x := y \mid x := y.f \mid x.f := y \mid x := y.m(z)\}$
σ	the environment that maps local variables to addresses, $\sigma \in Env = Var \rightarrow Addr$
η	the heap that manages the values of the fields of heap objects, $\eta \in Heap = Addr \times Field \rightarrow Addr$
id	a constant in the ID set for layouts and widgets $\mathbb{Z}$, $\mathsf{id} \in \mathbb{Z}$

TABLE I: Syntactic expressions.

minimal and target SDK versions). Finally, UI-CTX captures comprehensive code-layer knowledge, encompassing specific details of all calls and their relationships.

C. UI Handler Graph (UHG) Construction

Code Context Entry Identification. UI widgets in apps often utilize various event handlers, such as `onTouch`, `onDrag`, `onKey`, `onClick`, `onLongClick`, and `onFocusChange`, to trigger their working logic. As discussed in Section II-A, to complete the UI widget setup, an app developer needs to set or inflate a UI layout on the current screen window, retrieve a widget from the layout, and assign an event listener (with its handler callback) to the widget. Consequently, existing works [4], [8], [12], [13] follow the UI widget setup diagram $layout \rightarrow widget \rightarrow event$ and adopt a forward analysis mechanism to associate widgets with event handlers. However, the forward analysis tends to over-approximate and generate false UI-handler linkages due to temporary registers generated by the IL (*e.g.,* $\$r6$ in Listing 2a). To address this issue, UI-CTX employs a backward analysis workflow, $event \rightarrow widget \rightarrow layout$, to identify the code context entry of UI widgets. Specifically, UI-CTX performs a two-step procedure: (1) tracing the data flow from the event handler to the UI widget allocation (*e.g.,* a local variable or a class field), and (2) mapping the widget object to the layout it attaches on.

To better illustrate how UI-CTX identifies the entry points of UI widget code contexts, we present the syntactic expressions used in our code analysis in Table I. We elaborate on our two-step analysis workflow as below.

event → widget: UI-CTX initializes its analysis from event register methods, which serve as the entry points for the widget's code contexts. For example, consider the method m_{reg} = `setOnClickListener` in Figure 6, which registers an event listener ($x = listener$) for a widget object where c = `MainActivity` and f = `btnLogin`. Formally, the process can be described as:

$$\langle c.f.m_{reg}(x), \sigma, \eta \rangle \rightarrow \langle \sigma, \eta[c.f \mapsto c.f \cup \xi_x] \rangle.$$

The tracing process starts from the event register method m_{reg} and extends to the allocation point of the widget object. Specifically, we trace the register variable $\$r2$ through the following path: Line 28 ($\$r2$) → 25 (`btnLogin`) → 17 ($\$r2$) → 16 ($\$r1$) → 15 (the callsite of m_{find} = `findViewById`). Next, we examine the argument of method m_{find} to get the widget ID id_w. If the argument is a literal value in $\mathbb{Z}$, id_w is directly obtained. Otherwise, if the argument is a local variable, a similar backward data flow analysis is conducted to determine id_w. To confirm the listener class ξ_x for x ($\$r1$ in Line 28), we trace the data flow of x from Line 28 to Line 23 and resolve ξ_x = `MainActivity$1`. Once ξ_x is identified, we investigate its contextual information and locate the event handler implemented by ξ_x in Line 30. Consequently, we determine the event handler m_h = `MainActivity$1.onClick` as the entry point of the widget's code contexts[2].

widget → layout: To identify the layout of a widget, we focus on the method calls that set activity contents. For

[2]In cases where x refers to the current class instance (*i.e.,* $x = c.\mathtt{this}$), we associate the event handler overridden by c with the widget.

```
1  <android.support...PercentRelativeLayout ...>
2    <LinearLayout ...>
3      <include layout:"@layout/panel" />
4    </LinearLayout>
5    <TextView ... /> ...
6  </android.support...PercentRelativeLayout>
```

(a) layout: activity_main.xml

```
1  <FrameLayout ...>
2    <ImageView android:id="@+id/btn_login" /> ...
3  </FrameLayout>
```

(b) layout: panel.xml

```
1  public class MainActivity extends Activity {
2    private android.widget.Button btnLogin;
3    private static final String BTN_TEXT;
4    <MainActivity: String BTN_TEXT> = "login";
5    public void onCreate() {
6      MainActivity r0;
7      r0 := <MainActivity: String BTN_TEXT>
8      invoke r0.<Activity: void setContentView(int)
           >(2131427357);
9      invoke r0.<MainActivity: void bindView()>();
10     invoke r0.<MainActivity: void setView()>();
11     return; }
12   private void bindView() {
13     MainActivity r0;
14     r0 := @this: MainActivity;
15     $r1 = invoke r0.<Activity: View findViewById(
           int)>(2131230720);
16     $r2 = (android.widget.Button) $r1
17     r0.<MainActivity: Button btnLogin> = $r2;
18     return; }
19   private void setView() {
20     MainActivity r0;
21     View$OnClickListener $r1;
22     r0 := @this: MainActivity;
23     $r1 = new MainActivity$1;
24     invoke $r1.<MainActivity$1: void <init>(..)>();
25     $r2 = r0.<MainActivity: Button btnLogin>;
26     $r3 = r0.<MainActivity: String BTN_TEXT>;
27     invoke $r2.<void setText(CharSequence)>($r3);
28     invoke $r2.<void setOnClickListener...>($r1);
29     return; }}
30 public static class MainActivity$1 implements
       View$OnClickListener {
31   public void onClick(android.view.View) {
32     MainActivity$1 r0;
33     r0 := @this: MainActivity$1;
34     invoke r0.<MainActivity: void doLogin()>();
35     return; }}
```

(c) code: class `MainActivity`

Fig. 6: UI and code contexts of a UI widget. We make some simplifications on the IL for illustration. ◄···►: layout-code relation, ◄···►: layout-layout relation, - - - ►: data dependency, ·····►: control dependency.

example, in Figure 6, the method m_{inf} = `setContentView` in Line 8 is invoked to set the layout $R.layout.activity\_main$ as the content to display when the class c = `MainActivity` is running. This method call updates the heap as follows:

$$\langle c.m_{inf}(x \mid \mathsf{id}), \sigma, \eta \rangle \rightarrow \langle \sigma, \eta[\xi_c \mapsto \xi_c \cup \xi_{root}] \rangle,$$

where ξ_{root}, the address of the root View in the layout (*i.e.,* `PercentRelativeLayout`), is added to the heap. Similar to obtaining the widget ID id_w, we conduct backward data flow tracing to extract the layout ID id_ℓ from the argument of m_{inf}. It is important to note that UI-CTX also handles m'_{inf} = `inflate`, another way that is more complex but more flexible to inflate a layout into the current activity based on View inflation semantics [26], which can be formulated as:

$$\langle z := x.m'_{inf}(y \mid \mathsf{id}), \sigma, \eta \rangle \rightarrow \langle \sigma[z \mapsto \xi_{root}], \eta \rangle, \text{or}$$
$$\langle c.f := x.m'_{inf}(y \mid \mathsf{id}), \sigma, \eta \rangle \rightarrow \langle \sigma, \eta[c.f \mapsto \xi_{root}] \rangle.$$

Here, the return value of the function can be assigned to either a class field ($c.f$) or a local variable (z). We trace its dataflow to the widget allocation (*e.g.,* the m_{find} = `findViewById` callsite) to determine its layout. By combining id_w and id_ℓ, we can uniquely identify a widget. The event handler corresponding to the event register from which we initiate our analysis is then determined as the entry point of the widget's code contexts — all the subsequent code contexts of this handler serve as *candidates* to describe the widget's functionality.

Widget Context Pruning and Updating. Even if we have pinpointed the accurate event handler for a widget, not all code blocks originating from the handler can be triggered by the widget. To prune unrelated widget code contexts, UI-CTX executes a fine-grained code analysis to investigate which branches are truly responsible for the widget's behavior. The process consists of five steps: (1) splitting the event handler into smaller code blocks based on code branches (*e.g.,* `switch` and `if`); (2) identifying the first and last statements of each block by checking the control flow labels (*e.g.,* `goto` and `return`); (3) listing the block conditions; (4) including the code snippets under a code block whose condition matches the widget (*e.g.,* view id equals to a value) as valid code contexts for the widget; (5) performing a forward depth-first search (DFS) rooted at the event handlers of widgets to extract the code graph representing widget code contexts, guided by the block conditions. Taking Listing 2b as an example, UI-CTX first splits the *lookupswitch* part into five blocks based on the case labels. Next, it identifies *goto* and *return* statements to determine the block boundaries. Then, it lists and resolves the block conditions, checking the widget ID and prunes the code contexts that should not be connected to the widget. Finally, it performs a DFS to construct the code graph representing the widget's intentions. Note that when encountering implicit inter-procedural control flows introduced by multi-thread usage, UI-CTX does not employ a simple caller-callee pairing as existing work [4], which is another source of overestimating. Instead, we select the correct thread that will be launched by a widget among all the thread callbacks (similar as the event callback selection) by a customized dataflow analysis (more details can be found in Appendix A).

Since code-layer widget manipulations (*e.g.,* `setText`) may occur at app runtime, we also account for updating the widgets' UI-layer contexts accordingly. For instance, the `ImageView` in Figure 6 is initially defined in the layout file with only an `id` attribute. However, this widget will act as a login button during the activity's lifecycle. In this case, we conduct data flow analysis from $\$r2$ and $\$r3$ in Line 27 to determine the widget's actual attribute (*i.e.,* determine its

text as *login* in Line 4). Finally, for each widget event, we generate a unique UI handler graph (UHG), where a widget (with attributes like `id` and `text`) is connected to an event handler, followed by the widget's reachable code contexts, with each method call represented as a node, and its executed functionality as a feature. It is important to note that a single widget may have multiple UHGs for different events (*e.g.,* `onClick` and `onFocusChange`).

Graph Encoding and Summarization. To represent the semantics of function calls, existing solutions often rely on human-readable names in code [13], [17], [18]. However, it is challenging for these methods to capture stable and meaningful function call semantics when obfuscation or overloading exists. For example, semantics in class name or method signatures are often obfuscated or removed in real-world apps according to Google's app development principles, like shortening class/member names into meaningless literals (*e.g., a* and *b*) to optimize apps [27]. Therefore, to encode function calls in a UHG, we turn to Dalvik Opcodes [28] of each instruction within a function to represent function features and capture fine-grained program behaviors (*e.g.,* register operations). The principle behind this choice is that Opcodes serve as the fundamental building blocks of function implementations, reflecting the specific operations performed by the function [29]. For instance, `int-to-char` converts an integer to a character, and `if-eq` checks for equality between two operators. Moreover, the Opcodes of a function are relatively stable, as similar operations are often required to complete a given task.

When extracting semantics from a UHG, we observe that a significant amount of tedious information is introduced by the implementations of third-party libraries (TPLs) and SDK APIs, which may obscure the core functionality of UI widgets. To mitigate the influence of such noisy information, we propose to summarize and abstract the semantics of API implementations. Specifically, for each API call in the widget code contexts, UI-CTX iteratively aggregates the semantic information of its downstream graph nodes, from the leaves to the API node itself. By summarizing the features of the subgraph nodes into a single representation at the API node, the aggregation process reduces the complexity of the graph. This approach minimizes the impact of extraneous information and provides a clearer representation of the UI widget's core functionality. The detailed algorithm is illustrated in Appendix B.

D. UHG-based UI Behavior Investigation

Graph Embedding. Given the UHG $\mathcal{G}$ derived from the code and layout contexts, the next step is to embed it into a vector space for further analysis. The UHG comprises structural information (*e.g.,* function call relations) and node features (*e.g.,* Opcode-based instruction semantics). The structural information is crucial for uncovering the intricate relationships and interactions between graph entities [30], [31]. Additionally, node features detail method implementations. Our objective is to distill both structural and node features into a meaningful and informative representation for downstream tasks.

For structural information modeling, we take inspiration from the graph Laplacian operator, which defines an embedding that maps graph data into an Euclidean space [32]. Specifically, we extract structural features based on the normalized Laplacian matrix, given by

$$\mathscr{L} = I - \mathscr{A} = D^{-1/2}(D - A)D^{-1/2},$$

where $\mathscr{A}$ is the normalized adjacency matrix, A is the adjacency matrix of $\mathcal{G}$, and D is the diagonal matrix of node degrees. Among the eigenvalues $\boldsymbol{\lambda}$ of this Laplacian matrix, we select the top k *minimal* ones $\lambda_1, \lambda_2, \cdots, \lambda_k \in \boldsymbol{\lambda}$, because the smallest eigenvalues better capture the global structural information (*e.g.,* community structure and connectivity patterns) of the graph rather than falling into localized graph neighborhoods [33]. Thus, the structural embedding $\mathfrak{e}_{\text{str}}$ is defined as the average of the selected eigenvalues:

$$\mathfrak{e}_{\text{str}} = (\sum\nolimits_{i=1}^{k} \lambda_i)/k.$$

For node semantics, we implement a graph-level readout by combining the average-pooling $\mu(\cdot)$ and the standard deviation $\sigma(\cdot)$ of node features. Average pooling summarizes the central tendency of the features, while standard deviation captures the dispersion of the attributes, indicating diversity. Formally, the node embedding $\mathfrak{e}_{\text{node}}$ is defined as the concatenation of the average and standard deviation:

$$\mathfrak{e}_{\text{node}} = \mu(\boldsymbol{X}) \parallel \sigma(\boldsymbol{X}),$$

where $\parallel$ denotes the concatenation operator.

Finally, the UHG embedding is obtained by concatenating the structural and node embeddings:

$$\mathfrak{e} = \mathfrak{e}_{\text{str}} \parallel \mathfrak{e}_{\text{node}}.$$

Clustering Analysis. To validate the effectiveness of our UHG-based UI widget analysis, we conduct a clustering task to group UHG instances with similar functionalities. We use Agglomerative Hierarchical Clustering Analysis (HCA) to cluster the semantic representations of UHGs derived from the aforementioned code contexts. HCA is an unsupervised algorithm that recursively merges or splits clusters based on a distance metric, forming a tree-like structure of nested clusters. In our case, we utilize the Ward variance minimization method to determine cluster similarity with Euclidean distance, which minimizes the within-cluster variance when merging cluster pairs. The cluster distance d is calculated by:

$$d(u,v) = \sqrt{\frac{|v|+|s|}{T}d(v,s)^2 + \frac{|v|+|t|}{T}d(v,t)^2 - \frac{|v|}{T}d(s,t)^2},$$

where u is a new cluster formed by merging clusters s and t, v is a cluster to be compared, $|\cdot|$ denotes cardinality, and T is the sum of the cardinalities of v, s, and t. In summary, the clustering analysis allows us to identify and analyze patterns and similarities in the functionality of different UI widgets.

Category	Item	Average	Max
App overview	# of method call	12,006	141,572
	# of UI widget	150	12,237
	- # of in-code attribute	2.09	286
	# of edge	126,693	3,620,132
	# of event link in an app	7.49	1,228
	# of subgraph in an app	7.45	1,228
Code pruning	ratio of pruned node*	0.09	~ 1 ($\Delta < 10^{-4}$)
	ratio of pruned edge	0.09	~ 1 ($\Delta < 10^{-6}$)
UHG	# of node in a UHG	1,367	92,590
	# of edge in a UHG	11,444	3,053,091

* For the graph pruning operation $\mathcal{P}(\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathbf{X})) = \mathcal{G}'(\mathcal{V}', \mathcal{E}', \mathbf{X}')$, the ratio of pruned node is calculated by $|\mathcal{V} - \mathcal{V}'| / |\mathcal{V}|$. Other ratios in this table are given in a similar way.

TABLE II: Dataset overview.

IV. Evaluation

In this section, we evaluate UI-CTX by answering the following research questions (RQs):

- How UHG performs in widget intention description compared with existing representations? (§ IV-C)
- To what extent do the different design choices in UI-CTX contribute to its performance? (§ IV-D)
- Can UI-CTX facilitate real-world app analysis? (§ IV-E)
- What is the overhead of UI-CTX? (§ IV-F)

A. Implementation

We prototype UI-CTX in about 3.8 K lines of Java code and 2.6 K lines of Python code. Specifically, UI-CTX uses FlowDroid [34], a data flow analysis tool to decode app resources (*e.g.,* layout files) and manifest. We also leverage it to run inter-procedural code analysis and build apps' call graphs. Our customized code analysis, including pinpointing the code context entry of UI widget, extracting in-code UI widget property, and identifying code contexts that should be pruned, are based on the Jimple intermediate representation (IR) parsed by Soot [35]. We utilize LibRadar [36] to identify TPL/SDK APIs used in an app. To get the Opcodes from method implementations, we use dexlib2 [37] to parse dex files unzipped from apps and Android SDKs. We implement clustering analysis based on the SciPy library [38].

B. Dataset

To systematically investigate the performance of UI-CTX, we randomly collect 40,000 Android apps from AndroZoo [39], a publicly available and continually expanding app repository, following prior studies [40], [41]. To ensure that our dataset reflects the real-world app distribution in recent years, the apps are sampled from various market sources, such as Google Play, VirusShare, Mi, and Anzhi. Additionally, the dataset spans the past ten years, with 4,000 apps collected from each year, enhancing the comprehensiveness of the dataset and improving the generalizability of our evaluation.

Table II shows the statistics of the dataset. On average, UI-CTX generates 7.45 subgraphs containing widget functionalities for each app. The value is slightly less than the average $\langle widget, layout, event \rangle$ UI-code triples extracted from each app (7.49), because we discard code graphs with no concrete code contexts (*e.g.,* directly `return` after the event is triggered). Within each subgraph, the pruned code contexts that a widget cannot trigger account for 9.0% of the original subgraph on average, in terms of node and edge numbers. After pruning, for each app, UHGs' methods and call edges account for 10.1% and 9.3% of the complete app call graph on average, respectively.

Category	Metric	Permission	Call Sequence	UHG
delete	precision	0.67	0.57	0.57
	recall	0.19	0.75	0.89
	f1-score	0.29	0.65	**0.69**
login	precision	0.89	0.75	0.78
	recall	0.22	0.69	0.77
	f1-score	0.36	0.72	**0.78**
logout	precision	0.23	1.00	1.00
	recall	1.00	0.94	0.95
	f1-score	0.37	0.97	**0.98**
send	precision	0.93	0.65	0.79
	recall	0.45	0.71	0.67
	f1-score	0.60	0.68	**0.72**
search	precision	0.80	0.73	0.81
	recall	0.19	0.56	0.67
	f1-score	0.31	0.63	**0.73**
download	precision	0.74	0.72	0.88
	recall	0.34	0.81	0.85
	f1-score	0.47	0.76	**0.86**
share	precision	0.80	0.96	0.99
	recall	0.81	0.87	0.88
	f1-score	0.81	0.91	**0.93**
save	precision	0.85	0.65	0.72
	recall	0.24	0.62	0.70
	f1-score	0.38	0.63	**0.71**

TABLE III: Effectiveness results for different representations.

C. Effectiveness

Benchmark Setup. For better and fair evaluation, we build a benchmark dataset with ground truth labels. Following previous work [4], [41], we assume that a UI widget's functionality aligns with its appearance in benign apps (*i.e.,* apps with no malware detection engines on VirusTotal [42] flag as risky). For example, a *login* button is expected to facilitate user login. This assumption is based on the core user experience (UX) principle that UI widgets should be visually comprehensible to convey their purpose [43]. Additionally, benign apps are verified to be safe and perform their intended functions as expected [44], [45]. While edge cases like widgets with misleading texts or icons may exist, their rarity minimally impacts our findings.

In our benchmark, we focus on UI widgets related to account management and data operation, as they are common in apps and are highly related to user privacy, which is a critical concern in mobile security. Specifically, we choose eight categories of UI widgets with unambiguous UI texts, including *login*, *logout*, *send*, *delete*, *search*, *save*, *refresh*,

and *download*. During the benchmark construction process, we ensure that UI widgets in different categories are balanced. This is important to avoid overestimating and underestimating small classes and generating biased evaluation results [46]. We also implement strict labeling strategies to avoid inherent bias such as label shift [47], [48]. Specifically, we utilize the text on a widget to present its functionality. In the labeling process, we only consider widgets whose text exactly matches a functionality semantic, such as labeling a widget with the text *"login"* as *login*. This strategy helps to rule out noisy labels and misunderstandings of UI functionalities. For example, a *"Can't login?"* button is not considered as performing *login*, and we can filter it out. Finally, we collect and label 2,000 UI widget functionalities (250 for each category) from different apps to build our benchmark dataset.

Comparison of Widget Functionality Representations. To demonstrate UI-CTX's effectiveness in describing UI widget functionalities, we first compare UHG with two widget functionality representations applied in existing state-of-the-art methods, including permission in DEEPINTENT [4] and call sequence in DESCRIBECTX [13]. For a fair comparison among different representations, we conduct the same code context pruning and noise reduction before UHG, permission and sequence-based embedding. The baselines include:

- *Permission*: using triggered permissions to represent widget functionality. Following the prior literature [4], we traverse the UHG, collect method calls, and map them to a permission set using PSCOUT [49]. Then, we one-hot encode the permission set to represent the widget functionality.
- *Call sequence*: using the sequences of method names invoked by a widget to represent its functionality. We extract call sequences from UHGs. Then, similar to prior work [13], we feed the sequences into a learning model for context-aware text embedding to represent the widget functionality.

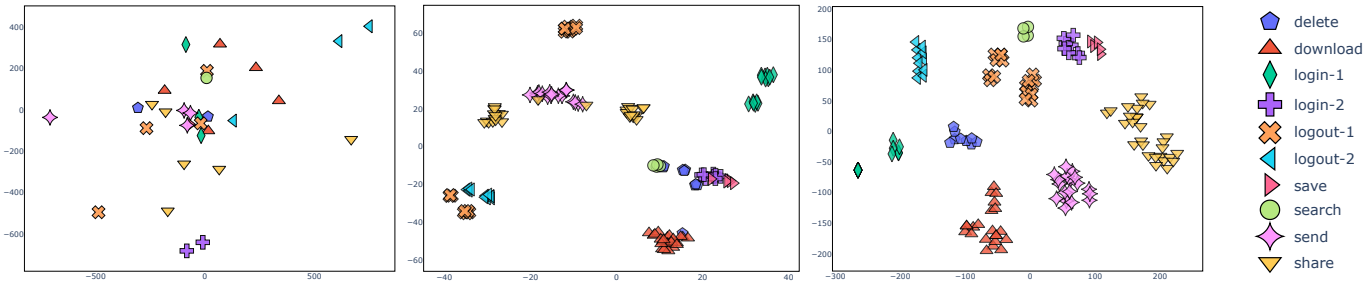
In this experiment, we conduct an eight-category classification task to evaluate the performance of UI-CTX and the baselines. We use precision (p), recall (r), and F1-score ($f1$) as the evaluation metrics. To eliminate bias and make a fair comparison, we extract permissions and call sequences from the same code graphs associated with widgets, which are also used for embedding UHG. Note that if we do not apply pruning and summarization on the code graph, all the representations will suffer a considerable performance decline (we put more details in Section IV-D). Table III illustrates the comparison results. From the table, we observe that UI-CTX achieves remarkable performance in all categories. In all the categories, it outperforms the baselines in terms of F1-scores, and achieves an F1-score improvement by 95.2% and 8.2% on average compared with permission set and call sequence, respectively. This validates UI-CTX's effectiveness in describing UI widget functionalities, especially for complex functionalities. We attribute the superior performance of UI-CTX to its ability to represent the widget functionality with an accurate and concise UHG, which can capture both the code semantics and the code interaction semantics of the widget.

To further illustrate the effectiveness of UI-CTX, we visualize the widget representations generated by different methods. We use t-SNE to project the embedding space into a 2D-plane to get an intuition of the embedding distribution [50]. Figure 7 shows the visualization for different categories. The UHG instances are automatically clustered into 10 groups, exceeding the 8 predefined categories. Upon manual inspection, we find that the additional clusters reflect fine-grained functional differences. For example, the *login* category splits into two clusters: a simpler one where the username is passed to another activity for further processing, and a more complex one that encrypts and sends user credentials, checks the response, and saves the login status (detailed descriptions are provided in Appendix D). This aligns with real-world scenarios where multiple implementations exist for the same functionality. By investigating the visualization results, we find that the UHG representations derived by UI-CTX are more accurate and concise than the permission and call sequence representations. Specifically, the samples in the same cluster are more similar to each other, and the boundaries between different clusters are more clear. This demonstrates the effectiveness of UI-CTX in describing UI widget functionalities.

Applicability of UI-CTX. In UI-CTX, we directly embed the structural and node features of UHGs without relying on priori signal (e.g., ground truth label) to fit the representation towards a convergence. Therefore, our approach may introduce errors when conducting unsupervised clustering. To assess its real-world applicability, we present the detailed clustering results in Table IV. On average, UI-CTX achieves an accuracy of 95.0% across all widget categories, demonstrating its effectiveness in capturing widget functionalities. Although the clustering results are promising, some misclassifications are observed. Upon closer investigation, two main factors contribute to UHG misclassification. First, app behaviors might be implemented by multiple UI pages and widgets. For instance, both the *send* button in app *com.autotaxi_call.patra18300* and the *search* button in *com.example.xianji* launch a new page for user confirmation or input. In these cases, the widgets are actually performing similar *navigation* functionality (with similar UHG structure), and will be very close in the embedding space. Such mistakes can be filtered out by applying simple rules, such as checking if the UHG contains API calls that only from specific namespaces like `android.app.Activity` or `android.view`. Another source of false positives or negatives is third-party libraries. For example, using LibRadar as a plug-in tool for library detection, UI-CTX fails to identify and summarize some libraries like *google.gson* and *adrt.ADRT*, leading to external codebases overshadowing UHG semantics. To address this, UI-CTX allows for loading an SDK/TPL list to complement the library detection module, or replacing the module with more advanced tools. The integration of existing tools will be discussed in Section V.

D. Contributions of Design Choices

In this section, we answer the second RQ by exploring the contributions of different design choices in UI-CTX.



(a) Permission set (b) Call sequence (c) UI handler graph

Fig. 7: Visualization of different representations.

Category	TN	TP	FP	FN	Category	TN	TP	FP	FN
delete	1,580	221	170	29	login	1,696	193	54	57
logout	1,750	237	0	13	send	1,705	167	45	83
search	1,711	166	39	84	download	1,721	211	29	39
share	1,748	221	2	29	save	1,682	175	68	75

TN - True Negative, TP - True Positive, FN - False Negative, FP - False Positive.

TABLE IV: Detailed effectiveness in describing functionalities for different widget categories.

Specifically, we focus on three components, including code context pruning, API summarization, and UHG embedding.

Code Context Pruning (Challenge A). Pinpointing accurate code contexts for UI widgets is the key step of UI-CTX. Here, we investigate the impact of our code context pruning strategy on functionality representation. We compare the capacity of permission set, call sequence and UHG on depicting widget functionality with and without removing the false UI-code linkages and unrelated condition branches by (1) replacing UI-CTX's event analysis module with GATOR [12] used in prior works [4], [13], and (2) disabling code branch checking. To measure the clustering performance, we apply *Homogeneity*, *Completeness*, *V-measure* [51] and *Adjusted Rand Index (ARI)* [52] as the metrics, all of which are widely used in clustering analysis. These metrics range from 0 to 1, with higher values indicating better clustering results. The detailed definitions of these metrics are provided in Appendix C.

Figure 8 shows the clustering results for different representations with and without code context pruning. Without the code context pruning, the clustering results of all the widget representations are significantly worse in terms of all the metrics. Specifically, the *V-measure* falls off by 28.9% on average across these three representations. This is within our expectation, as the over-approximation analysis introduces much noise into the code contexts for the widgets, leading to incorrect descriptions of their functionalities. Upon further analysis, we find that each UI widget in the benchmark is, on average, associated with 54.3% more code contexts that are unreachable [3]. We also observe that while most samples within each cluster have the same labels (indicating relatively high *Homogeneity*), the samples from the same category can

[3]Among the apps used for the benchmark, GATOR cannot finish analyzing 2,980 apps in a predefined timeout (30 minutes/app) and no events are obtained from these apps. We exclude these apps when calculating the metrics.

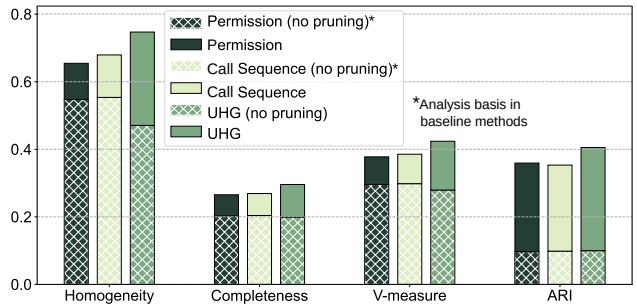


Fig. 8: Clustering results with and without pruning.

be dispersed across multiple clusters (indicating low *Completeness*). This results in a lower overall consistency between the clustering results and the true labels (reflected in a relatively low *ARI*). This phenomenon is consistent with the fact that there are multiple ways to implement the same functionality.

$ARI < 0.1$ for all representations based on unpruned code contexts, indicating near-random clustering results. It is important to note that prior methods, such as DEEPINTENT [4] and DESCRIBECTX [13], rely on permissions and call sequences from unpruned widget code contexts for their analysis tasks. However, after pruning unreachable widget code contexts, the *ARI* scores improve by an average of 3.6 times across the three representations, highlighting the critical role of code context pruning in improving widget analysis accuracy.

API Summarization (Challenge B). UI-CTX summarizes the semantics of API calls provided by external libraries (*i.e.,* TPL and SDK) instead of taking their detailed implementations. The API summarization not only reduces the graph size to a large extent but also improves the UHG's ability to represent widget functionalities. For example, when each API is summarized in one node, UI-CTX achieves the best performance in terms of the highest F1-score 0.81 with the least number of nodes (-67.9% on average) and edges (-72.1% on average). We list the detailed results for different API neighborhood hop numbers $1, 2, 3, 4, 5, 6$ and ∞ (*i.e.,* keeping all API implementations) in Appendix B.

Figure 9 illustrates the CDF curves for the ratio of the graph size after summarization to the original graph size. The node number-probability curve is pretty close to a linear distribution when we summarize each external API in a 4-hop neighborhood, indicating that the proportion of the reduced

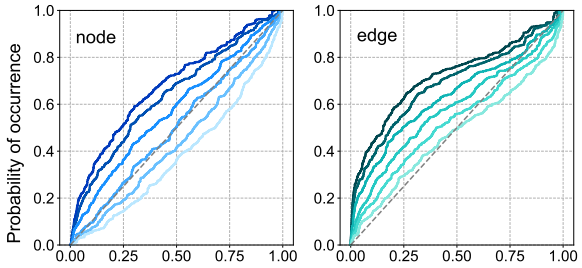


Fig. 9: Graph reduction by summarization. The x-axis represents the ratio of the graph size after summarization to the original graph size (the left and right subfigure uses nodes and edges as the unit, respectively). The y-axis denotes the probability of occurrence. Darker (top) to lighter (bottom) lines indicate API neighborhood hops from 1 to 6.

node number obeys an even distribution. This means if we take a smaller hop number, for most graphs, more than half of the nodes can be reduced after summarization. For example, when we use only one hop, the node number will be just one quarter (*i.e.,* abscissa value = 0.25) of the original value for 51.4% graphs. We also observe that the curves for edge reduction are closer to the upper left compared to those for node reduction for all the hop numbers, showing that more edges can be reduced than nodes. This is reasonable because one API may have more than one outgoing edges to other APIs (*e.g.,* JDK 1.8's `concat` method calls 5 APIs to concatenate a string to the end of another one).

UHG Embedding (Challenge C). Embedding is the prerequisite to represent a UHG as a vector for downstream tasks (*e.g.,* clustering analysis and distance visualization). We explore the power in widget functionality characterization of the following UHG embedding method variations:

- *structure only*: only keep graph adjacency information and abandon node features.
- *node only*: only use node features without considering graph structure and node relationships.
- *name only*: distill code semantics from method names instead of the instruction-level method implementations (we embed method names based on their contexts in sequences using the same model as in Section IV-C).

The results are shown in Table V. Among all the embedding methods, the embedding approach that integrates both call interaction and call implementation semantics achieves the best performance. Additionally, we observe that node features contribute more to the UHG embedding than structural features. Nevertheless, structural features still provide useful and complementary information to node features for improving the functionality representation.

We make another comparison between implementation- and name-based method embeddings. All of the F1-scores for different widget categories decrease when relying on names. This is consistent with prior observations that name-based methods are vulnerable to code obfuscation [29] and renaming [41]. Since the semantics of names are not stable, not concrete, and prone to obfuscation and mutation, it is needed to present a method call by its inner implementation rather than its superficial names.

Category	Metric	$UHG_{structure}$	UHG_{node}	UHG_{name}	UHG
delete	precision	0.37	0.63	0.64	0.57
	recall	0.31	0.79	0.70	0.89
	f1-score	0.34	**0.70**	0.67	0.69
login	precision	0.35	0.78	0.71	0.78
	recall	0.64	0.77	0.77	0.77
	f1-score	0.45	0.78	0.74	**0.78**
logout	precision	0.46	1.00	0.98	1.00
	recall	0.63	0.96	0.95	0.95
	f1-score	0.54	**0.98**	0.96	0.98
send	precision	0.47	0.78	0.66	0.79
	recall	0.44	0.66	0.71	0.67
	f1-score	0.45	0.72	0.68	**0.72**
search	precision	0.46	0.83	0.67	0.81
	recall	0.19	0.65	0.57	0.67
	f1-score	0.27	0.73	0.62	**0.73**
download	precision	0.34	0.87	0.83	0.88
	recall	0.30	0.85	0.78	0.85
	f1-score	0.32	0.86	0.80	**0.86**
share	precision	0.53	0.92	0.97	0.99
	recall	0.67	0.92	0.90	0.88
	f1-score	0.59	0.92	0.93	**0.93**
save	precision	0.48	0.64	0.62	0.72
	recall	0.19	0.75	0.65	0.70
	f1-score	0.28	0.69	0.63	**0.71**

TABLE V: Effectiveness of different UHG embeddings ($UHG_{structure/node}$ represent for embeddings that only retain graph structural and node features, respectively. In UHG_{name}, we embed UHG based on method names instead of Opcodes).

E. Security Implications of UI-CTX

UI widgets are popular attack surfaces for threats like counterfeiting [53], permission misuse [3], [4], and privacy leakage [5], [7]. UI-CTX is proposed to capture and represent the intended behaviors behind general UI widgets to provide in-depth and decisive semantics for security analysis and threat mitigation, such as anomaly widget detection.

In-depth widget analysis. Existing solutions primarily focus on describing widget behaviors with clear and specific appearances [3], [4], [8], [13]. Intuitively, the more specific a widget's appearance, the easier it is to analyze its behavior. For example, *login* buttons are likely associated with login functionality, making it straightforward to determine their behavior by examining the triggered APIs or permissions. However, many widgets are ambiguous and lack clear UI semantics, making it difficult to determine their intended functionality. For instance, a *next* button, due to its vague UI semantics, may trigger various APIs or permissions to perform different actions. Such ambiguous widgets are overlooked by prior researches. In contrast, UI-CTX extracts the code context for all widgets accessible within the app's codebase, without any assumptions or prerequisites on their UI-layer appearance. It represents widget behavior using UHG, a graph-based representation where edges capture interactions between calls, and each node encapsulates instruction-level operations within a single method. This enables more comprehensive widget analysis by providing decisive semantics, extending the

App sha-256 prefix	Login button functionality	Detailed description	Permission I	Permission L
0FA72E3C	Login (SSO)	Request login via SSO	✓	
1BE86C35	Login (local)	Compare input with a value		
1C9E007F	Login (web)	Get cookie via Apache	✓	
2BEB633E	Login (web)	Get cookie via Signpost	✓	
3C216A25	Login (SQL)	Validate secret via H2		
5B2ADCAF	Login (web)	Get cookie and location	✓	✓
1B2E8C76	Phishing	Send privacy via email	✓	
7AF4A6E2	Phishing	Send privacy via email	✓	

TABLE VI: Functionalities and triggered permissions of login buttons in different apps. I: Internet, L: Location.

analysis beyond specific icons or buttons to all widgets that interact with users and trigger code functionalities.

To illustrate this, we analyze the functionality of a *next* button in a real-world malicious app [54]. As shown in the UHG in Figure 10, this button initiates a series of privacy-stealing actions, including collecting phone numbers, bank account names, and passwords, zipping them into a file, and sending them via HTTP and email. The code context semantics provided by UI-CTX allow us to precisely identify the malicious behavior behind the *next* button, which remains obscured and under-analyzed by previous approaches.

Anomaly widget detection. UI widgets with similar visual appearance are expected to perform similar functionalities. For instance, buttons with the same "login" text should be used for user authentication. In Table VI, we list the functionalities and triggered permissions of eight *login* buttons from different apps. As shown, there are multiple ways to implement the login functionality in Android apps, such as using single sign-on (SSO) interfaces provided by widely used social platforms or conducting user cookie synchronization supported by TPL APIs (*e.g.,* Apache and Signpost). All these apps launch interactions with remote web servers for logon operations, which require the same `INTERNET` permission. A benign login button may also trigger more permissions (*e.g.,* using `ACCESS_FINE_LOCATION` to recommend delicacies in the neighborhoods in a takeaway app) or fewer permissions (*e.g.,* login by simply comparing the input password with a specific value stored locally, or by querying user information from a local SQL database). However, malicious *login* buttons can steal user privacy by emailing the username and password to the adversary, requiring the same `INTERNET` permission as the benign ones. It is hard to distinguish such anomaly functionalities from normal online logins by permissions invoked by widgets. For example, the UI-permission pair $< login, \texttt{INTERNET} >$ is regarded by DEEPINTENT [4] as benign, thus the phishing widgets would bypass the detection.

Nevertheless, differences still exist in the code contexts between phishing and benign widgets. For example, a normal web-based login request checks the server's return code after sending a message to confirm the user's login status, while the phishing apps send mail messages out without checking the response. As we depict the fine-grained code contexts of a widget in a UHG, widgets with different functionalities are well-separated with considerable margins, which is crucial for anomaly detection and malicious behavior identification.

Step	(i)	(ii)	(iii)	(iv)
Average seconds	1.35	11.41	0.59	8.35
Max seconds	4.52	182.02	3.26	370.02

TABLE VII: Time cost of different steps in app knowledge extraction and UHG construction. (i):UI resource parsing, (ii): code graph building, (iii): event handler pinpointing, (iv): UI context refining and summarization.

F. Performance Overhead

To evaluate the performance overhead of UI-CTX, we measure the time cost and storage overhead. All the experiments are conducted on a Linux server with Intel® Xeon® E5-2660 v4 @ 2.00 GHz and 64 GB memory. The operating system is Red Hat 8.5.0-18. Table VII illustrates the time cost of different steps for app knowledge extraction and UHG construction. The majority of the time cost is for the code graph building (ii) and UI context refining and summarization (iv). UI-CTX's efficiency can be further improved, because each step in it supports running in multiprocessing (i, ii and iii) or multithreading (iv). UI-CTX's behavior investigation phase is much faster. It takes 0.02 seconds to get the embedding of a UHG on average (max value: 0.70 seconds). The clustering analysis on our benchmark finishes in 0.20 seconds.

The storage overhead of UI-CTX mainly comes from two databases holding all the graph nodes and edges extracted from apps. For an app, it takes on average 497.91 KB and up to 14.71 MB to storage graph nodes, on average 285.90 KB and up to 36.90 MB to storage graph edges. For the UHG database, the average and max on-disk size for an app is 27.74 KB and 5.92 MB, respectively. In total, we use 10.29 GB to store all graph entities (6.53 GB for nodes and 3.75 GB for edges) and 372.95 MB to save UHGs for all apps.

V. DISCUSSION

UHG embedding methods. There are many learning-based methods, such as graph neural network (GNN) [55], [56] and graph2vec [57], to embed graphs. However, these techniques are computationally intensive, especially for large graphs [58]. Moreover, their performance is not always interpretable and is heavily influenced by the quality of the training dataset. For instance, DEEPINTENT [4] requires manually labeling the training dataset and lacks generalization to new widgets.

In UI-CTX, we simply *present* a UHG by aggregating its structural and node features rather than *predicting* the graph embedding with some uncertainty. Evaluation results show that, despite employing a lightweight graph embedding approach, the embedding preserves sufficient code context semantics to represent and distinguish subtle widget functionalities that cannot be captured by more coarse-grained widget features (*e.g.,* UI appearance). We can expect that if a high-quality labeled dataset for a specific task is available, leveraging learning models that are powerful in distilling

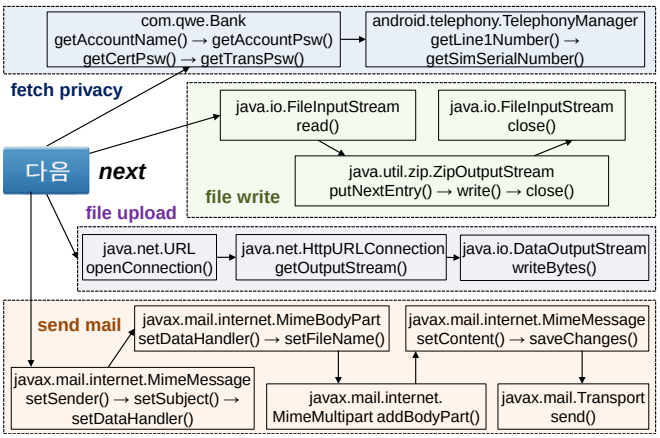


Fig. 10: Detected functionality for a "next" ImageView.

correlations between data and their labels could lead to better performance on specific analysis tasks.

Threat to validity. Malicious app behaviors are not exclusively implemented through UI. Instead of aiming to provide an end-to-end malware detection mechanism, UI-CTX focuses on enhancing the understanding of the functionalities behind UI widgets, offering a unique perspective for analyzing app behaviors such as phishing attempts.

UI-CTX extracts code contexts of UI widgets based on static analysis, which is superior in speed and scalability compared to dynamic methods. To mitigate the inherent limitations of static analysis like over-approximation, UI-CTX examines the entry points of widget code contexts and code branches to identify a subset of code contexts that are would be *truly triggered* at dynamic app runtime. Despite these efforts, UI-CTX cannot always capture precise and concise widget functionalities. For instance, code analysis tools, like FLOWDROID [34] and SOOT [35] used in our approach, may fail when dealing with apps with packed or encrypted source code [21], [59]. Analyzing UI widget functionalities for such apps remains a challenging subject for future research. Besides, as shown in our evaluation, LIBRADAR [36], the external tool we used for TPL/SDK detection, cannot identify all the TPLs/SDKs in app codebase, this will impact the summarization performance of UI-CTX. To solve this, we can update the TPL/SDK list or borrow the capacity from more advanced and recent library detection methods [60], [61].

VI. RELATED WORK

UI Appearance Investigation. As the first impression of an app, UI appearance is a crucial aspect in describing app functionality and enhancing user experience. Investigating UI appearance is the cornerstone of many app analysis tasks, such as detecting similar apps [62]–[66] and identifying similar UI layouts [53], [67]–[70]. For example, Malisa et al. [67] calculate perceptual hashing values of UI screenshot images to facilitate the detection of repackaging apps. DROIDEAGLE [68] represents UI appearance as layout-tree structures and uses a tree edit distance algorithm to measure the similarity between UI widgets. UIHASH [53] abstracts and encodes UI controls across screen regions, then checks the pairwise UI similarity with a Siamese network. However, the visual appearance itself is not sufficient to reveal the real functionality behind UI widgets. Whether a UI widget triggers its functionality as users expected, or how the widget fulfills its functionality remains undetermined and obscure to app users and analysts.

UI Intention Analysis. UI widgets provide an intuitive way for users to interact and trigger specific operational logic. Additionally, these widgets often use and manipulate users' sensitive data, such as location and contacts, posing potential threats to user privacy. Understanding the true intentions behind UI widgets is crucial for avoiding harmful behaviors that violate user expectations. Recent studies have explored the intentions of widgets from different perspectives, such as UI appearance [8], [9], permissions [3], [4], and data flows [5], [71]. For example, ICONINTENT [8] utilizes computer vision and natural language processing to analyze the appearance of icons (*e.g.,* shape and textual content) to investigate their intents and classify them into different categories. DEEPINTENT [4] and DROIDGEM [3] represent widget intentions through permissions and employ deep neural networks to predict the permissions required by UI widgets. Additionally, FLOWCOG [5], SUPOR [71] and UIPICKER [7] conduct taint analysis to analyze information leakage from UI widgets.

UI-based Forensics. Integrating UI knowledge can significantly enhance the accuracy and visibility of security forensics. For instance, UISCOPE [72] attributes system events to high-level UI widgets and UI events, providing improved visibility for attack forensics (*e.g.,* investigating remote code execution) and addressing the dependence explosion problem when auditing syslog. Similarly, TESEC [73] proposes an attack forensics method for web services, identifying UI elements intruders utilize to facilitate attack interception and web application fixing. These works are based on the fundamental observation that the UI in GUI applications is often the entry point of attacks; therefore, analyzing UI widgets is crucial for enhancing the accuracy of security analysis tasks.

VII. CONCLUSION

Understanding the functionalities of UI weights provides a fine-grained view to inspect app behavior. In this paper, we propose UI-CTX, an automated approach to analyze UI widget functionalities and identify their patterns. We identify three challenges faced by prior works in finding the correct widget code contexts, focusing on widget functionality while not being distracted by tremendous external contexts, and representing concrete code semantics. UI-CTX addresses these challenges by conducting a backward code analysis flow, summarizing external APIs, and leveraging instruction-level semantics that can reflect app runtime behavior, respectively. Our experimental results show that UI-CTX can accurately extract and represent UI widgets' intended functionalities compared to state-of-the-art widget representations.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable feedback and insightful suggestions. This work is supported by the National Research Foundation, Singapore, under its Industry Alignment Fund - Pre-positioning (IAF-PP) Funding Initiative, the National Natural Science Foundation of China (No. 62172027 and No. U24B20117), and the Zhejiang Provincial Natural Science Foundation of China (No. LZ23F020013). Any opinions, findings, conclusions, and recommendations presented in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore.

REFERENCES

[1] K. W. Miller, J. Voas, and G. F. Hurlburt, "Byod: Security and privacy considerations," *It Professional*, vol. 14, no. 5, pp. 53–55, 2012.

[2] Elluminati, "Importance of mobile applications in everyday life and hence the businesses," https://www.elluminatiinc.com/importance-of-mobile-application-in-everyday-and-business/.

[3] V. K. Malviya, Y. N. Tun, C. W. Leow, A. T. Xynyn, L. K. Shar, and L. Jiang, "Fine-grained in-context permission classification for android apps using control-flow graph embedding," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1225–1237.

[4] 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 *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 2421–2436.

[5] X. Pan, Y. Cao, X. Du, B. He, G. Fang, R. Shao, and Y. Chen, "FlowCog: Context-aware semantics extraction and analysis of information flow leaks in android apps," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1669–1685.

[6] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang, "Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing," in *Proceedings of the 2012 ACM conference on ubiquitous computing*, 2012, pp. 501–510.

[7] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "Uipicker: User-input privacy identification in mobile applications," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[8] X. Xiao, X. Wang, Z. Cao, H. Wang, and P. Gao, "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, 2019, pp. 257–268.

[9] L. Li, R. Wang, X. Zhan, Y. Wang, C. Gao, S. Wang, and Y. Liu, "What you see is what you get? it is not the case! detecting misleading icons for mobile applications," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 538–550.

[10] S. Chen, L. Fan, C. Chen, M. Xue, Y. Liu, and L. Xu, "Gui-squatting attack: Automated generation of android phishing apps," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2019.

[11] A. Rountev and D. Yan, "Static reference analysis for gui objects in android software," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

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

[13] S. Yang, Y. Wang, Y. Yao, H. Wang, Y. Ye, and X. Xiao, "Describectx: context-aware description synthesis for sensitive behaviors in mobile apps," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 685–697.

[14] Google, "Activity," https://developer.android.com/reference/android/app/Activity.

[15] V. Ristić, "How to use android javamail api to send emails," https://mailtrap.io/blog/android-javamail-api/.

[16] Google, "Handle click events," https://developer.android.com/develop/ui/views/components/menus/#PopupEvents.

[17] 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," in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[18] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Maldozer: Automatic framework for android malware detection using deep learning," *Digital investigation*, vol. 24, pp. S48–S59, 2018.

[19] E. B. Karbab and M. Debbabi, "Petadroid: Adaptive android malware detection using deep learning," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2021, pp. 319–340.

[20] "ProGuard," https://github.com/Guardsquare/proguard.

[21] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *IEEE Symposium on Security and Privacy (SP)*, 2015.

[22] G. You, G. Kim, S.-j. Cho, and H. Han, "A comparative study on optimization, obfuscation, and deobfuscation tools in android." *J. Internet Serv. Inf. Secur.*, vol. 11, no. 1, pp. 2–15, 2021.

[23] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *Security and Privacy in Communication Networks: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part I*. Springer, 2018, pp. 172–192.

[24] "Androguard," https://github.com/androguard/androguard.

[25] Sable Research Group, "android-platforms," https://github.com/Sable/android-platforms.

[26] Google, "LayoutInflater," https://developer.android.com/reference/android/view/LayoutInflater.

[27] ——, "Shrink, obfuscate, and optimize your app," https://developer.android.com/build/shrink-code.

[28] ——, "Dalvik bytecode format," https://source.android.com/docs/core/runtime/dalvik-bytecode.

[29] C. Gao, M. Cai, S. Yin, G. Huang, H. Li, W. Yuan, and X. Luo, "Obfuscation-resilient android malware analysis based on complementary features," *IEEE Transactions on Information Forensics and Security (TIFS)*, 2023.

[30] J. Zeng, X. Wang, J. Liu, Y. Chen, Z. Liang, T.-S. Chua, and Z. L. Chua, "Shadewatcher: Recommendation-guided cyber threat analysis using system audit records," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 489–506.

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

[32] N. G. Trillos, F. Hoffmann, and B. Hosseini, "Geometric structure of graph laplacian embeddings," *Journal of Machine Learning Research (JMLR)*, vol. 22, no. 63, pp. 1–55, 2021.

[33] X.-D. Zhang, "The laplacian eigenvalues of graphs: a survey," *arXiv preprint arXiv:1111.2897*, 2011.

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

[35] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[36] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th international conference on software engineering companion*, 2016, pp. 653–656.

[37] Google, "dexlib2," https://android.googlesource.com/platform/external/smali/+/144951a/dexlib2/src/main/java/org/jf/dexlib2.

[38] "SciPy," https://scipy.org/.

[39] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 468–471.

[40] J. Liu, J. Zeng, F. Pierazzi, L. Cavallaro, and Z. Liang, "Unraveling the key of machine learning solutions for android malware detection," *arXiv preprint arXiv:2402.02953*, 2024.

[41] Y. He, Y. Liu, L. Wu, Z. Yang, K. Ren, and Z. Qin, "MsDroid: Identifying malicious snippets for android malware detection," *IEEE*

Transactions on Dependable and Secure Computing (TDSC), vol. 20, no. 3, pp. 2025–2039, 2022.
[42] "VirusTotal," https://www.virustotal.com/.
[43] Google, "Accessibility," https://developer.android.com/design/ui/mobile/guides/foundations/accessibility.
[44] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *2015 IEEE 39th annual computer software and applications conference (COMPSAC)*, vol. 2. IEEE, 2015, pp. 422–433.
[45] M. Alecci, J. Samhi, T. F. Bissyandé, and J. Klein, "Revisiting android app categorization," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–12.
[46] G. Forman, "A pitfall and solution in multi-class feature selection for text classification," in *International conference on machine learning (ICML)*, 2004, p. 38.
[47] Z. Lipton, Y.-X. Wang, and A. Smola, "Detecting and correcting for label shift with black box predictors," in *International conference on machine learning (ICML)*. PMLR, 2018, pp. 3122–3130.
[48] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3971–3988.
[49] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security (CCS)*, 2012.
[50] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics." in *Network and Distributed System Security Symposium (NDSS)*, 2021.
[51] A. Rosenberg and J. Hirschberg, "V-measure: A conditional entropy-based external cluster evaluation measure," in *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, 2007, pp. 410–420.
[52] J. M. Santos and M. Embrechts, "On the use of the adjusted rand index as a metric for evaluating supervised classification," in *International conference on artificial neural networks (ICANN)*. Springer, 2009.
[53] J. Li, J. Mao, J. Zeng, Q. Lin, S. Feng, and Z. Liang, "UIHash: Detecting similar android uis through grid-based visual appearance representation," in *33nd USENIX Security Symposium (USENIX Security 24)*, 2024.
[54] VirusShare, https://virusshare.com/file?8f073e57b619c9f66a497ca8a1516ad7ad21159692da16cf6ee60cb788c0fb5c.
[55] J. Liu, J. Zeng, X. Wang, and Z. Liang, "Learning graph-based code representations for source-level functional similarity detection," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 345–357.
[56] J. Liu, J. Zeng, X. Wang, K. Ji, and Z. Liang, "Tell: log level suggestions via modeling multi-level code block information," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022, pp. 27–38.
[57] F. Yang, J. Xu, C. Xiong, Z. Li, and K. Zhang, "PROGRAPHER: An anomaly detection system based on provenance graph embedding," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.
[58] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021, pp. 551–568.
[59] Z. Dong, H. Liu, L. Wang, X. Luo, Y. Guo, G. Xu, X. Xiao, and H. Wang, "What did you pack in my app? a systematic analysis of commercial android packers," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE/ESEC)*, 2022.
[60] X. Liu, Z. Jin, J. Liu, W. Liu, X. Wang, and Q. Liu, "ANDetect: A third-party ad network libraries detection framework for android applications," in *Proceedings of the 39th Annual Computer Security Applications Conference (ACSAC)*, 2023, pp. 98–112.
[61] Y. Wu, C. Sun, D. Zeng, G. Tan, S. Ma, and P. Wang, "LibScan: Towards more precise Third-Party library identification for android applications," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3385–3402.
[62] S. Jiao, Y. Cheng, L. Ying, P. Su, and D. Feng, "A rapid and scalable method for android application repackaging detection," in *Information Security Practice and Experience: 11th International Conference, ISPEC 2015, Beijing, China, May 5-8, 2015, Proceedings*. Springer, 2015, pp. 349–364.
[63] F. Lyu, Y. Lin, and J. Yang, "An efficient and packing-resilient two-phase android cloned application detection approach," *Mobile Information Systems*, 2017.
[64] S. Yue, W. Feng, J. Ma, Y. Jiang, X. Tao, C. Xu, and J. Lu, "RepDroid: an automated tool for Android application repackaging detection," in *IEEE/ACM International Conference on Program Comprehension (ICPR)*, 2017.
[65] T. Nguyen, J. T. McDonald, W. B. Glisson, and T. R. Andel, "Detecting repackaged android applications using perceptual hashing," in *Proceedings of the 53rd Hawaii International Conference on System Sciences*, 2020.
[66] N. Karunanayake, J. Rajasegaran, A. Gunathillake, S. Seneviratne, and G. Jourjon, "A multi-modal neural embeddings approach for detecting mobile counterfeit apps: A case study on google play store," *IEEE Transactions on Mobile Computing (TMC)*, 2020.
[67] L. Malisa, K. Kostiainen, M. Och, and S. Capkun, "Mobile application impersonation detection using dynamic user interface extraction," in *European Symposium on Research in Computer Security (ESORICS)*, 2016.
[68] M. Sun, M. Li, and J. C. Lui, "DroidEagle: Seamless detection of visually similar android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2015, pp. 1–12.
[69] L. Malisa, "Security of user interfaces: Attacks and countermeasures," Ph.D. dissertation, ETH Zurich, 2017.
[70] A. G. Patil, M. Li, M. Fisher, M. Savva, and H. Zhang, "Layoutgmn: Neural graph matching for structural layout similarity," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.
[71] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "SUPOR: Precise and scalable sensitive user input detection for android apps," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 977–992.
[72] R. Yang, S. Ma, H. Xu, X. Zhang, and Y. Chen, "UIScope: Accurate, instrumentation-free, and visible attack investigation for gui applications." in *Network and Distributed System Security Symposium (NDSS)*, 2020.
[73] R. Wang, Y. Peng, Y. Sun, X. Zhang, H. Wan, and X. Zhao, "TeSec: Accurate server-side attack investigation for web applications," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2799–2816.
[74] Google, "Thread," https://developer.android.com/reference/java/lang/Thread.

APPENDIX A
DETAILS OF THREAD CALLBACK IDENTIFICATION

In addition to handling multiple events, Android apps can manage multiple threads. Similar to event listeners having handler callbacks, each thread has its corresponding runnable callback (*e.g.,* `run`). However, just as a single widget can be mistakenly linked to multiple unrelated event handlers (*i.e.,* the widget-event pair overestimating issue in Section II-C), existing methods overlook the overestimation problem caused by apps' multi-threading. For example, if only a `Thread.start` call is given without further code context, static analysis cannot determine which runnable is triggered, thus all runnable code blocks will be regarded as "implicit" reachable.

Specifically, every thread is managed by a named or anonymous class, either as a subclass of `Thread` or implementing the `Runnable` interface [74]. Accordingly, to solve the correct runnable callback for a thread, UI-CTX handles both cases. First, it conducts backward dataflow tracking (similar to event listener identification) from the `Thread.start` callsite to determine the subclass of `Thread`, tracing back to the instance creation of the object that calls `Thread.start`. If the results remain unsolved, it checks the class constructor

Algorithm 1: UHG summarization

Input: UI property graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{X})$, where node feature $x_v \in \mathbf{X}$ for $v \in \mathcal{V}$.
Output: Summarized UHG $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathbf{X}')$.

$\mathcal{V}' = \mathcal{V}$, $\mathcal{E}' = \mathcal{E}$;
visited $= \emptyset$; // track processed nodes
cache $= \emptyset$; // cache aggregated node features
foreach *API node* $v_a \in \mathcal{V}$ **do**
  $\mathbf{X}[v_a] \leftarrow$ `Aggregate`$(v_a, \mathcal{G}, cache, visited)$;
$\mathbf{X}' \leftarrow \{x \mid v \in \mathcal{V}' \mid x_v = x \in \mathbf{X}\}$;
return $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathbf{X}')$;

Function `Aggregate`$(v, \mathcal{G}, cache, visited)$ **:**
  if v *is a leaf node* **then**
    return $\mathbf{X}[v]$;
  if $v \in cache$ **then**
    return $cache[v]$;
  $h_{\text{sum}} \leftarrow \mathbf{X}[v]$;
  foreach *child* u *of* v *in* $\mathcal{G}$ **do**
    if $u \notin visited$ **then**
      $h_{\text{sum}} \leftarrow h_{\text{sum}}$ + `Aggregate`$(u, \mathcal{G}, h, cache, visited)$;
    $visited$.`add`(u);
  $cache[v] \leftarrow h_{\text{sum}}$;
  for *each parent* p *of* v *in* $\mathcal{G}$ **do**
    $\mathcal{E}' \leftarrow \mathcal{E}' \setminus \{(p, v)\}$;
  for *each child* u *of* v *in* $\mathcal{G}$ **do**
    $\mathcal{E}' \leftarrow \mathcal{E}' \setminus \{(v, u)\}$;
  $\mathcal{V}' \leftarrow \mathcal{V}' \setminus \{u \mid u \in visited\}$;
  $\mathcal{E}' \leftarrow \mathcal{E}' \cup \{(v, u) \mid u \in visited\}$;
  return h_{sum};

API hop	∞	6	5	4	3	2	1
% of Node	100	61.7	56.0	49.8	42.7	36.3	**32.1**
% of Edge	100	52.5	47.2	41.9	36.5	31.3	**27.9**
F1-score	0.76	0.77	0.78	0.79	0.79	0.80	**0.81**

TABLE VIII: Average nodes, edges, and macro F1-score when summarizing API in k-hop neighborhoods.

and identifies the class of the `Runnable` object. Finally, UI-CTX links the thread initialization site to the method $class$: $void\ run()$, where $class$ refers to the resolved class.

APPENDIX B
DETAILS ON API SUMMARIZATION

We outline the detailed procedures for SDK/TPL API summarization in Algorithm 1, with the corresponding code context reduction and clustering performance results presented in Table VIII. The results demonstrate that API summarization significantly reduces UHG complexity while preserving its ability to describe widget functionality.

APPENDIX C
DETAILS OF CLUSTERING PERFORMANCE METRICS

We provide brief explanations of the metrics used for clustering performance assessment as below:

Cluster	Widget Functionality
delete	Remove data from a list (*e.g.,* `ArrayAdapter`)
download	Parse an external URL and navigate to it
login-1	Transfer user credential to a secondary page to process login
login-2	Encrypt and send user credential, check and save login info
logout-1	Iteratively search for user data, then delete it
logout-2	Clear saved data and finish the current activity
save	Get a set of values, and store them by `ContentValues.put`
search	Search data by operating a database cursor
send	Get input text and send it via inter-component communication (ICC)
share	Load a file URI, and launch the Android Sharesheet

TABLE IX: Widget Functionality Description.

- *Homogeneity* $h = 1 - H(C|K)/H(C)$: measures the extent to which each cluster contains only samples of a single class. A clustering result satisfies homogeneity if all of its clusters contain only data points that are members of a single class. $H(C)$ and $H(K)$ are the entropy of the class distribution and the cluster distribution, respectively.
- *Completeness* $c = 1 - H(K|C)/H(K)$: measures the extent to which all samples of a given class are assigned to the same cluster. A clustering result satisfies completeness if all the data points that are members of a given class are elements of the same cluster.
- *V-measure* [51] $v = 2 \cdot h \cdot c/(h + c)$: the harmonic mean of homogeneity and completeness, providing a balanced measure between the two.
- *Adjusted Rand Index (ARI)* [52]: measures the similarity between two clustering results by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusters, adjusted for the chance grouping of elements. ARI is given by $\text{ARI} = (\text{RI} - \mathbb{E}[\text{RI}])/(\max(\text{RI}) - \mathbb{E}[\text{RI}])$, where RI is the Rand Index, and $\mathbb{E}[\text{RI}]$ is the expected Rand Index of random labeling. As ARI is label-based, it directly corresponds to the difference between the predicted and true clusters.

APPENDIX D
WIDGET FUNCTIONALITIES IN THE VISUALIZED CLUSTERS

We manually review the 10 clusters visualized in Figure 7 and summarize their widget functionality descriptions in Table IX. The results indicate that similar behaviors can be implemented in various ways. For instance, a login widget might simply pass the username to a secondary page that requests the user's password to complete the login process. Alternatively, apps may perform login by encrypting user credentials, checking the login status, and saving the corresponding user data.