DARPA: Combating Asymmetric Dark UI Patterns on Android with Run-time View Decorator

Zhaoxin Cai[†], Yuhong Nan*[†], Xueqiang Wang[‡], Mengyi Long[†], Qihua Ou[†], Min Yang[§] and Zibin Zheng[†]
[†]Sun Yat-sen University, Guangzhou, Guangdong, China
[‡]University of Central Florid, Orlando, Florida, USA
[§]Fudan University, Shanghai, China
caizhx8@mail2.sysu.edu.cn, nanyh@mail.sysu.edu.cn, xueqiang.wang@ucf.edu,
{longmy5, ouqh}@mail2.sysu.edu.cn, m_yang@fudan.edu.cn, zhzibin@mail.sysu.edu.cn

***Abstract*—It has been extensively discussed that online services, such as shopping websites, may exploit dark user interface (UI) patterns to mislead users into performing unwanted and even harmful activities on the UI, e.g., subscribing to recurring purchases unknowingly. Most recently, the growing popularity of mobile platforms has led to an ever-extending reach of dark UI patterns in mobile apps, leading to security and privacy risks to end users. A systematic study of such patterns, including how to detect and mitigate them on mobile platforms, unfortunately, has not been conducted. In this paper, we fill the research gap by investigating the dark UI patterns in mobile apps. Specifically, we show the prevalence of the asymmetric dark UI patterns (AUI) in real-world apps, and reveal their risks by characterizing the AUI (e.g., subjects, hosts, and patterns). Then, through user studies, we demonstrate the demand for effective solutions to mitigate the potential risks of AUI. To meet the needs, we propose *DARPA* – an end-to-end and generic CV-based solution to identify AUIs at run-time and mitigate the risks by highlighting the AUIs with run-time UI decoration. Our evaluation shows that *DARPA* is highly accurate and introduces negligible overhead. Additionally, running *DARPA* does not require any modifications to the apps being analyzed and to the operating system.**

***Index Terms*—mobile security, usable security, dark UI pattern**

I. Introduction

Prior discussions [26], [43], [45], [52] highlighted the prevalence of *dark user interface (UI) patterns* in online services. Using the dark UI patterns, online service providers may gain illicit benefits by misleading users into performing unwanted or even harmful activities, e.g., deceiving the customers of a shopping website into signing up for recurring purchases with a hidden subscription [43]. To help evaluate the risks, prior studies investigated the taxonomy of dark UI patterns in a variety of online services, e.g., online shopping [43], online gaming [52], and website privacy settings [26], etc.

Dark UI patterns on mobile platforms. With the increasing popularity of mobile platforms, dark UI patterns are extending their reach to mobile apps. An example is shown in Figure 1, the app provides two options for users, i.e., subscribing to a promotion which can get free English-learning courses (the large area around the eye-catching round button), or closing the page and back to the main app content (the close button on the top-right). In order to attract more user clicks and subscriptions, the app makes the user-preferred option (close the page) hardly noticeable by adjusting the color and size of the button, while leaving the subscribe option easily accessible. Even worse, the size of the close button is too small that users who intend to click the close button may accidentally enter the subscription page since the surrounding areas of the close button will trigger the subscription as well. While it might not be malicious, the design of the UI definitely causes harm to app users by negatively impacting the user experience, especially for users with less consciousness or senior users that are visually impaired.

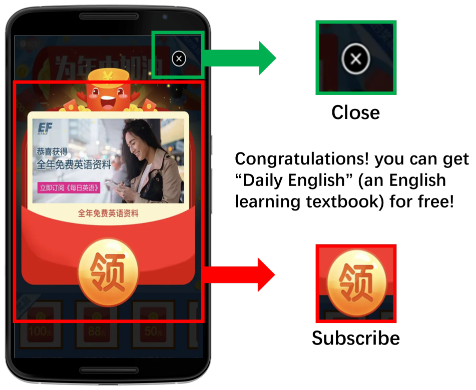

Close
Congratulations! you can get "Daily English" (an English learning textbook) for free!
Subscribe
EF
恭喜获得
全年免费英语资料
立即订阅《每日英语》
全年免费英语资料
领

Fig. 1: An example of Asymmetric Dark UI (AUI) in mobile apps. The main content shows a fake promotion campaign (a region in a red rectangle), and the button to close this page is hidden in the upper-right corner (a region in a green rectangle).

Essentially, the app UI of the above example is designed in an asymmetric way that emphasizes the UI option that benefits the app developers (and associated stakeholders like advertising providers), while understating or even hiding the UI option that is preferred to end users. We call such a pattern the *asymmetric dark UI (AUI)*. Besides impacting app usability, AUI may lead to severe security and privacy consequences, e.g., leading to financial losses due to misleading subscriptions, or bypassing privacy legislation with AUI that targets privacy disclosures.

Understanding AUI in mobile apps. Given the negative impacts of AUI on app users, to the best of our knowledge, a systematic understanding of AUI on mobile platforms,

* Corresponding author: Yuhong Nan

however, has not been conducted before. In this paper, we fill this research gap by investigating AUI in mobile apps.

To perform this study, we first built the first-of-its-kind, high-quality AUI dataset which consists of 1,072 manually verified samples. These samples are collected from 632 real-world popular apps. Based on the AUI dataset, we show the prevalence of AUI in the real-world and reveal their risks by analyzing the characteristics of the collected samples such as subjects, hosts, and patterns (see Section III-A).

Further, we perform a user study to better evaluate user perceptions of AUI. The responses from 165 participants of mobile device users proved that AUI is indeed commonly seen in mobile apps, and its presence introduces significant challenges to the users since many of them frequently fall for the misleading AUI. More importantly, most users expect an effective countermeasure for them to detect AUI in the first place, and highlight (or even automatically opt-in) the preferable UI option for them (see Section III-B).

Combating AUI with *DARPA*. To meet users' needs and enhance their app experience, we propose *DARPA*, an end-to-end solution for combating asymmetric **Dar**k UI **P**atterns on **A**ndroid. A key step for *DARPA* is to effectively identify AUI, which is challenging for several reasons. First, AUI appears not only in the advertising context, but also in many diverse contexts corresponding to the apps' functionalities (e.g., in-app purchases and promotion campaigns). Therefore, previous approaches that focus on detecting UI problems in a particular context (e.g., advertising [31], [39], [40], privacy policy [30]) may not work. Second, mobile apps that host AUI often obfuscate their code and encrypt its behaviors (e.g., traffic), causing challenges to mobile UI analysis techniques that rely on UI layout code [31], [34], [49], [53], [54], and app traffic [36], [40], [55], etc.

To overcome the challenges, we design a *generic* approach using computer vision (CV) techniques, based on the observation that AUI introduces uneven *visual* perceptions to app users. For this purpose, we first train a lightweight CV-model for identifying the asymmetric UI options in AUI. Such a CV-based identification model generalizes to different apps, as the model is capable of identifying UIs that are visually associated with AUI, while ignoring the specifics of UI context (e.g., app category, language, and textual semantics on UI). Besides, this approach is also resistant to obfuscation, as it does not require access to app code.

By deploying the CV-based model to user devices, *DARPA* identifies the AUIs at run-time and mitigates its threat by highlighting them to users with UI decoration. This step is non-trivial as it needs to access the UIs of other apps, which often requires instrumenting the apps being analyzed or modifying the operating system.To circumvent such requirements, we leverage *Accessibility Service* (AS), which is originally provided by the Android platform for assisting device and app users with disabilities [3]. We use AS to capture real-time UIs on the user devices and feed the UIs to the model so as to flag the AUIs. Additionally, since checking all app UIs demand extensive computing resources, we carefully designed our approach, e.g., the proper timing for execution, in order to reduce its overhead. As an ultimate goal, we identify the UI option that is preferred to end users, and highlight the option with UI decorators. Our experiments on real-world apps show that, compared to other alternative solutions (e.g., other CV-models and UI-based analysis), *DARPA* is highly effective in detecting AUI, and introduces low and negligible overhead to end users. Lastly, to benefit future research, we release the AUI dataset used in our research, as well as the source code and artifacts of *DARPA*[1].

In summary, this paper makes the following contributions:

- We perform the first in-depth understanding of the asymmetric dark UI patterns in mobile apps. We demonstrate the high demand for effective solutions to mitigate the potential risks of AUI through a user study.
- We design and implement *DARPA*, the first generic, lightweight framework, which combines CV and Accessibility Service to enhance app usability against AUI for Android devices.
- We perform a thorough evaluation to show *DARPA* is highly effective without incurring observable performance overhead.

II. Background and Motivation

In this section, we present more background knowledge, including key terminologies for describing Asymmetric Dark UI (AUI), the causes and consequences of AUI, as well as related techniques used in our research.

A. Asymmetric Dark UI (AUI) Pattern

A prominent type of dark UI in mobile apps is the visually Asymmetric UI, in which an app intentionally manipulates UI options in favor of developers, while downplaying UI options of interest to users. For ease of description and reference, we give the following two definitions for the UI options.

• **App-guided Option (AGO)**. App-guided Option refers to the UI options, such as buttons or icons, that are beneficial to app developers, and intentionally made visually appealing to app users. Typical examples of AGO are those buttons that trigger advertising pages to add financial gain for app developers. To attract more users to click, the AGOs could be carefully crafted to have a larger clickable area, be centered on the screen, and have a significant color contrast with other content. In some cases, utilizing human psychology, AGO can even be visually deceptive or misleading.

• **User-preferred Option (UPO)**. User-preferred Option represents the UI options that meet the expectations of app users. Examples of UPOs are skipping the advertising page, or proceeding without choosing additional services during product purchase. In the AUI, to avoid the users from selecting this option, a UPO is intentionally designed in a way that is barely noticeable (e.g., has high transparency to the surrounding backgrounds), or is located in an area that requires more effort to reach (e.g., in the corners of the screen).

[1] https://github.com/DARPA-4-AUI

Causes and consequences of AUI. The root cause of AUI is that most apps are provided for free. Therefore, app developers are financially motivated to deploy AUIs to gain more revenues through ways such as subscriptions, advertising clicks, and purchasing rebates. Unfortunately, such a design of AUI may significantly reduce app user experience, and most importantly, result in harm to user interests, e.g., "stealing" money or private information from users with AGOs associated with purchasing and privacy disclosure.

Different from fraud advertisements which are explicitly prohibited by regulators [7], [8], [12], the legitimacy of AUIs often falls into the gray area [33]. Besides, due to the high volume of mobile apps and their dynamic nature (e.g., frequently updated), it is hard for app distributors/markets to practically regulate AUIs with policy enforcement. This situation gets particularly worse in app markets which are less regulated. For example, previous research has shown that apps distributed other than Google Play tend to contain more harmful behaviors [54].

B. Accessibility Service

Accessibility Service (AS) [3] is a service provided by the official Android SDK for app developers to provide interactive help for app users with disabilities (such as visually impaired people). Essentially, AS is a *system service* with high privileges, which can perform many sensitive operations on the Android system. As one of its many powerful features, AS can listen to a set of UI events and act on them. It can also instrument the UIs of running apps. Due to its powerful capability, AS has become a popular attack surface that is abused by malware to obtain sensitive capabilities [35], [46], [50] (e.g., intercepting user credentials). On the other side, it has also been used by previous studies for security enhancement purposes (e.g., fraud ad detection [31], [40]). As we will show further in Section IV, our research also leverages a series of AS capabilities such as monitoring system events and taking screenshots, to improve the user experience on AUI.

C. Scope of our research

Our research focuses on AUI, a specific type of dark UI pattern commonly used in mobile apps. Our user study and prototype of DARPA are mainly based on the analysis of apps in China – one of the largest mobile ecosystems in the world. However, the concepts, techniques and insights introduced in our research are not limited to the Chinese ecosystem, as we analyzed the visual perceptions of AUI without distinguishing the language of the apps. Particularly, *DARPA* is generic and can detect AUIs in different languages, since it does not rely on specific text and its natural-language semantics.

While *DARPA* is implemented for Android, we believe the knowledge contributed by this research is generic and transferable to other mobile platforms, due to the consistent patterns of AUI and the UI similarity across different platforms. Particularly, *DARPA* provides a generic method for visually identifying and labeling AUIs, which is platform-agnostic. Similar to Android, other platforms such as iOS also provide accessibility features [21], [22]. Therefore, adopting *DARPA* to other platforms is technically feasible.

We make several designs to ensure security of *DARPA*, since it detects AUI using privacy-sensitive user screenshots. For example, *DARPA* asks for minimal permissions (e.g., no Internet permission), and stores all screenshots locally and discard them immediately after use (see discussions in Section IV-E). We assume that the supply chain of *DARPA* is not compromised. Thus, adversaries can not directly break the above security designs. We also assume that the Android OS is not compromised, and *DARPA* is always protected by OS security features, such as app-level sandboxing. Considering the minimized attack surfaces and the protections in place, we believe it is difficult for an external adversary to exploit *DARPA* (e.g., via buffer overflow) and perform malicious behaviors.

III. Understanding Asymmetric UI dark Patterns

A. AUIs in the Wild

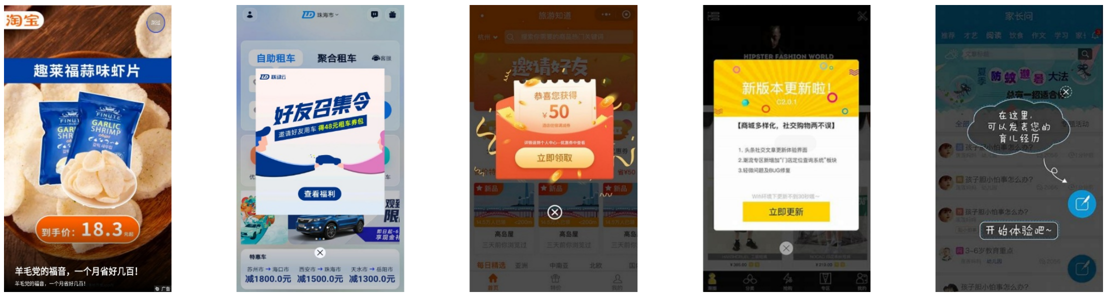
To better understand the patterns of AUIs in the wild, we collect a dataset of real-world apps and an AUI dataset (i.e., screenshots) from these apps through a large-scale and semi-automatic analysis.

- **App dataset** (D_{app}) To build the AUI dataset, we first crawled 632 popular apps from the leaderboard of *Mi Store* [18] (one of the most popular app stores in China) in June 2022. These apps span a variety of categories, including banking, instant messaging, social media, and utilities, etc.
- **AUI dataset** (D_{aui}) We use a mix of automatic testing and manual verification over the apps in D_{app} to collect the AUI dataset. Specifically, we ran each of the apps in D_{app} for 1 minute with Monkey, and automatically gathered all their screenshots. Additionally, to increase coverage of our dataset, we crawled huaban.com [10] – a UX designer website that contains over 100,000 real app screenshots. We gathered a total of 8,855 screenshots, with 7,884 by running the apps and 971 from huaban.com. We further asked 5 researchers to review these screenshots and label those containing AUIs. In total, we obtained 1,072 AUI screenshots that are labeled by all researchers.

TABLE I: Distribution of different types of AUI

AUI Type	Number of instances	Percentage
Advertisement	696	64.9%
Sales promotion	179	16.7%
Lucky money (Red packet)	131	12.2%
App upgrade	43	4.0%
Operation guide	16	1.5%
Feedback request	4	0.4%
Sensitive permission request	3	0.3%
Total	1,072	100%

Subjects of AUI. Table I shows the distribution of different types of AUIs (according to the app context) based on our manual classification. We noticed that most of the AUIs (696, 64.9%) are posted by advertising providers for the purpose of recommending online products, services, and apps to app



(a) Advertisement **(b)** Sales promotion **(c)** Lucky money **(d)** App upgrade **(e)** Operation guide

Fig. 2: Examples of each type of AUI discovered in our research.

users[2]. While these AUIs are often not immediately harmful to users, they can mislead them to perform unexpected activities. As an example shown in Figure 2a, a product is promoted with an extremely low price. However, an interested user could be redirected to a new page that is not aligned with the statements shown in the advertisement. Following the advertisement are the sales promotion AUI, which accounts for 179 (16.7%) AUIs in our dataset. In such AUIs, the apps launch in-app promotion campaigns to promote their own services. For example, a user that fails to click the UPO shown in Figure 2(b) will be redirected to an external page asking him/her to submit personal contact information (e.g., email, phone number) and then subscribe to the promotion. Another popular type of AUIs is the 131 (12.2%) lucky money AUIs. These AUIs are similar to sales promotion, but the app claims users will be rewarded with real money for specific actions they take, e.g., downloading/installing an app, playing a game, registering a new account, etc. Unfortunately, the cash reward could be very difficult to obtain due to various restrictions setup by app vendors. The remaining AUIs are related to the operation guides provided by apps, requesting feedback for the apps or services, asking app users for permissions to carry out security/privacy operations, etc. Figure 2 shows part of the AUI examples for different AUI types[3].

Hosts of AUI. For each AUI, it may either comes from the app itself, or from third-party components which are integrated by the app. Based on the distributions of AUIs in our collected samples, it can be seen that around 35.1% (376 out of 1,072) AUIs are designed by app developers themselves. In addition, for the rest 64.9% AUIs from third-party libraries (i.e., advertisements), the apps hosting these ads may even not aware such ads are hurting user experiences of their customers.

Layout patterns of AUI. We perform a statistical analysis to show the layout patterns of the two essential AUI options, i.e., App-guided Option (AGO) and User-preferred Option (UPO). Our analysis showed that most (94.6%) of the AUIs place the AGO in the central area of the UI. In contrast, 73.1% AUIs with a UPO place the UPO on the corner of the UI (similar to the upper right corner as in Figure 1). For the remaining AUIs, while the UPOs are not in the corner, they are not visually as significant as the AGOs. As shown in Figure 2d, the option that closes an app upgrade request window is much smaller in size compared to the option that upgrades the app (the button in yellow).

[2]As required by regulation, an advertisement must explicitly indicate its identity by placing the term “advertisement” on its ad content. While in most cases, this indicator is barely noticeable by app users due to the very small font size and ambiguous color contrast to the surrounding background, it is sufficient for us to manually classify the screenshot as an advertisement.

[3]More examples for each AUI type can be accessed in our project repository: https://github.com/DARPA-4-AUI/paper_data/tree/master/AUI_examples

B. Understanding User Perceptions to AUI

To better understand user perceptions of AUI, we design a user study by asking about users’ experiences of interacting with AUI. Particularly, the study aims to find out (1) how users think of AUI, (2) how users deal with AUI, and (3) what are the user expectations for alleviating the negative impact of AUI. The feedback collected from the user study confirmed that app user experience is indeed affected by AUI, and it is necessary to provide practical solutions against AUI.

Design of the user study. The study contains three parts that have 12 questions in total. At the beginning of the survey, we present two typical AUI examples to help participants quickly understand the context of our study. Then, we ask the participants “if they feel the two UIs presented are misleading and could lead to unintended clicks” (Q1). Then, we asked the participants whether they have past experiences of clicking the unintended UI option during their everyday use of apps, and if so, how often that happens (Q2).

The second part of the study (Q3-Q5) aims at getting a quantitative measurement about the easiness for users to identify the UI options in AUIs, i.e., accessibility of the AGOs and UPOs. More specifically, we randomly select another three AUI examples and ask users to give accessibility ratings (with a score from 1 to 10) for the AGOs and UPOs shown on these UIs. Here, a higher rating means the UI options can be easily identified, while a lower rating means the options are hard to identify. In addition, we ask app users which scenarios are more likely to cause unintended clicks (Q6), their emotions when the unintended clicks happen (Q7), and users’ experiences of using apps from different countries, e.g., whether they experience the same amount of AUIs while using apps from China and other countries (Q8).

The third part of the study is designed to understand users' expected countermeasures for AUI (Q9-Q12). Specifically, we ask users how they thought the UIs should be properly designed, whether it is necessary for the mobile OS to make the UI options more accessible, and if so, what are users' preferred design choices, i.e., highlight the UI options, or directly skip the UIs to avoid unintended clicks, etc.

Finally, we collect the participants' demographics such as their age range and educational background. Such information helps to better evaluate the impact of AUIs on the general public. Note that we do not collect any personally identifiable information of the participants. Also, this study was reviewed and approved by the ethical review board of our institution.

The survey was conducted anonymously through Wenjuanxing [15], the most popular online survey service platform in China, from November 21, 2022 to November 24, 2022. To ensure the quality of responses collected, we set the survey completion threshold at 90 seconds and above, to avoid any responses from robots and careless participants (Fortunately, all the responses are from real participants and valid). We awarded the participants 0.5 USD for completing the survey. In terms of the distribution of the participants, 74 of the participants are male and the other 91 are female. The majority (76.4%) of the participants are in the age range of 18-35. The questions and the summarized responses of this survey are available at: https://github.com/DARPA-4-AUI/paper_data/tree/master/user_study.

Results and findings. We summarize the following findings from the 165 valid responses. The vast majority (156/165, 94.5%) of the participants feel that the two AUI examples are misleading (Q1). According to the accessibility ratings specified by the participants (Q3-Q5), accessing the AGOs (e.g., opening the advertisement) are significantly easier than visiting the UPOs (e.g., skipping the advertisement). The average rating for AGOs is 7.49 (out of 10), while the average rating for UPOs is only 4.38. Further, 120 (72.7%) of the participants believe that UPOs are at least equally important as AGOs (Q9). Therefore, it looks evident that AUIs introduce asymmetrical user experience to app users, which leads to unexpected challenges for them to access their preferred UI options, i.e., UPOs.

Finding 1: App users strongly agree that AUIs are misleading.

According to the responses from Q2, most participants are affected by such AUIs in their daily app use – 127 (77.0%) participants stated that they often trigger unintended button clicks, 34 (20.6%) participants stated that such cases happened occasionally, and only 4 (2.4%) participants never triggered unintended clicks. In addition, for the participants that trigger such clicks, 83.0% of them (137/165) feel bothered and they want to quickly exit the unintended UI (Q7).

Among the 112 participants that also have experienced using apps from other countries, 86 (76.8%) of them think apps in China tend to have more AUIs (Q8). We suspect this is mainly caused by the fact that Chinese app stores are less regulated, in which there are deceptive and misleading behaviors (e.g., ads) rarely discussed in other countries [7], [8], [12].

Finding 2: AUI indeed brings negative impacts to the usability of apps, particularly for the apps in China.

Finally, most participants felt that it is important to have an effective solution to provide better accessibility against such AUIs. The average rating for having such a solution is 7.64, with 48 participants giving a rating of 9 and above. More than half of the participants expect the solution to highlight the available UI options for them (e.g., making the UPOs easier to identify).

Finding 3: Users expect practical solutions to enhance the accessibility against AUI.

Note that, although we collected responses from over a hundred app users, the above results can still be biased since most participants are relatively young and well-educated (Q13-Q14). For instance, 93.9% of the participants have a bachelor's or equivalent degree. These people tend to be more familiar with mobile devices and thus less affected by AUIs than the other user groups, e.g., senior or visually defective users. Therefore, we believe that, in reality, there can be a much stronger need to have an effective solution for AUIs.

IV. Design of DARPA

Given the above understanding of AUI, we present *DARPA*, an end-to-end solution that detects and highlights the presence of AUI, to meet the users' needs (Section III-B). In this section, we will start with design objectives and an overview of *DARPA*, and then demonstrate how we meet the objectives in the detailed design of *DARPA*.

A. Design Objectives and Overview

Design objectives. In order for *DARPA* to work on normal user devices, we built it with the following objectives in mind:

• Generality. Given the wide variety of AUI as described in Section III-A, *DARPA* should not be tailored towards specific types of mobile apps, or specific types of UI (e.g., advertisement). Rather, *it needs to be capable of detecting any AUI on the user devices in a general manner.*

• Easy Integration. End users should be able to integrate *DARPA* easily to their devices. A straightforward solution that analyzes other apps' UIs may require modifications to the operating system or rewriting the app being analyzed, because of the presence of cross-app isolation. This requirement is hardly met for normal users. Therefore, *we expect them to use DARPA with very minor effort, as if installing an app.*

• Lightweightness. Another important consideration is the overhead of *DARPA*. *We need to minimize the performance overhead to end users so as to ensure their user experiences.*

• Security. *DARPA* can essentially monitor the app UIs of end users, which, if not designed sensibly, may accidentally introduce security issues such as privacy leaks. Therefore, we need to put extra safety measures so that end users are willing to use *DARPA*.

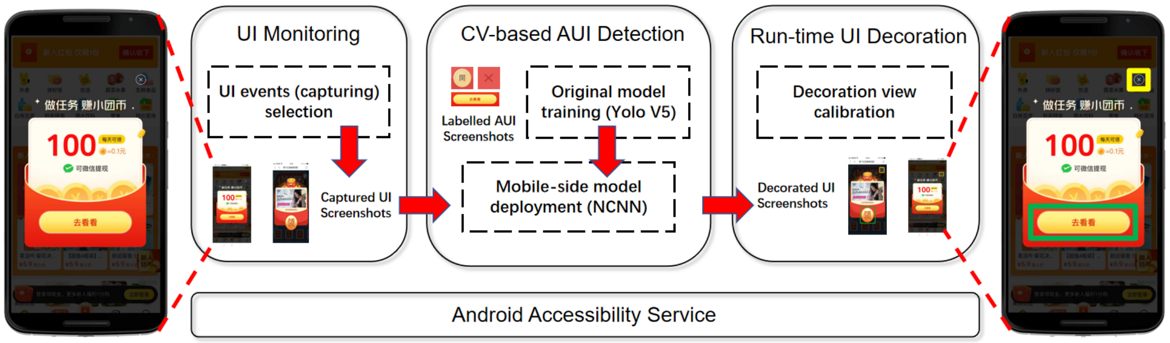

UI Monitoring
UI events (capturing) selection
Captured UI Screenshots
CV-based AUI Detection
Labelled AUI Screenshots
Original model training (Yolo V5)
Mobile-side model deployment (NCNN)
Run-time UI Decoration
Decoration view calibration
Decorated UI Screenshots
Android Accessibility Service

Fig. 3: The overview of *DARPA*.

Design Overview. Figure 3 presents the overview of *DARPA*. *DARPA* works as an individual app that monitors the user interface at runtime. If an AUI is detected, it draws the corresponding UI decorators to highlight user-desired information (e.g., the UPO in AUI). As mentioned earlier, to achieve this, *DARPA* utilizes the Accessibility Service (AS) which is built in by Android for the necessary data access (e.g., UI content parsing) and operation (view decoration). In this way, it does not require any customized modification (i.e., framework-level instrumentation) to the Android system.

In the first module of *DARPA*, we collect the app UIs that may contain AUIs by monitoring the UI-related events. The second module analyzes the app UIs at run-time to determine if they indeed contain any AUI. For this purpose, we adopt a CV-based model. Specifically, with the previously collected AUI samples (D_{aui}) from real apps, we train a YOLOv5 model and port it to Android devices so that it can detect AUIs within app UIs in real-time. The output of this module is the UPO in an AUI. In the last module, we highlight the UPO by applying UI decorators directly on the app screen. In the remainder of this section, we describe each module of *DARPA* in detail and demonstrate how we achieve the design objectives.

B. Collecting App UIs for Analysis

The very first step for detecting AUI is to collect the app UIs that are displayed on the end-user devices. Conventional methods for this task may require modifications to the Android system so that *DARPA* can run as a privileged system component, or rewriting the apps being analyzed so that they can directly emit the UI information. As noted earlier, both methods will cause integration issues for normal users. In this study, we observed that an effective approach could be built on top of a native service supported by Android, i.e., accessibility service (AS), without modifying either the system or the apps. Specifically, AS provides a series of system events that represent app UI updates [2], such as window content changes (`TYPE_WINDOW_CONTENT_CHANGED`) and view focused (`TYPE_VIEW_FOCUSED`). Once an app updates its UI, the corresponding system events will be issued and callbacks will be delivered to any app that subscribes to AS events. Therefore, in *DARPA*, we can subscribe to all UI updates events, and collect app UIs in the form of screenshots after being notified by AS. Using this method, *DARPA* can run as a regular app that interacts with AS via APIs provided by the Android SDK, which greatly simplifies its integration.

However, there is a prominent challenge that prevents *DARPA* from analyzing all UI updates. Our preliminary study shows that any minor changes to a UI may lead to a newly issued UI update event. For example, Android may generate nearly 32 events when running *Taobao* [16], a popular shopping app in China, for a minute. Due to the high frequency of UI update events, *DARPA* may cause significant overhead to user devices. Even worse, our study reveals that filtering the events based on the event types is not feasible as AS events are designed to be generic and not leak any specific UI information, e.g., whether the UI is an AUI or not. Our solution to resolve the challenge is straightforward yet effective: we use *DARPA* to check a subset of UIs that last for a longer period of time, based on the intuition that AUIs often need to gain enough exposure to the app users. For this purpose, we gather an app UI for future analysis if no new UI update event is issued after a cut-off time period (*ct*). As we will evaluate in Section VI-E, a *ct* value of 200ms allows *DARPA* to improve efficiency and achieve good coverage of AUI at the same time.

C. Detecting AUI from App UIs

After collecting an app UI, *DARPA* detects in real-time if the UI represents an AUI that contains misleading UI options (i.e., an AGO). This task is non-trivial given the wide variety of AUIs (see the AUI subjects in Section III-A). Prior studies that rely on app traffic and UI text features [31], [40], as we will evaluate in Section VI-C, are not applicable because of the difficulties to compile a complete list of such features (e.g., resource and view id for advertising UI), and the frequent adoption of obfuscation techniques that make feature extracting challenging. We observe that, regardless of the underlying implementations, the AUIs will always display user options with unique visual patterns, e.g., AGO, for the purpose of misleading the users. Thus, we can design a general approach to identifying AUIs with a CV-based method.

Overall workflow of CV-based detection. We first build a ground-truth AUI dataset from the screenshots of real-world apps. We label the user options (AGO or UPO) on the AUIs and annotate the size and coordinates of the options. Then, we train an object detection model on top of the ground-truth dataset (i.e., D_{aui}). After that, we port the model to Android devices and use it to detect AUIs. The output of the detection is the size, coordinates, and confidence rate of the on-UI objects, i.e., the possibility of whether one UI object corresponds to AGO or UPO. Note that to detect and highlight AUI, one may consider simply identifying and labeling the small "close button" as the UPO. However, this mechanism would introduce a significant number of false positives, as many normal app UIs with an small close option are actually not AUIs (i.e., there is no AGO in such UIs) [4].

Model selection. Object detection has been studied for years and there are various types of models that can potentially be applied to our use case. In this study, we surveyed a series of most commonly used models, such as two-stage detection models (e.g., RCNN) and one-stage detection models (e.g., YOLO, SSD). We first checked whether the models could run on mobile platforms, and, if not, the ease of porting them. We found that most of the models are not natively supported on Android, but there are off-the-shelf frameworks to port them for mobile use. After that, we looked into another important factor that guides our selection – the detection speed. Our analysis shows that YOLO, a one-stage object detection technique that first identifies the bounding boxes of objects and classifies them based on CNN features, has a significantly faster speed than the other alternatives. Thus, we choose YOLOv5 [19], the currently most out-performance implementation version of YOLO built on top of PyTorch framework, in our *DARPA* implementation.

Model porting. Object detection models are often used on GPU servers (or desktops) that are highly performant, but they are not friendly to resource-constrained mobile devices. To resolve the issue, we first train our model on a server and then port it to mobile platforms using a neural network inference framework. In *DARPA*, we adopt the open-source framework proposed by Tencent, *ncnn* [13], which is a state-of-the-art model for neural network computing on mobile devices. Prior research [29] has shown that *ncnn* outperforms other deep learning inference frameworks such as *mnn* [11], *Neo* [42], since it is highly optimized for mobile platforms with ARM CPUs. For example, compared to other deep learning models, *ncnn* generates models with smaller size, and with minimal accuracy loss. Specifically, we first convert the YOLO model generated by the Pytorch framework [48] to ONNX format [1], an open format for interchanging models between ML frameworks. Then, we replace the internal redundant calculations in the model with constants to accelerate its forwarding speed. After that, we convert the model from ONNX format to *ncnn* format, and use it as the final model. We embed the final model in *DARPA* so that it could be invoked on mobile devices. While the current design of *DARPA* implements *ncnn* for model porting, it is not limited to this specific type of deep learning inference framework for our task. Altering *ncnn* to other models with better effectiveness is trivial.

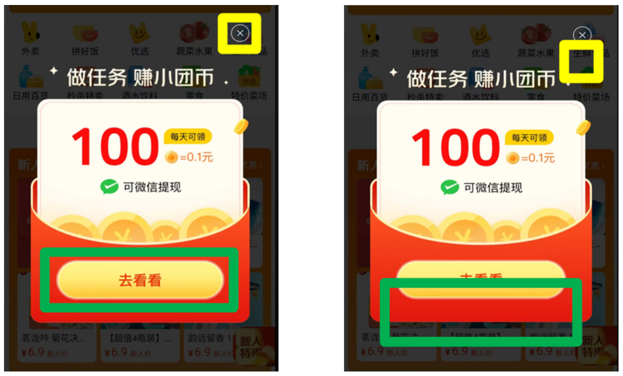
[4]See examples in https://github.com/DARPA-4-AUI/paper_data/tree/master/not_AUI

D. Run-time AUI Decoration

According to the user study (Section III-B), most of the participants expect to have the available AUI options highlighted so that they can identify the options more easily. Upon this request, *DARPA* provides a module to highlight the AUI options with easily noticeable decorators. Essentially, we place an additional decoration view around the bounding boxes of the identified AUI options. By default, the decoration views use a high-contrast color compared to the background color of the AUI options, for the purpose of attracting the attention of app users. We also allow users to customize the shape and color of the decoration view.

In *DARPA*, we first take the screenshots of an app, and detect the AUI options using an object detection method (Section IV-B and IV-C). The output of the model is the AUI options on screen, which contains the coordinates of the options relative to the top/left corner of the *whole screen*. However, when adding the decoration view to the screen, *DARPA* actually expects the coordinates with respect to the app *window* – a section of the screen made available to the app. When the app is in the *full-screen* mode, we can safely place decoration views right at the same coordinates as the bounding boxes of the AUI options. However, it becomes problematic if the app is not in full-screen mode, which is pretty common for Android apps. For example, in Figure 4, the app only occupies part of the screen, with a status bar on top of the app window and a navigation bar underneath. In this case, if we use the same coordinates as the AUI options, the decoration views will be positioned below the actual options, e.g., by an offset of the height of the status bar (see Figure 4(a)). To fix the problem, we need to know the exact offset of the app window and use it to calibrate the decoration view. However, to the best of our knowledge, Android SDK does not provide any API that can directly get the exact offset.

Decoration view calibration. A possible way to retrieve the above offset is to call the `View.getLocationOnScreen` API with the *view* object corresponding to the app window. However, getting the view object at run-time is not possible for *DARPA* since Android restricts access to the memory space of the other apps. In our study, we overcome the challenge by adding an unnoticeable anchor view to the very top/left (i.e., coordinate <0,0>) of the current window. Then we call the `View.getLocationOnScreen` API to check the coordinates (i.e., `offset_x`, `offset_y`) of the anchor view within the screen. For full-screen mode, the `offset_x` and `offset_y` will be 0s; for the cases that the app window occupies part of the screen, the `offset_x` and `offset_y` represent the actual offsets of the app window with respect to the screen. In this way, we can calibrate the decoration view by subtracting the offsets from the coordinates of the AUI window. In addition to decorating AUIs on screen, *DARPA* also provides an alternative option that can automatically bypass those harmful AGO. More specifically, with this option,



(a) Run-time view decoration without calibration. **(b)** Run-time view decoration with calibration

Fig. 4: AUI decoration while the app is not in full-screen mode

DARPA automatically sends a click event to the UPO region and closes the AUI.

E. Security Considerations

As mentioned earlier, security concerns may arise when end users run *DARPA* on their devices, as *DARPA* can take screenshots of other app. In some cases, the screenshots taken may contain sensitive user data such as user profile, chat history, etc. To this end, we take extra measures to address the concerns. Specifically, we put all the modules of *DARPA*, including the CV-model, within an individual Android app. The app does not request any sensitive capabilities (e.g., Internet access and disk access) that allow it to expose sensitive data. Further, the app stores the app screenshots in the app's internal storage, and rinses them immediately after running the CV-model against them. Additionally, the app is not able to update itself (with malicious components), without going through the authorized app updating process, e.g., app submission, review, and OTA updates of the app store. To meet privacy legal compliance, *DARPA* also provides end users with a detailed privacy policy, and obtains user consent to proceed when the app is first executed.

V. IMPLEMENTATION

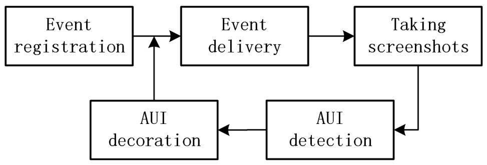


Fig. 5: The life-cycle of *DARPA* during AUI detection.

Life-cycle of *DARPA* runtime. Here we present the implementation of *DARPA* at runtime, particularly its detailed life-cycle during AUI detection and decoration.

- Event registration. As shown in Figure 5, once *DARPA* starts, it first registers all 23 Accessibility Events to monitor potential UI updates of the system. In the meantime, *DARPA* sets up a 200ms delay for event notification, to avoid being overwhelmed by those redundant UI updates not necessary for screenshot taking.
- Event delivery. After event registration, *DARPA* will passively wait for related UI events. More specifically, the OS will announce *DARPA* with the corresponding event code while any change happens on the screen. For example, the event `TYPE_WINDOWS_CHANGED` corresponds to code `0x00400000`.
- Taking screenshots. *DARPA* takes screenshots via the API `AccessibilityService.takeScreenshot`. Note that before taking screenshots, *DARPA* will remove its previous AUI decoration if there is any.
- AUI detection. Each captured screenshot is directly passed to the CV model by Native Development Kit (NDK) [20]. For security considerations, *DARPA* deletes the previously taken screenshot right after the CV model gives the detection results.
- AUI decoration. Based on the obtained coordinates (and offsets) of the identified AUI options, *DARPA* uses the API `android.view.WindowManager.addView` to draw the corresponding bounding box(es) on the screen, and hence highlights the AGO and UPO to users.

Implementation for AUI decoration. Figure 6 shows the simplified code for AUI decoration. In line 2, we get an instance of `android.view.WindowManager` and use it for adding a decoration view to the window. Then, in lines 3-9, we set the dimension and coordinates of the decoration view according to the information of the AUI. In lines 8-9, obtain the coordinates of an AUI option in the main window. Finally, in lines 10 and 11, we create the decoration view and add it to the current window.

```
1  public void decorate(AUI aui, int offset_x, int
       offset_y) {
2      WindowManager wm = (WindowManager) getSystemService
           (Context.WINDOW_SERVICE);
3      WindowManager.LayoutParams lp = new WindowManager.
           LayoutParams();
4      lp.type = aui.type;
5      lp.width = aui.width;
6      lp.height = aui.height;
7      // <aui.x, aui.y> represents the coordinates of an
           AUI option w.r.t the screen, and <lp.x, lp.y>
           represents the coordinates w.r.t the main
           window.
8      lp.x = aui.x - offset_x;
9      lp.y = aui.y - offset_y;
10     ImageView decorateView = new ImageView(this);
11     wm.addView(decorateView, lp);
12 }
```

Fig. 6: Simplified code for run-time AUI decoration

VI. EVALUATION

In this section, we report our evaluation of *DARPA* with a series of experiments. Specifically, we aim to answer the following research questions:

- *RQ1:* What is the overall effectiveness of *DARPA* in detecting AUIs with its CV-based model?

• *RQ2:* Does *DARPA* achieve better detection performance compared with other alternative techniques (models)?

• *RQ3:* What is the performance overhead of *DARPA* when it is deployed on real devices?

• *RQ4:* What is the proper timing for executing *DARPA* on user devices?

A. Evaluation Setup

We train our CV-model on a Ubuntu 20.04 server with 512 GB RAM, 24-core Intel 8260 CPU, and 2 NVIDIA RTX 3090 GPU. Unless otherwise specified, we run all other experiments, e.g., testing against Android apps and deploying *DARPA*, on a Redmi 10 smartphone with Android 11[5].

To train our CV-based model for AUI detection, we use the same AUI dataset (D_{aui}) as elaborated on Section III-A. We split the D_{aui} into the training, validation, and test sets by a ratio of 6:2:2. The distribution of each set (including the *UPO* and *AGO*) is shown in Table II. Note that a screenshot may have more than one *UPO* options (which potentially results in a more confusing user experience given the presence of AUI).

For model training, we label the *AGO* and *UPO* options in all the screenshots, and annotate their bounding boxes following the format of COCO dataset [6] – a specific data labeling format for object identification. In this way, the screenshots are well-labeled and ready to be fed to the CV-model.

TABLE II: Distribution of the ground-truth dataset D_{aui}.

Set Type	AGO	UPO	Total
Training Set	453	657	642
Validation Set	150	223	215
Testing Set	141	222	215
Total	744	1,103	1,072

B. RQ1: Effectiveness in Detecting AUIs

Evaluation metrics. As described in Section IV-C, we train a YOLOv5 model with our ground-truth dataset D_{aui}, for the purpose of identifying the AGO and UPO in the AUIs. This is essentially similar to other object detectors for images. Therefore, we leverage a common evaluation metric, Intersection over Union (IoU), to measure detection accuracy. IoU represents the intersection area (*I*) between the ground-truth bounding box (*G*) and the predicted bounding box (*P*), i.e., $I/(G+P-I)$.

To decide if our model identifies the UI options correctly, we need a specific IoU threshold. In this study, we choose a pretty high threshold, 0.9 (which is also adopted by [28]), since our end-to-end solution requires us to identify the AUIs and the UI option areas with high accuracy. We report a true positive (TP) case once our model identifies a UI option with an IoU over the threshold. Similarly, we report the true negative (TN), false positive (FP), and false negative (FN) cases. Then, we evaluate the effectiveness of our model with three metrics, i.e., *Precision* (TP/(TP+FP)), *Recall* (TP/(TP+FN), and *F1-score* (2*TP/(2*TP+FP+FN)).

[5]*DARPA* is generic to all devices with Android version 11 and above, since the AS module under these versions provides the screenshot API – a key feature utilized by *DARPA*.

TABLE III: Overall effectiveness of *DARPA*.

AUI Type	Precision	Recall	F1-score
UPO	0.901	0.852	0.876
AGO	0.815	0.802	0.808
All	0.858	0.827	0.842

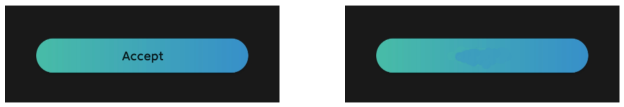
Overall Effectiveness. As the first step, we train and fine-tune the parameters of the YOLOv5 model with the training and validation sets in D_{aui}. Specifically, we use a batch size of 256, and apply the Adam optimizer during training. We also train the model multiple times with varying epochs, e.g., 500, 1,000, etc. At the same time, we observe the validation losses and ensure this process does not incur overfitting. Eventually, we select the model with the best fitness for training 2,500 epochs. We then integrate the model into the Redmi 10 smartphone by porting it by the ncnn inference framework (YOLOv5-ncnn, see Section IV-C). We run YOLOv5-ncnn on the mobile device to determine its effectiveness on the testing set. Our experiment shows that the model can detect AUI with a precision of 85.8%, a recall of 82.7%, and an F-score of 84.2% (Table III). Particularly, the model detects the UPOs, which are preferred by users, with an even higher precision (90.1%) and recall (85.2%).

We manually analyzed the causal of the FNs and FPs. The reason for most FNs is that apps can use UPO-related buttons that are not only small in size, but also of a transparent background. While we were able to identify the textual data on the buttons with a very careful review of the apps, our CV-model failed to capture the visual features of such buttons. Most FPs are indeed associated with UI options that are hardly noticeable, e.g., a small *Add to Cart* button in a UI with bad design. However, we don't think such a UI is indeed an AUI, since there is no such a user-preferred option. We believe many of the FPs and FNs can potentially be eliminated by gathering a larger set of high-quality ground-truth data, with more clear-cut AUIs.

TABLE IV: Effectiveness of the YOLOv5 model.

Model	AUI Type	Precision	Recall	F1-score
YOLOv5 (on Server)	UPO	0.925	0.867	0.895
	AGO	0.837	0.81	0.823
	All	0.881	0.838	0.859
YOLOv5 (with texts masked)	UPO	0.871	0.899	0.885
	AGO	0.882	0.762	0.818
	All	0.877	0.830	0.853

Model migration. To understand whether migrating the model to mobile devices reduces the effectiveness of YOLOv5, we perform a comparison analysis by running YOLOv5 on our server and comparing the results with *DARPA* (i.e., YOLOv5-ncnn migration on the smartphone). As shown in Table IV, the comparison results indicate that compared to the original YOLOv5 model, converting it with ncnn as in *DARPA* introduces negligible losses, i.e., the F-score is decreased by 1.7% (from 85.9% to 84.2%) when running the model on the smartphone.



(a) unmasked AUI **(b)** masked AUI

Fig. 7: The difference between unmasked and masked AUI.

Model generalization to different languages. In addition, as mentioned earlier, since *DARPA* detects AUI based on the visual appearance, it is generic to apps with different languages. To this end, we perform an additional experiment to show this capability in *DARPA*. More specifically, for all the training and testing AUIs used in *DARPA*, we intentionally masked (i.e., blurred) all the texts presented on AGO/UPO (as shown in Figure 7), and we re-trained another YOLOv5 model and checked its effectiveness. As shown in Table IV, our evaluation showed that the performance is nearly the same as the original *DARPA*, meaning that the effectiveness of *DARPA* indeed comes from the visual appearance of UIs, rather than on-UI text such as "cancel", "open" in a specific language.

C. RQ2: Comparison between DARPA and alternative detection techniques

Comparing YOLOv5 to other CV-models. Besides YOLOv5, we can leverage many other alternative CV-models to detect AUIs. To evaluate if YOLOv5 is more suitable for our use case, we compare it to other state-of-the-art object detection algorithms – Faster RCNN and Mask RCNN. Specifically, for both algorithms, we use VGG16 and ResNet50 respectively as the backbones for feature extraction, which essentially leads to four models, i.e., Faster RCNN+VGG16, Faster RCNN+ResNet50, Mask RCNN+VGG16, and Mask RCNN+ResNet50. We train the models on the same ground-truth dataset D_{aui}, and fine-tune their parameters to get optimal performance. We use the default settings for these models, as, according to prior studies [28], the default settings usually generate satisfying results for object detection tasks. Our comparison results (in Table V) show that YOLOv5 outperforms the other models by a large margin for all the evaluation metrics. For example, the best RCNN model (Mask RCNN+ResNet50) achieves an F-score of 80.9%, which is over 5.0% lower than the YOLOv5 model. In addition to the above evaluation metrics, we want to highlight that YOLOv5 is much faster (∼2.5 times) than RCNN models when detecting objects on UI. This is another critical factor for our choice as mobile devices usually require very quick UI responses for user interactions.

End-to-end comparison between *DARPA* and FraudDroid-like approach. While *DARPA* focuses on detecting dark design patterns (i.e., AUI), it is essentially an instance of *mobile UI analysis techniques* [28], [31], [39], [40], [47]. Most of the techniques, apart from the CV-based techniques

TABLE V: Comparison between YOLOv5 and other models.

Model	Precision	Recall	F1-score
Faster RCNN+VGG16	0.732	0.710	0.721
Faster RCNN+ResNet50	0.744	0.698	0.720
Mask RCNN+VGG16	0.802	0.762	0.781
Mask RCNN+ResNet50	0.829	0.789	0.809
YOLOv5	0.881	0.838	0.859

(which we compared earlier), rely on the analysis of the UI-related strings and placement features. A prominent example is FraudDroid [31], which uses the resource id of a UI, and its size and placement features to identify advertising UIs. We believe the same features can be used to detect AUI as well. Thus, we compare *DARPA* with a FraudDroid-like approach. A key problem is that the FraudDroid module that detects advertising UIs using the above features, *AdViewDetector*, is not readily available for comparison since it is close-sourced. We aim at a relatively fair comparison by implementing FraudDroid based on the paper and then extending it to AUI detection. Specifically, we enrich the UI string features by adding resource ids corresponding to the AUIs (as summarized in Section III-A). Then, we reuse the heuristic-based approach and the placement features of FraudDroid to determine if a UI contains an AUI.

TABLE VI: Confusion matrix of *DARPA* and FraudDroid

		FraudDroid		DARPA	
		AUI	Non-AUI	AUI	Non-AUI
Labeled	AUI	35	208	213	30
	Non-AUI	11	242	21	232

For evaluation, we randomly select 100 apps from D_{app} and run each of them manually for one minute by Monkey [17], and use the FraudDroid-like approach and *DARPA* to evaluate the app screenshots captured by AS and the corresponding metadata of screenshots captured by ADB tool [4]. Then, we review the UPOs flagged by the approaches and decide if they indeed correspond to the UI options of AUIs. We show the confusion matrices in Table VI. As can be seen, *DARPA* identifies over 87.6% (213 out of 243) screenshots that have UPOs with a pretty high precision (91.0%), while the FraudDroid-like approach only identifies 14.4% UPOs. This is mainly due to the infeasibility to find the resource ids (which the FraudDroid-like approach relies on) as many apps choose to obfuscate them to defeat reverse engineering or the resource ids are generated dynamically, and the difficulty to curate a comprehensive list of resource ids for the wide variety of AUIs (see Section III-A). The distinctive coverage of AUI substantiated our point that, by leveraging a CV-model that captures the visual features of a UI, *DARPA* is more generic in detecting various types of AUIs.

D. RQ3: Performance Overhead

We report the performance overhead of *DARPA* by comparing several performance metrics of the smartphone when running apps with and without *DARPA*. Specifically, we randomly select 100 apps from D_{app} and manually run each app

TABLE VII: Performance overhead of *DARPA*.

	Avg. CPU Usage (%)	Avg. Memory Usage (MB)	Avg. Frame Rate (fps)	Avg. Power Consumption (milliWatt)
Baseline (w/o *DARPA*)	55.22	4291.96	81	443.85
Baseline + UI monitoring	55.91 (↑ 1.25%)	4352.21 (↑ 1.40%)	79 (↓ 2.47%)	451.88 (↑ 1.81%)
Baseline + UI monitoring + AUI detection	57.11 (↑ 2.15%)	4407.56 (↑ 1.27%)	78 (↓ 1.27%)	469.63 (↑ 3.93%)
DARPA (UI monitoring + AUI detection + UI decoration)	57.76 (↑ 1.14%)	4413.85 (↑ 0.14%)	74 (↓ 5.13%)	474.12 (↑ 0.96%)
Total overhead (compared to baseline)	2.54 (↑ 4.6%)	121.84 (↑ 2.8%)	7.00 (↓ 8.6%)	30.27 (↑ 6.8%)

TABLE VIII: Performance of *DARPA* under different selected time interval.

Selected interval (ms)	Avg. CPU Usage (%)	Avg. Memory Usage (MB)	Avg. Frame Rate (fps)	Avg. Power Consumption (milliWatt)
50	86.5	4452.53	59	586.92
100	69.8	4419.69	66	499.55
200	57.8	4413.85	74	474.12
300	54.8	4401.12	69	481.5
400	59.7	4360.52	76	469.96
500	56.1	4354.63	79	464.85

for one minute, and record the events (e.g., clicks) used to trigger the app. At the same time, we execute SoloPi [14] to gather the performance metrics, such as memory/CPU usage, frame rate, and power consumption. After that, we re-run the apps by replaying the recorded events using Airtest [5], an open-source app UI automation framework, and start *DARPA* to check the apps at run-time. We execute SoloPi a second time to gather the performance metrics with *DARPA* running.

Table VII shows the average performance overhead on the 100 D_{app} apps. As it can be seen from the last row of the Table, *DARPA* causes an increase of 4.6%, 2.8%, and 6.8% in terms of the usage of CPU, memory, and power consumption, respectively. *DARPA* also results in an 8.6% decrease of the average frame rate. To understand where the performance overhead comes from, we measured the overhead by incrementally adding the individual components, i.e., *UI monitoring*, *AUI detection*, and *UI decoration*, and inspecting the corresponding overhead changes. As can be seen in Table VII, the main reason for the overhead is running the AUI detection model (i.e., YOLO), including processing the captured screenshots and detecting AGO and UPO from them. For example, the AUI detection component brings a 3.93% increase of power consumption, while UI monitoring and decoration components bring in 1.81% and 0.96% increase, respectively. The overhead from AUI detection can practically be reduced by using a smaller network size in YOLO with potential trade-off of lower accuracy, running the YOLO model on top of GPU (we used CPU implementation in our prototype due to hardware limitations), etc.; the other two components rely primarily on system services to function (i.e., accessibility service and window manager), and thus reducing their overhead may require OS optimizations. We believe the overheads are relatively low or even unnoticeable for end users. For example, *DARPA* introduces 121.84MB more memory usage (mainly for hosting the CV-model and related data), which accounts for less than 2% memory given that an average smartphone has at least 6GB RAM [9]. Additionally, instead of monitoring all running apps as the default setting, a possible way to further reduce such overhead could be selectively running *DARPA* on those less-trusted apps (e.g., apps not from google play).

E. RQ4: Proper Timing for Executing DARPA

As discussed in Section IV-B, analyzing all UI update events (e.g., making screenshots, and running AUI detection) is not feasible due to the high frequency of the events. Therefore, we apply a ct value to determine the subset of UI events that we need to analyze. We perform a series of experiments to find the optimal ct that increases the efficiency of *DARPA*, and, at the same time, does not lead to a significant impact on the coverage of AUIs. Specifically, we set ct to six values 50ms, 100ms, 200ms, 300ms, 400ms and 500ms. For each value, we run *DARPA* on the D_{app} apps, and evaluate the coverage of AUIs and the performance metrics using the same methods as in Section VI-C and Section VI-D. As shown in the trendlines in Figure 8, the number of UI change events drops with the increase of the ct value, as the number of identified AUIs. Compared to the smallest ct value (50ms), setting ct to 200ms still allows *DARPA* to detect 94.1% (191 out of 203) of the AUIs. In the meantime, the overall workload, i.e., the total number of events and UI changes to evaluate, dropped by over 67.1% (1,538 out of 2,291). Therefore, we choose 200ms as an optimal ct value as *DARPA* can achieve efficiency and good coverage on AUI at the same time. This is also confirmed by the performance metrics in Table VIII. Coincidentally, this ct value also aligns with the human reaction time to changes [32], [38], i.e., app users may require a similar amount of time to react to UI changes as well.

VII. Discussion

Our study focuses on the AUIs that can mislead users into performing unwanted behaviors, e.g., clicking advertising UIs or subscribing to unintended services. We aim to mitigate such risks by identifying and highlighting such AUIs to users at app run-time. In addition to helping end users, our approach can contribute to other use cases, such as assisting app stores or regulatory agencies in detecting apps with misleading UIs. These use cases are fundamental to ensure the usability of

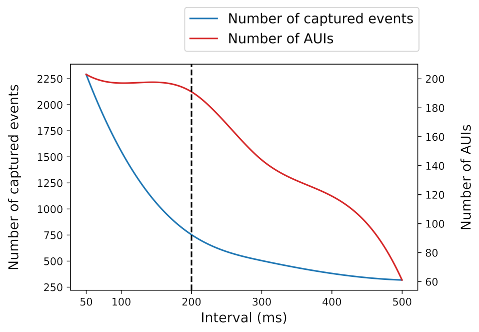


Fig. 8: AUI coverage under different interval thresholds

mobile devices. Besides, we want to note that, although we focus on AUIs that mislead users and many AUIs detected in this study might not be immediately malicious (see examples in Figure 2), *DARPA* can potentially help to identify some malicious behaviors that take advantage of AUIs, e.g., apps that trick users into authorizing dangerous app permissions.

System reliability. As a solution designed to identify AUIs, *DARPA* faces challenges from attackers who use various techniques to evade its detection. Some attackers may make straightforward changes to the appearance of the UIs, such as altering the color, size, or location of UI elements. As we demonstrated in Section VI-B, *DARPA* is capable of detecting a wide range of AUIs in D_{aui} effectively, we believe that it can still identify such changes as long as the UI elements remain visible and usable to users. In the worst case, *DARPA* may fail to recognize AGO or UPO options on an AUI, and thus is not able to highlight the options by UI decoration. In such a case, app users need to make their own decisions without guidance from *DARPA*, exposing them to potential misleading UIs.

Limitations. Admittedly, the current design of *DARPA* shares the following limitations. (1) Determined attackers can freely test the adopted CV-model to develop targeted attacks, such as adversarial patch attacks commonly used in object detection models [27], [37], [41], [51]. Currently, *DARPA* cannot defend against such targeted attacks, but it can be supplemented with more resilient models [25]. (2) Further, our current *DARPA* implementation is limited to analyzing and identifying AUIs on the Android platform. We believe similar ideas of CV-based detection are applicable to other platforms as well, e.g., by using screen capture [23] and window controls overlay [24] APIs on web (or browsers), and by directly integrating the detection into operating systems such as iOS. We leave them as future work as porting the ideas to these platforms requires additional implementation and integration efforts.

VIII. Related Work

Dark UI patterns. Prior studies [26], [43], [45], [52] reported the prevalence of dark UI patterns in a variety of settings. Mathur et al. [43] studied 11K shopping websites, and revealed how the websites leverage dark patterns to deceive users into making more purchases, etc. Zagal et al. [52] presented the concept of dark game design patterns, and discussed the questionable behaviors of online game designers. Mejtoft et al. [45] analyzed a number of home cooking recipe websites for the purpose of understanding the deceptive design patterns with regard to the use of cookies. Bosch et al. [26] introduced the dark strategies/patterns used by malicious parties to destroy the privacy awareness of users of IT systems. Additionally, Mathur et al. [44] noticed the need of a consistent conceptual definition of dark patterns, and designed a set of normative perspectives for such patterns. Unlike the studies that investigate dark patterns in specific settings such as shopping and privacy, our work builds a more *generic* approach on top of computer vision to identify the *asymmetric dark patterns* appearing in the *mobile apps*, based on the observation that such patterns introduce uneven visual perceptions to app users so as to deceive them into performing unwanted activities.

UI-based analysis. Our study detects dark patterns by analyzing app UIs, which is similar to other research on UI-based analysis. A major line of such research is about ad fraud [31], [39], [40]. For example, FraudDroid [31] identifies ad views with UI placement and string features and detects ad fraud by inspecting UI transition graphs and network traffic. DECAF [39] detects placement fraud, e.g., small or hidden ads, by checking the UI layouts and states against several detection rules. Maddroid [40] detects devious ad contents using a set of tools, e.g., using Google Vision API and OCR to identify censored images (e.g., gambling and violence images). While our approach can potentially cover specific types of ad fraud, e.g., placement fraud with small ads, it is not designed for detecting fraudulent activities on UI. Instead, we focus on the analysis of dark UI patterns, which can visually mislead users into performing unwanted activities (e.g., sign-up to recurrent payments) but may not lead to UI-based fraud. More related to our study is the line of research on UI element detection, e.g., Chen et al. [28] identifies UI elements using the boundary, shape, texture and layout and other UI features, and REMAUI [47] detects UI elements (e.g., images, lists) using CV and OCR techniques. Similar to REMAUI, we adopt a CV-based approach for identifying specific UI elements involved in dark patterns. Additionally, we apply our approach to mobile devices by interpolating it into the apps' runtime. This allows us to perform real-time detection of dark UI patterns, and end-to-end mitigation for such patterns.

IX. Conclusion

In this paper, we perform the first in-depth analysis for the Asymmetric Dark UI pattern in the mobile ecosystem. To achieve this, we performed a user study to understand use perceptions and expectations of AUI. Our research showed that AUI is pervasive in mobile apps and brings negative impacts on user experience. To combat AUI, we propose *DARPA*, a generic, lightweight framework to highlight key elements in AUI for app usability improvement for Android devices. An extensive evaluation showed that *DARPA* is indeed practical, with neglectable performance overhead to the original system.

Acknowledgment

We are grateful to our reviewers and shepherd for their support and suggestions. This work was partially supported by the National Natural Science Foundation of China (No.62202510), Special Projects in Key Fields of Universities in Guangdong Province (No. 2022ZDZX1001), and the Fundamental Research Funds for the Central Universities, Sun Yat-sen University (No.22lgqb26). We especially thank the Alibaba Innovative Research Program (AIR) for the support of this research.

References

[1] How tracking social media shares can strengthen your campaigns. https://sproutsocial.com/insights/tracking-social-media-shares/, 2021.

[2] Accessibility event. https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent, 2022.

[3] Android accessibility service. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService, 2022.

[4] Android debug bridge(adb). https://developer.android.com/studio/command-line/adb, 2022.

[5] Aolopi, cross-platform ui automation framework for games and apps. https://github.com/AirtestProject/Airtest, 2022.

[6] Common objects in context. https://cocodataset.org/#home, 2022.

[7] Ftc calls for a reboot on business guidance about digital advertising. https://www.ftc.gov/business-guidance/blog/2022/06/ftc-calls-reboot-business-guidance-about-digital-advertising, 2022.

[8] Global mobile advertising guidelines. https://www.mmaglobal.com/policies/global-mobile-advertising-guidelines, 2022.

[9] How much ram does a smartphone need? https://www.makeuseof.com/how-much-ram-smartphone-need/, 2022.

[10] Huaban, a usable chinese design website. https://huaban.com/, 2022.

[11] mnn. https://github.com/alibaba/MNN, 2022.

[12] Mobile marketing - legal do's don'ts. https://www.loeb.com/en/insights/publications/2010/04/mobile-marketing--legal-dos--donts, 2022.

[13] ncnn. https://github.com/Tencent/ncnn, 2022.

[14] Solopi, a tools to record and show the app's performance data such as cpu, memory, internet speed while do the testing. https://github.com/alipay/SoloPi/blob/master/README_eng.md, 2022.

[15] "survey star", an online questionnaire crowd-source platform in china. https://www.wjx.cn/, 2022.

[16] Taobao, a popular shopping app in china. https://play.google.com/store/apps/details?id=com.taobao.taobao, 2022.

[17] Ui/application exerciser monkey. https://developer.android.com/studio/test/other-testing-tools/monkey, 2022.

[18] Xiaomi app store. https://www.mi.com/in/appdownload/, 2022.

[19] Yolo - an object detection algorithm. https://github.com/ultralytics/yolov5, 2022.

[20] Android ndk. https://developer.android.com/ndk, 2023.

[21] Building accessible apps. https://developer.apple.com/accessibility/, 2023.

[22] An introduction to mobile app accessibility. https://bitrise.io/blog/post/an-introduction-to-mobile-app-accessibility, 2023.

[23] Screen capture api. https://developer.mozilla.org/en-US/docs/Web/API/Screen_Capture_API, 2023.

[24] Window controls overlay api. https://developer.mozilla.org/en-US/docs/Web/API/Window_Controls_Overlay_API, 2023.

[25] Naveed Akhtar, Ajmal Mian, Navid Kardan, and Mubarak Shah. Advances in adversarial attacks and defenses in computer vision: A survey. *IEEE Access*, 9:155161–155196, 2021.

[26] Christoph Bösch, Benjamin Erb, Frank Kargl, Henning Kopp, and Stefan Pfattheicher. Tales from the dark side: Privacy dark strategies and privacy dark patterns. *Proc. Priv. Enhancing Technol.*, 2016(4):237–254, 2016.

[27] Tom B Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. *arXiv preprint arXiv:1712.09665*, 2017.

[28] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. Object detection for graphical user interface: Old fashioned or deep learning or a combination? In *proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1202–1214, 2020.

[29] Vanessa Courville and Vahid Partovi Nia. Deep learning inference frameworks for arm cpu. *Journal of Computational Vision and Imaging Systems*, 5(1):3–3, 2019.

[30] Ming Di, Shah Nazir, and Fucheng Deng. Influencing user's behavior concerning android privacy policy: An overview. *Mobile Information Systems*, 2021, 2021.

[31] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. Frauddroid: Automated ad fraud detection for android apps. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 257–268, 2018.

[32] Burkhart Fischer and E Ramsperger. Human express saccades: extremely short reaction times of goal directed eye movements. *Experimental brain research*, 57:191–195, 1984.

[33] Colin M Gray, Yubo Kou, Bryan Battles, Joseph Hoggatt, and Austin L Toombs. The dark (patterns) side of ux design. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pages 1–14, 2018.

[34] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046, 2014.

[35] Jie Huang, Michael Backes, and Sven Bugiel. A11y and privacy don't have to be mutually exclusive: Constraining accessibility service misuse on android. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3631–3648, 2021.

[36] Arash Habibi Lashkari, Andi Fitriah A Kadir, Laya Taheri, and Ali A Ghorbani. Toward developing a systematic approach to generate benchmark android malware datasets and classification. In *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–7. IEEE, 2018.

[37] Mark Lee and Zico Kolter. On physical adversarial patches for object detection. *arXiv preprint arXiv:1906.11897*, 2019.

[38] Letizia Leocani, Leonardo G Cohen, Eric M Wassermann, Katsunori Ikoma, and Mark Hallett. Human corticospinal excitability evaluated with transcranial magnetic stimulation during different reaction time paradigms. *Brain*, 123(6):1161–1173, 2022.

[39] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. {DECAF}: Detecting and characterizing ad fraud in mobile apps. In *11th USENIX symposium on networked systems design and implementation (NSDI 14)*, pages 57–70, 2014.

[40] Tianming Liu, Haoyu Wang, Li Li, Xiapu Luo, Feng Dong, Yao Guo, Liu Wang, Tegawendé Bissyandé, and Jacques Klein. Maddroid: Characterizing and detecting devious ad contents for android apps. In *Proceedings of The Web Conference 2020*, pages 1715–1726, 2020.

[41] Xin Liu, Huanrui Yang, Ziwei Liu, Linghao Song, Hai Li, and Yiran Chen. Dpatch: An adversarial patch attack on object detectors. *arXiv preprint arXiv:1806.02299*, 2018.

[42] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing cnn model inference on cpus. 2019.

[43] Arunesh Mathur, Gunes Acar, Michael J Friedman, Eli Lucherini, Jonathan Mayer, Marshini Chetty, and Arvind Narayanan. Dark patterns at scale: Findings from a crawl of 11k shopping websites. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–32, 2019.

[44] Arunesh Mathur, Mihir Kshirsagar, and Jonathan Mayer. What makes a dark pattern... dark? design attributes, normative considerations, and measurement methods. In *Proceedings of the 2021 CHI conference on human factors in computing systems*, pages 1–18, 2021.

[45] Thomas Mejtoft, Erik Frängsmyr, Ulrik Söderström, and Ole Norberg. Deceptive design: cookie consent and manipulative patterns. In *34th Bled eConference-Digital support from crisis to progressive change, Online, June 27-30, 2021*, pages 397–408, 2021.

[46] Mohammad Naseri, Nataniel P Borges Jr, Andreas Zeller, and Romain Rouvoy. Accessileaks: Investigating privacy leaks exposed by the android accessibility service. 2019.

[47] Tuan Anh Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 248–259. IEEE, 2015.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep

learning library. *Advances in neural information processing systems*, 32, 2019.

[49] Charles E Robertson, J Kirk Harris, Brandie D Wagner, David Granger, Kathy Browne, Beth Tatem, Leah M Feazel, Kristin Park, Norman R Pace, and Daniel N Frank. Explicet: graphical user interface software for metadata-driven management, analysis and visualization of microbiome data. *Bioinformatics*, 29(23):3100–3101, 2013.

[50] Lukas Stefanko. Insidious android malware gives up all malicious features but one to gain stealth. *Accessed: Jan*, 27:2021, 2020.

[51] Simen Thys, Wiebe Van Ranst, and Toon Goedemé. Fooling automated surveillance cameras: adversarial patches to attack person detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pages 0–0, 2019.

[52] José P Zagal, Staffan Björk, and Chris Lewis. Dark patterns in the design of games. In *Foundations of Digital Games 2013*, 2013.

[53] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104, 2012.

[54] Tong Zhu, Yan Meng, Haotian Hu, Xiaokuan Zhang, Minhui Xue, and Haojin Zhu. Dissecting click fraud autonomy in the wild. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 271–286, 2021.

[55] Onur Zungur, Gianluca Stringhini, and Manuel Egele. Libspector: Context-aware large-scale network traffic analysis of android applications. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 318–330. IEEE, 2020.