A Comparative Study between Android Phone and TV Apps

Yonghui Liu[1], Xiao Chen[2], Yue Liu[1], Pingfan Kong[3], Tegawendé F. Bissyandé[3], Jacques Klein[3], Xiaoyu Sun[4], Li Li[*,5], Chunyang Chen[6], John Grundy[1]

[1]Department of Software Systems and Cybersecurity, Monash University, Australia.
[2]School of Information and Physical Sciences, University of Newcastle, Australia.
[3]SnT, University of Luxembourg, Luxembourg.
[4]The School of Computing, Australian National University, Australia.
[5]School of Software, Beihang University, China.
[6]Department of Computer Science, Technical University of Munich, Germany.

Contributing authors: lilicoding@ieee.org;

Abstract

Smart TVs have surged in popularity, leading developers to create TV versions of mobile apps. Understanding the relationship between TV and mobile apps is key to building consistent, secure, and optimized cross-platform experiences while addressing TV-specific SDK challenges. Despite extensive research on mobile apps, TV apps have been given little attention, leaving the relationship between phone and TV apps unexplored.

Our study addresses this gap by compiling an extensive collection of 3,445 Android phone/TV app pairs from the Google Play Store, launching the first comparative analysis of its kind. We examined these pairs across multiple dimensions, including non-code elements, code structure, security, and privacy aspects. Our findings reveal that while these app pairs could get identified with the same package names, they deploy different artifacts with varying functionality across platforms. TV apps generally exhibit less complexity in terms of hardware-dependent features and code volume but maintain significant shared resource files and components with their phone versions. Interestingly, some categories of TV apps show similar or even severe security and privacy concerns compared to their

mobile counterparts. This research aims to assist developers and researchers in understanding phone-TV app relationships, highlight domain-specific concerns necessitating TV-specific tools, and provide insights for migrating apps from mobile to TV platforms.

Keywords: Android, Apps, Phone, TV

1 Introduction

Smart TVs have become the dominant TV type. As reported by Laricchia (2022a), more and more users are switching from traditional TVs to Smart TVs, which offer additional features and conveniences such as allowing users to watch Netflix and YouTube directly on TVs through internet connectivity. According to Group (2021), the global smart TV market is expected to reach a value of USD 253 billion by 2023. Among various types of Smart TVs on the market, Android TV, with a lot of benefits extended from the popular Android ecosystem, has undoubtedly become one of the most popular ones (Tileria and Blasco 2022). Similar to Android phones, Android TVs enable users to connect to the Google Play store to download and update apps, as well as utilize Google Assistant to accomplish hands-free tasks.

Despite the growing popularity of the smart TV industry and the increasing number of TV devices in the Android ecosystem, the number of available TV apps (around 7,000) is significantly lower than the number of smartphone apps (over 2.6 million on Google Play) Laricchia (2022b). As Android TVs expand the features of the Android ecosystem, they also inherit potential security vulnerabilities present in other Android devices. Moreover, customizations made for Android TVs may introduce additional security risks for users. Aafer et al. (2021) discovered 37 unique high-impact vulnerabilities in 11 Android TV boxes using log-guided fuzzing, while Liu et al. (2021) and Tileria and Blasco (2022) analyzed the security and privacy of TV apps.

Previous studies have primarily focused on mobile apps, leaving the differences between mobile and TV apps poorly understood. The gap between smartphones and smart TV apps is often overlooked, resulting in a limited understanding of TV apps compared to smartphones. Our lightweight literature search, conducted on Google Scholar using various combinations of keywords such as "Android TV app," "Smart TV app," "similarity," and "app pairs," reveals that the connection between smartphone and TV apps has not been thoroughly explored. Only one related work, by Hu et al. (2023), exists to help understand the relationship between Android TV and Android mobile apps. However, this study was limited by a small dataset of only 298 app pairs.

Exploring the relationship between TV and mobile apps is essential for creating consistent, optimized, and secure user experiences across platforms. As users could interact with the same services on both devices, understanding this relationship enables developers to design seamless transitions and tailored interfaces that account for distinct interaction patterns, such as touch versus remote control. Additionally, TV apps often inherit vulnerabilities from their mobile counterparts, and platform-specific

customizations may introduce new risks, making security and privacy a critical focus. By analyzing the similarities and differences between these app versions, developers can streamline development processes, reuse code, and reduce costs while addressing the growing demand for high-quality TV applications. Furthermore, this exploration fills a significant research gap, as TV apps have received far less attention than mobile apps, and provides insights that can drive innovation, such as cross-platform integrations and synchronized content. Research in this area would provide valuable knowledge to bridge the gap between smartphone and TV app development, ultimately benefiting both developers and users in the Android ecosystem.

Currently, no publicly available dataset primarily focuses on Android phone/TV app pairs, as existing datasets like AndroZoo (Allix et al. 2016) consist mainly of Android mobile apps. Collecting a comprehensive dataset for Android TV and phone app pairs is challenging, and to the best of our knowledge, the largest dataset only contains 298 phone/TV app pairs (Hu et al. 2023). To gain a comprehensive understanding of phone/TV app pairs, we aim to deliver a market-scale collection of these pairs. We further analyze them quantitatively and qualitatively from various perspectives, including non-code aspects (metadata, resources, permissions), code elements (components, methods, user interactions), and security and privacy analysis. Based on our findings, we provide insightful discussions and recommendations for future research directions in this domain.

The key contributions of our work are the following:

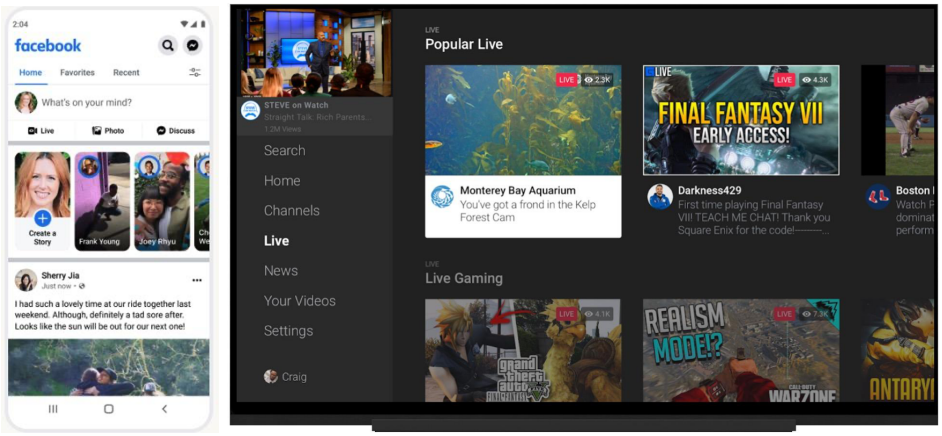
- We gathered a substantial dataset consisting of 3,445 Android phone/TV app pairs from the Google Play Store, representing a comprehensive sample of the market. To encourage further research in this field, we have made the dataset publicly available online through the Zenodo [1].
- We experimentally examined Android phone and TV app pairs from multiple angles, including code/non-code similarities, security, and privacy concerns.
- We provided insight based on our empirical results and presented a set of practical recommendations and future research directions.

The remainder of the paper is organized as follows. Section 2 discusses the study's rationale and motivation. Section 3 details the data collection procedure, and data characteristics. Section 4 offers the results and conclusions of our empirical investigation, followed by a discussion of the study's implications and potential risks to validity in Section . Section 6 presents the future research direction. Section 7 discusses relevant literature, and section 8 concludes the article.

2 Motivation

Smartphones and smart TVs serve distinct purposes in daily life, leading to significant differences in app usage and design. While smartphones are portable devices used for various tasks like communication and financial transactions, smart TVs are better suited for immersive entertainment on large screens. **To understand these usage scenario differences, we analyzed the top 100 apps on both platforms from**

[1] https://zenodo.org/records/7675881



(a) Smartphone (b) Smart TV

Fig. 1 app, Facebook, with different SHA256 for base.apk installed on Smartphone and Smart TV.

the Google Play Store. Our analysis revealed stark contrasts in app categories: entertainment dominates TV apps, accounting for 46% of top apps, compared to just 12.5% on smartphones. Moreover, the Google Play Store offers 39 unique categories for smartphones but only 12 for TV devices, reflecting more focused use cases on TVs. These findings highlight the distinct roles these devices play in users' daily lives and the need for platform-specific app development strategies.

Beyond these quantitative differences, TV apps exhibit qualitative differences due to hardware limitations and usage scenarios. TVs tend to serve a distinct role from other devices, and they lack the hardware features that other Android-powered devices generally offer. Some hardware features on other Android devices, such as touch displays, cameras, and GPS receivers, are not common on TVs. As introduced in Google Developer Doc (Google 2022e), TVs are fully reliant on third-party hardware components, and users must utilize a remote control or a gaming pad to engage with TV apps. Due to the hardware capabilities and screen size differences between smartphones and TVs, the appearance and functionality of the same app (i.e., same package name) can vary across phone/TV devices.

A case study of the Facebook app illustrates these differences. As seen in Figure 1, the screenshots of the home page of the app, *"Facebook"*, in phone and TV devices, respectively, are completely different. However, both the phone and TV versions share the same package name, *com.facebook.katana*[2], but distribute different *App Package File (APK)* for different platforms. The phone's version of *Facebook* is known as an app for social with the size of 61.5 Mb. However, one live video streaming app with the size of 1.6 Mb is provided for the TV version of *Facebook* that is not collected by the well-known Android mobile app dataset, AndroZoo (Allix et al. 2016; Allix 2021). This particular APK can only be distributed to the TV devices as the introduction of *Android App Bundles (AAB)* and *Multiple APK support* on Google Play. An *Android App Bundle* is a publishing format with which Google Play

[2]https://play.google.com/store/apps/details?id=com.facebook.katana

uses app bundle to generate and serve optimized APKs for each device configuration Google (2022a). *Multiple APK support* is a Google Play feature that enables you to publish several APKs for your app, each of which is optimised for a certain device configuration Google (2022b). For instance, *Facebook*[3] distribute the different APKs on different platforms (i.e., TV and Phone). In addition, apps may distribute the same APK file to several platforms, indicating the multiple-platform artifacts are packaged into a single APK.

The introduction of *Android App Bundles* and *Multiple APK support* affects the dataset in previous work. The work from both Tileria and Blasco (2022) and Liu et al. (2021) neglect the effect of *Android App Bundles* and *Multiple APK support* on TV-specific app analysis. Consequently, the TV APKs that are distributed independently from the mobile version are not thoroughly analyzed in the present study. Additionally, APKs containning components for multiple platforms, would lead to misinterpretations of TV-specific app characteristics. Our research aims to clarify the differences between TV and mobile apps, especially for those with platform-specific distributions.

3 Dataset Collection and Dataset Evaluation

In this section, we describe the methodology we used to gather real-world phone/TV app pairs (cf. Section 3.1). Subsequently, we evaluate the proportion of app pairs with the identical or a different App Package Name (cf. Section 3.2). Next, we categorize app pairs as having the identical or a distinct App Package (cf. Section 3.3).

3.1 Dataset collection

To the best of our knowledge, there is no publicly accessible dataset of phone/TV app pairs; thus, we create one from scratch. Figure 2 depicts the whole strategy. Detailed explanations of each step are provided below.

App Package Name Collection. Too few apps are shown on the listing page of the official Google Play store for TV devices (2024). The information available on the official Google Play store is not sufficient to compile a comprehensive dataset of phone/TV app pairs. Our approach involves initially gathering the package names of all apps available from the Google Play Store and then verifying whether they offer versions for both Android phones and TVs. However, obtaining a complete list of app package names from the Google Play Store is a complex task. To address this issue, we have opted to source the list of package names from AndroZoo (Allix et al. 2016; Allix 2021) instead of attempting to scan the entire Google Play Store. AndroZoo is an Android app repository that contains more than 18 million Android smartphone apps (and is still growing) collected from various app markets. We keep the apps sourced from official Google Play store and further remove the duplicated versions (i.e., historical versions of the same app), we obtain 6,400,075 unique package names.

Phone/TV App Pairs Collection. With the gathered package names, we next determine if they are searchable on Android phones and TVs. According to the official Android documentation (2021), app developers may opt to show their apps from

[3] com.facebook.katana

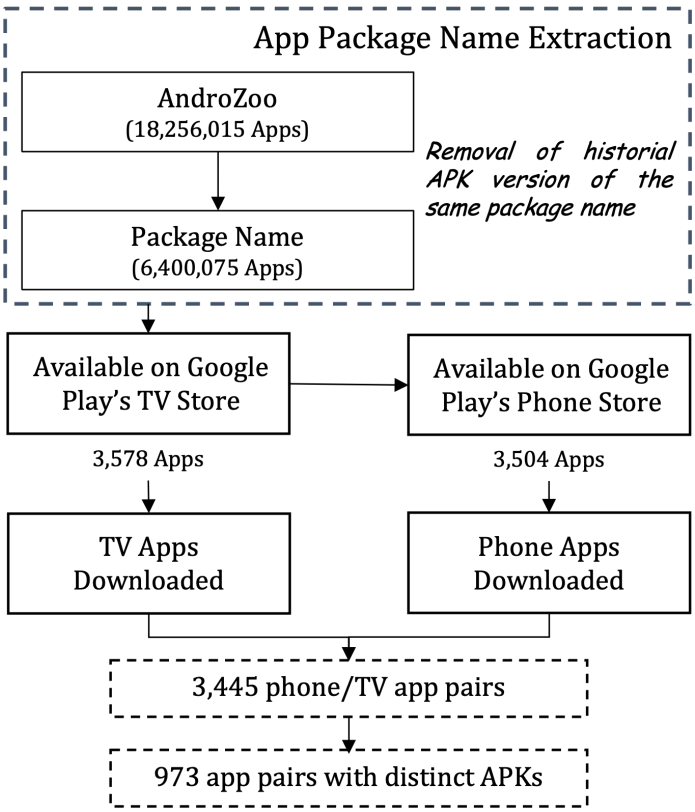


Fig. 2 Working Process to Collect Android phone/TV app pairs from AndroZoo dataset.

the Google Play store on just certain devices (e.g., TV, watch, automobile, etc.) and will not be searchable or loaded on other devices. Our idea is to simulate the desired devices (i.e., Android phone and TV, both with Android API level 30 and x86-based architecture) when querying the Google Play APIGoogle (2021a) with the package name. We also exclude the paid apps and only keep the free ones. Later, using the searchable package names of free apps, we query Google Play API with the identical device settings to download these apps for both platforms. In consideration of app version compatibility, we simultaneously download phone/TV apps for each pair. Finally, we gathered a total of 3,445 phone/TV app pairs.

3.2 Pairs with Identical/Different App Package Name

It is worth mentioning that our above approach can only collect app pairs that have identical package names on both platforms. While it is possible that a developer have separate apps for mobile devices and TV devices, Google recommends having a single app that supports both mobile devices and TV devices Google (2023a). It is non-trivial to determine if two apps with different package names just contain the same app content optimized for different platforms. Therefore, our dataset exclude app pairs

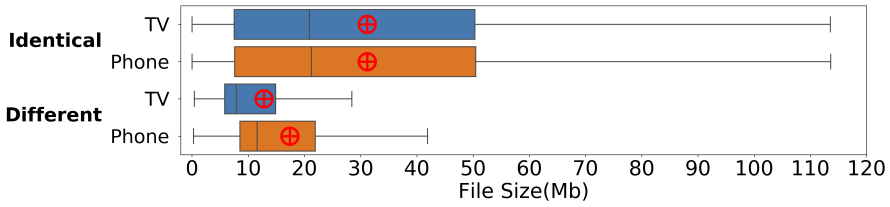


Fig. 3 Distribution of App size

that have different package names on phone and TVs. Nevertheless, to give readers an implication of how many app pairs our approach may miss, we further conducted an additional app pairs collection to show the proportion of phone/TV app pairs that have the different package name. To this end, we retrieve the package names of all TV apps on an alternative Android app store APKMirror (2021), which keeps the information of an app's compatible platforms (e.g., TV, watch, car, etc.). We have collected in total 819 unique TV apps from APKMirror, within which 667 apps are available on Google Play. Within the 667 TV apps, approx. 75% (i.e., 502) of them have a smartphone version with an identical package name. We further confirmed manually (by searching the other apps from the same developers, comparing the descriptions and running the apps on the TV and phone emulators) that approx. 11% (i.e., 72) of them have a smartphone version with different package names, while approx. 14% (i.e., 93) of them do not have a smartphone version. The results suggest that only a small percentage of apps (i.e., 11%) use different package names on different platforms that we may miss in our dataset.

3.3 Pairs with Identical/Different App Package Artifacts

To enable developers publishing one app on different platforms, Goolge allow developers to develop multiple APKs for the same app or package multiple-platforms artifact into one APK. The feature, *multiple APK support*, on Google play enables different APKs to be published for the same app depending on the device's characteristics. The rationale for *multiple APK support* on Google Play is that even though your app is represented by a single listing, various devices may be distributed with various APKs. All APKs published for the same app must have the same app package name and be signed with the same certificate key Google (2022b). By comparing the SHA256 of the apps in each pair, we find that there are total $2,472$ pairs that have identical APKs on phones and TVs. In contrast, 973 pairs were distributed with the different APKs for the same app, which is a clear signature that these apps are published with the *Multiple APK suport*. For apps in particular categories (e.g., Music & Audio, Sports, News & Magazines, etc.), more developers tend to publish multiple APKs for the same App.

Additionally, we further investigated the size of the APKs in phone/TV app pairs. As shown in Figure 3, for the apps with the identical APK for both platforms, the average APK size is 31.1 Mb. On the other hand, the size of the apps with different APKs on the phone and TV is significantly smaller (i.e., 12.6 Mb and 17.1 Mb for TV and phone apps, respectively). It is noteworthy that the size of the APK, which remains the same across both platforms, is slightly larger (31.1 Mb) than the combined

size (29.7 Mb) of the phone and TV versions with separate APKs. This implies that apps that utilize the identical APK for both TVs and phones tend to include code and resources for both platforms within the one APK.

4 Empirical Results

In our empirical study, we analyze phone/TV app pairs from two perspectives: artifact composition and security/privacy. We specifically exclude app pairs that use identical APKs, as these merge artifacts from multiple platforms (phone and TV) into a single APK, making it challenging to isolate the artifact for each platform. Additionally, existing research does not provide platform-specific information in their analysis results. Therefore, we focus our comparative analysis on 973 pairs of apps that have distinct APKs for different device platforms. We begin by examining the composition of the apps, including their code, non-code elements, and permissions, and then delve into analyzing their underlying behavior, focusing on privacy and security aspects. The composition and behavior of these apps are integral to the software development lifecycle, requiring developer attention during the design, coding, and maintenance phases. This analysis is intended to inform practitioners about the current state of phone/TV app pairs. The sub-sections of this section will outline our findings in response to two following research questions.

- **RQ1: [*Code and Non-Code*] To what extent are phone apps similar to their TV counterparts?** In this RQ, we analyze the non-code, code elements, and permission declared of phone/TV pairs and want to find out to what extent phone apps are similar to their TV counterparts.
- **RQ2: [*Security and Privacy*] To what extent are the security/privacy issues different between phone/TV app pairs?** In this RQ, we compared the security and privacy analysis results, aiming to determine to what extent the security and privacy risks differ between phone and TV apps

4.1 RQ1: APK Artefact

Similarity Analysis. In this RQ, we examine the similarity of the non-code (e.g., resources) and the code (e.g., Activities, Permissions) artifacts of phone/TV app pairs. Resources are non-code assets that the app code could access, such as images, textual data, and User Interface (UI) layouts. In terms of the app code, both components and methods are inspected. An Android app has four primary components, including Activities, Services, Broadcast Receivers, and Content Providers. Each component has a well-defined lifecycle and they are the essential building blocks of an Android app. Apart from the components and the methods, we also compare the permissions declared. Comparing the permissions can help us understand the feature between app pairs.

We leverage SimiDroid (Li et al. 2017) to compare the pairwise similarities and differences across apps in each phone/TV pair in terms of two comparison levels (i.e., resource, and component). Key/value mapping pairs (map1 and map2) are constructed for each app based on the comparison level. For resource level comparison,

hash values of the files' content are calculated for the key/value pairs. SimiDroid's component-based comparison extracts the component name as the key and other package information corresponding to component capabilities, such as action, which describes the type of behaviour matched by the component (e.g., Main component), and category, which specifies what the component represents (e.g., LAUNCHER) as value. The following metrics are then used to compare them: (1) identical, when compared key/value pairs are exactly matched; (2) similar, where the key is the same but values differ; (3) new, where the key exists only in map2 (i.e., the block only exists in the TV version); and (4) deleted, where the key only exists in map1(i.e., the block only exists in the smartphone version). Finally, the similarity score is calculated as:

$$similarity = max\{\frac{identical}{total-new}, \frac{identical}{total-deleted}\}$$

where:

$$total = identical + similar + new + deleted$$

Table 1 Distribution of the number of Top 10 Interaction Event for Pairs with Different APKs

Interaction	**Method**	**TV**	**Phone**
CLICK	onClick, onClick attr in XML file	103	220
TOUCH	onTouch, onTouchEvent	54	75
KEY	onKeyDown, onKeyUp, onKeyLongPress onKeyShortcut, onKeyMultiple, onKey	48	43
SCROLL	onScroll, onNestedScroll onScrollChanged, onScrollChange	31	42
FLING	onFling, onNestedFling	11	12
HOVER	onHoverEvent, onHover	10	11
LONG CLICK	onLongClick	4	5
LONG PRESS	onLongPress	3	3
DOWN	onDown	3	3
TRACK BALL	onTrackballEvent	2	1

Similarity Comparision Results. 44.6% of resource files and 50.4% of component blocks are reused on average when building multi-platform apps for mobile and television devices. *Services*, *Broadcast Receivers*, and *Content Providers* are the most commonly reused component types, where the most frequently reused package in phone/TV app pairs is *com.google.android.gms*, which is a background service and API package that provides access to a variety of Google services. During the conversion from a phone app to a TV app, *Activity* is the component type most often discarded. An Android *Activity* is a single, focused task with which a user may engage. It shows a single screen that may include widgets like buttons, text fields, and pictures. The lower the number of *Activities*, the less User Interface(UI) will be shown in Android TV apps. In general, the resources are the most altered part of the apps throughout most categories, compared with the components. Additionally, the similarity score at the code level for the Music & Audio category is quite concentrated,

which indicates that the variance for each phone/TV pair is close. This observation implies that, despite the hardware and functionality differences, a certain amount of code is migrated from the phone to the TV version without modification in the apps in *Music & Audio* category.

A closer inspection of the same file types indicates that **PNG files are the most often reused, accounting for 70.8%** of all reused files. Android devices most favor PNG files as one of the available file formats for displaying App icons, logos, and other images (Google 2022d). The high incidence of reuse of PNG files suggests that the element of User Interface is quite similar. A well-designed UI may make it simpler for users to traverse the system and perform their intended tasks (Johnson 2020). While they share a similar User Interface, the TV and mobile user experiences may vary greatly due to the devices' distinct input modalities and use scenarios. To understand how input mechanisms of user interaction between Android TV and Android phone, we then further analyze the callback events inside the app pairs. We identify the user interaction events from the following three places. (1) ***Event Handler*** When the requested action is performed on the object, the Android framework executes these methods. For example, when a View (such as a Button) is touched, the *onTouchEvent()* function on that object is invoked. To intercept this, the class has to be extended to override this method, which is referred to as Event Handlers. (2) ***Event Listener*** For easier managing interaction, Android introduces the Event Listener which is an interface where methods can be registered so that it can be triggered by the user interaction (e.g., *onTouch* method in the *android.view.View.OnTouchListener* interface can be registered by *setOnTouchListener* method in *View* class). (3) ***Event in XML*** For the CLICK interaction, it also can be implemented by declaring the corresponding method in XML layout (Google 2022f).

Event Handler and Event Listeners are used to handle the users' interaction when users trigger specific widgets. By analyzing the Event Listeners and Event Handlers in the apps, we want to discover how the logic of users' interaction differs between phone/TV apps. To this end, we decompile the bytecode to Java source code using AndroGuard (Desnos 2021), and search for the presence of Event Listeners and Event Handlers of user input provided in official Android documentation (Google 2022c). In addition, for the CLICK interaction, we further count the occurrence of *onClick* attribute that appeared in the layout (.xml) files in each app.

The top ten user interactions in phone/TV app pairs are shown in Table 1. As indicated, **the top 10 user interaction types are the same in TV and phone apps**. The most involved user interaction event on both phones and TVs is *CLICK*. On average, each phone app has 220 CLICK-related inputs, more than twice the number seen in a TV app (i.e., 103). In general, phone apps get equipped with more user interaction event than TV apps. The exceptions are the KEY-related event, which is declared on average 43 times in phone apps and 48 times in TV apps.

As mentioned before, a View is a widget generally used to display something like Buttons, ListViews, etc. To make these Views look well-organized, Android introduces the *Activity* class, where the View component can be placed in the UI by using *setContentView(View)*. Activities are primarily shown to the user as full-screen windows; however, they may alternatively be displayed as floating windows (through a

theme with the *R.attr.windowIsFloating* property set) in the Multi-Window mode, or embedded inside other windows. As a result, the number of Activities in an Android app may reflect the complexity of the UI layout to some extent. We also compare the number of Activities in phone/TV pairs for each category. The average number of Activities (i.e., 24) in smartphone apps is more than twice the number (i.e., 10) in TV apps. One possible reason may be that the larger screen size on TV could contain more content, so fewer UI screens (i.e., activities) are needed.

Instead of utilizing touch screens, TV users must rely on peripherals, such as a remote controller, to engage with TVs, which is distinct from how users interact with their phones. TOUCH-related events can be triggered on TV devices despite the lack of touch screens. The TOUCH-related events contain one parameter of type *MotionEvent* that, depending on the type of device, can report movement events (e.g., mouse, pen, finger, trackball, etc.)(Google 2023b). Using the type of object, *MotionEvent*, as a parameter, TOUCH-related events could handle various interactions and are the second most common kind of event in TV apps. Our empirical results show that the type of interaction events in the apps on both devices are comparable. The evidence above demonstrated the possibility of reuse of code for user interaction on the elements of user interface in the phone apps and their TV versions.

Table 2 Top 15 permissions in phone and TV apps for Pairs with Different APKs

Permissions in **TV** Apps	Count	Protection Level	Permissions in **Phone** Apps	Count	Protection Level
INTERNET	952 (92.0%)	Normal	INTERNET	962 (92.9%)	Normal
ACCESS_NETWORK_STATE	941 (90.9%)	Normal	ACCESS_NETWORK_STATE	961 (92.9%)	Normal
WAKE_LOCK	781 (75.5%)	Normal	WAKE_LOCK	941 (90.9%)	Normal
FOREGROUND_SERVICE	517 (50.0%)	Normal	RECEIVE[2]	874 (84.4%)	Signature
BIND_GET_INSTALL_REFERRER_SERVICE[1]	383 (37.0%)	Normal	FOREGROUND_SERVICE	833 (80.5%)	Normal
ACCESS_WIFI_STATE	312 (30.1%)	Normal	VIBRATE	586 (56.6%)	Normal
RECEIVE_BOOT_COMPLETED	274 (26.5%)	Normal	**ACCESS_COARSE_LOCATION**	550 (53.1%)	**Dangerous**
RECEIVE[2]	255 (24.6%)	Signature	**ACCESS_FINE_LOCATION**	544 (52.6%)	**Dangerous**
BILLING[3]	232 (22.4%)	Normal	GET_ACCOUNTS	514 (49.7%)	Normal
READ_EXTERNAL_STORAGE	189 (18.3%)	**Dangerous**	**READ_PHONE_STATE**	495 (47.8%)	**Dangerous**
WRITE_EXTERNAL_STORAGE	175 (16.9%)	**Dangerous**	BIND_GET_INSTALL_REFERRER_SERVICE[1]	488 (47.1%)	Normal
RECORD_AUDIO	117 (11.3%)	**Dangerous**	**WRITE_EXTERNAL_STORAGE**	451 (43.6%)	**Dangerous**
ACCESS_FINE_LOCATION	116 (11.2%)	**Dangerous**	**RECORD_AUDIO**	433 (41.8%)	**Dangerous**
ACCESS_COARSE_LOCATION	106 (10.2%)	**Dangerous**	ACCESS_WIFI_STATE	422 (40.8%)	Normal
CHANGE_WIFI_STATE	102 (9.9%)	Normal	RECEIVE_BOOT_COMPLETED	401 (38.7%)	Normal

[1] *com.google.android.finsky.permission.*BIND_GET_INSTALL_REFERRER_SERVICE
[2] *com.google.android.c2dm.permission.*RECEIVE
[3] *com.android.vending.*BILLING
All other pemissions are defined by Android, starting with *android.permission.*

Permission Analysis. The app permission framework is a critical component of the Android ecosystem since it ensures the security of Android users' privacy. According to the Android Developer Guide, permissions can be classified as *normal*, *signature*, or *dangerous* according to their riskiness (Google 2021b). *Normal* permission is lower-risk permission that would be granted at installation without approval. It provides requesting apps with access to isolated application-level features with minimal risk to other apps, the system, or the user. *Dangerous* permissions are higher-risk permissions that would give the requesting app access to private user data or control over the device, which can negatively impact the user. Due to the inherent danger associated with this type of permission, the system does not automatically grant it to the requesting app. Instead, such permissions need to be granted at run-time. *Signature* permission will only be automatically granted if the requesting app is signed with the same certificate as the app that declared the permission. In this study, we compare

the differences in the declared permissions for each phone/TV app pair to understand the kind of functionalities and resources the app may access. To this end, we utilize the Android Asset Packaging Tool (AAPT), a well-known android static analysis tool, to extract the declared permissions in the app pairs.

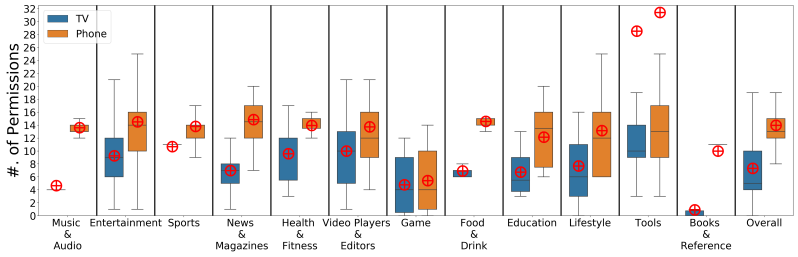


Fig. 4 Distribution of the count of the type of permissions in phone and TV apps for Pairs with Different APKs

Table 2 lists the most frequently declared permissions in phone and TV apps. The number of apps requesting network-related permissions in phone and TV apps are comparable. This makes sense as apps on both devices require Internet access. The number of apps containing other permissions, on the other hand, significantly decreases in the TV versions. For example, 550 mobile phone apps declare the permission *ACCESS_COARSE_LOCATION*, while only 106 TV apps include this permission. Figure 4 shows a category-wise breakdown of the count of permissions in each app. Overall, **TV apps declare significantly fewer permissions than phone apps**. On average, TV apps declare nearly half the number of permissions as their phone counterparts (i.e., each phone app declares 13.9 permissions while the corresponding TV version declares 7.3 permissions). Interestingly, in some categories (e.g., Game and Tools), the number of permissions declared in the TV apps is close to that in their phone versions, while some (e.g., Music & Audio and Books & References) show dramatic differences. This observation demonstrates that TV/phone app pairs in specific categories demonstrate similar functionalities and features, while in some categories, many features are removed from the TV versions.

We conducted an in-depth analysis of the permissions declared by apps in the Music & Audio category. Among TV apps in this category, four specific permissions appear in at least 80% of all apps, while the remaining permissions are each declared by fewer than 10% of these apps. Notably, these top four permissions match the leading TV permissions reported in Table 2. In contrast, smartphone apps in the Music & Audio category declare 11 permissions at rates of 88% or higher. Our findings indicate that, compared to smartphone apps, TV apps most frequently omit permissions related to Google Cloud Messaging (registering and receiving messages), reading phone state, vibrating, recording audio, accessing location, and accessing accounts. This suggests that TV apps often exclude features reliant on hardware components that are either absent or less relevant on TV devices. For example, push notifications and messaging services are commonly removed, likely because smart TVs are not designed for

real-time communication in the same way as smartphones. Similarly, location-based services are removed due to the lack of GPS hardware on most TV devices.

These findings highlight fundamental differences between TV and smartphone apps. Smartphone apps often include features that leverage mobile-specific hardware, such as GPS for location-based services, motion sensors for activity tracking, and a microphone for voice commands, audio recording, or phone calls. In contrast, TV apps focus on media consumption and user interactions via remote controls or voice assistants. TV-specific features include larger-screen-optimized interfaces, integration with external streaming devices, and remote-based navigation, which are rarely found in smartphone apps.

From a user experience perspective, the removal of certain features can have both positive and negative implications. On the positive side, eliminating unsupported permissions can streamline TV apps, reducing unnecessary resource consumption and improving security by minimizing access to sensitive data. On the negative side, some missing features, such as push notifications or account-based personalization, could limit the functionality of TV apps, making them less interactive or personalized compared to their smartphone counterparts. For example, a music streaming app on a smartphone can send reminders about new releases or recommendations based on location, whereas the TV version may lack such engagement features.

Despite these differences, we observed that some TV apps still declare permissions for features that are generally unavailable on TV hardware (e.g., camera, microphone, GPS). This could lead to unintended consequences, such as degraded performance, unexpected permission requests, or unnecessary code complexity. To mitigate such issues, our findings suggest the need for automated validation tools that analyze an app's declared permissions before it is published on the Google Play Store. These tools could help developers ensure that only relevant permissions are included, thereby enhancing both user experience and app efficiency on smart TV platforms.

Declaration of Duplicated Permissions. We then investigate how the phone/TV apps were implicated in the duplicated permission issues (i.e., the same permission is declared more than once in the manifest). According to our analysis, 366 phone apps and 31 TV apps declare the same permissions multiple times in their manifest file. Most of the TV apps with duplicated permissions (22 apps, 66.7%) are developed by the same developer, *iNmyStream*, spanning three categories, including *Entertainment*, *Music & Audio*, and *News & Magazines*. In the phone apps, package name of 91% apps with duplicated-permission begins with *com.icreo.*, but they are built by different developers. We observed that apps involve more than one type of duplicated permission. The average number of duplicated permissions in TV apps (3.4 duplicated permissions) is more than in phone apps (1 duplicated permission). There are 26 TV apps having dangerous permissions duplicated, but 6 phone apps get dangerous permissions duplicated. Among the 26 TV apps, 2 apps and 24 apps duplicated the dangerous permission, *WRITE_EXTERNAL_STORAGE*, and *READ_PHONE_STATE*, respectively. Developer *iNmyStream* provides 22 apps that duplicate the dangerous permission *READ_PHONE_STATE*. As revealed by Li et al. (2017, 2019), there is also a chance for app repackaging, especially for the apps with

a high occurrence of duplicated permissions. The above findings suggest the necessity for developers to improve their development practices.

Table 3 Distribution of Critical Vulnerabilities Found by AndroBugs for Pairs with Different APKs

Type	Detailed Vulnerability Descriptions	Music & Audio	Entertainment	Sports	Health & Fitness	News & Magazines	Food & Drink	Others	**Total**
					TV				
SSL_Security	SSL Connection	403	203	67	53	50	21	120	917
	Verifying Host Name in Custom Classes	344	9	1	0	2	15	9	380
	SSL Certificate Verification	20	23	0	0	2	2	4	51
Implicit_Intent	Implicit Service	337	43	10	30	7	2	30	459
WebView	WebView RCE Vulnerability	14	102	11	20	22	4	72	245
AndroidManifest	ContentProvider Exported	6	72	7	4	9	3	31	132
Command	Runtime Command	48	22	0	1	4	1	9	85
Permission	App Sandbox Permission	6	6	39	0	4	1	9	65
KeyStore	KeyStore Protection	0	25	1	2	2	13	10	53
Encryption	Base64 String Encryption	3	5	2	0	1	14	7	32
Fragment	Fragment Vulnerability	2	10	2	3	1	2	9	29
	Others	1	12	0	1	0	1	4	19
	Total	1184	532	140	114	104	79	314	2467
					Phone				
SSL_Security	SSL Connection	385	207	68	54	53	21	128	916
	Verifying Host Name in Custom Classes	53	12	1	1	2	0	15	84
	SSL Certificate Verification	4	15	1	1	2	2	13	38
WebView	WebView RCE Vulnerability	393	146	17	22	49	19	100	746
Implicit_Intent	Implicit Service	390	122	20	47	24	18	82	703
Fragment	Fragment Vulnerability	2	41	5	6	7	1	19	81
AndroidManifest	ContentProvider Exported	14	26	5	0	6	1	20	72
Permission	App Sandbox Permission	16	6	39	0	3	0	5	69
Command	Runtime Command	17	17	1	2	3	0	11	51
KeyStore	KeyStore Protection	0	24	2	2	3	2	12	45
Hacker	Base64 String Encryption	5	9	1	0	12	3	12	42
	Others	1	14	1	1	1	3	9	30
	Total	1280	639	161	136	165	70	426	2877

Answer to RQ1:

- *On average, 44.6% of resource files and 50.4% of component blocks are reused, with PNG files accounting for 70.8% of reused resource files.*
- *TV apps encounter fewer user interfaces than their phone counterparts, but the types of user interaction events are comparable.*
- *Phone apps tend to contain more code and utilize more permissions than their TV counterparts.*
- *Duplicated permissions exist in both phone and TV apps, and there are more TV apps involving dangerous permission duplication.*

4.2 RQ2: Security & Privacy Analysis

AndroBugs Analysis. We analyze the security and privacy risks in phone/TV pairs. We resort to AndroBugs (Lin 2015) and FlowDroid (Arzt et al. 2014) to understand the potential security vulnerabilities and privacy breaches (i.e., data leaks), respectively. We use AndroBugs to report the details of vulnerabilities discovered in each app, categorizing them as *Critical*, *Warning*, *Notice*, or *Info*, according to the harm they may bring.

Our analysis with AndroBugs reports that each app in phone/TV app pairs has at least 52 vulnerabilities. Among all the 973 phone/TV pairs, 961 phone apps, and

935 TV apps are flagged to contain critical security vulnerabilities. **Over half of the apps contain at least three critical issues**. The average number of critical issues is 3 and 2.5 for phone and TV apps, respectively. The majority of TV apps have fewer security vulnerabilities than phone apps, but in certain categories (*Music & Audio*, *Entertainment*, and *Food & Drink*), TV apps (approx. 11% of pairs) would suffer more vulnerabilities than their phone counterparts. It is interesting to observe that in more than 60% of pairs in the Food & Drink category, TV apps contain more critical vulnerabilities than phone apps. Considering the fact that TV apps contain fewer codes, the significance of critical issues in Android TV apps should not be overlooked.

Table 3 outlines the categories of critical vulnerabilities and their detailed descriptions in descending order of prevalence. Vulnerability types with fewer occurrences are classified as "Others". Now, we elaborate on three of the most common vulnerability categories in phone/TV app pairs:

- **SSL Security**: This kind of vulnerability accounts for $1,348$ (55%, $917+380+51$) and $1,038$ (36%, $916+84+38$) of all critical vulnerabilities found in the TV and phone apps, respectively. These flaws are inextricably linked to the apps' internet access mechanism. Malicious substances may be able to capture an app's information across the network if the app uses SSL incorrectly (Google 2021c). Android apps, for example, maybe vulnerable to *man-in-the-middle (MITM)* attacks if they connect to the internet without employing strong encryption (e.g., when using HTTP instead of HTTPS).
- **WebView**: In total, there are 245 (10%) and 746 (26%) vulnerabilities in this type found in TV and phone apps, respectively. The WebView vulnerabilities discovered by AndroBugs are a kind of WebView Remote Code Execution Vulnerability. It can be exploited by attackers to execute Java code in the host applications, therefore gaining access to command-line tools and posing additional security risks to users (Gao et al. 2019). For example (Thomas et al. 2015), a remote attacker may exploit a WebView to execute dynamic HTML content (written in JavaScript) and activate the Java *Runtime.exec()* API to perform commands such as *id* and *rm*.
- **Implicit Intent**: This kind of vulnerability accounts for 459 (19%) and 703 (24%) of all vulnerabilities. The Android ecosystem makes use of the Intent mechanism to facilitate the reuse of functionality (Enumeration 2017). However, implicit Intents may be intercepted by other components (Octeau et al. 2013). Therefore, it is a potential security concern since attackers may leverage implicit Intents to get access to sensitive information, posing threats to users' privacy.

As demonstrated in Table 3, *SSL_Security* is the most frequently detected critical vulnerability type in both phone and TV apps. The type of the second and the third most detected vulnerabilities are similar but ranked in reverse order (i.e., *Implicit_Intent* and *WebView* for TV apps, and *WebView* and *Implicit_Intent* for phone apps). Among the SSL_Security issues, the apps impacted by the vulnerability, *Verifying Host Name in Custom Classes*, would let attackers exploit a valid certificate to conduct MITM attacks without the users' awareness. As shown in Table 3, the TV versions in the app pairs are more likely to suffer from this vulnerability, with 84 and

380 vulnerabilities occurring in the phone and TV versions, respectively. This observation indicates that certain apps' TV versions are at significant security risk while the phone versions are far less vulnerable. The app designed for different phone and TV platforms can be implicated in a variety of different sorts of security vulnerabilities.

FlowDroid Analysis. To further spot potential privacy breaches in Android TV apps, we use FlowDroid (Arzt et al. 2014) to examine the collected app pairs. FlowDroid identifies data flows from sensitive sources to potentially dangerous sinks, where sources are methods through which such data flows enter the app and sinks are the methods from which they leave the app. To detect data leaks, we used the default sources and sinks provided by FlowDroid. For each of these apps, we set a 1.5-hour timeout and 64 GB memory limit. Our results show that FlowDroid successfully finished scanning 785 phone apps and 910 TV apps. Approximately 39% (307 apps) of the 785 phone apps and 56% (512 apps) of the 910 TV apps contain at least one privacy leak, revealing that TV apps are more implicated by privacy issues than their phone counterparts.

Table 4 Distribution of Sources Detected more than 20 times by FlowDroid for 755 Pairs with Different APKs

Sources	Music & Audio	Entertainment	Food & Drink	Sports	Game	Lifestyle	Others	**Total**
				TV				
android.content.pm.PackageManager.queryIntentServices	1197	13	0	14	0	13	22	1259
android.database.Cursor.getString	25	117	8	25	5	6	68	254
android.location.Location.getLatitude	0	84	0	29	22	24	35	194
android.location.Location.getLongitude	0	84	0	27	22	24	35	192
java.util.Locale.getCountry	5	2	115	0	9	0	2	133
java.net.HttpURLConnection.getInputStream	8	46	1	11	15	6	34	121
java.net.URLConnection.getInputStream	36	10	0	0	0	2	3	51
android.content.pm.PackageManager.queryIntentActivities	0	6	0	8	2	3	11	30
android.content.pm.PackageManager.queryBroadcastReceivers	0	5	0	7	2	4	5	23
Others	3	8	0	2	6	0	6	25
Total	1274	375	124	123	83	82	221	2282
				Phone				
android.database.Cursor.getString	166	234	4	31	4	17	409	865
android.location.Location.getLongitude	40	223	80	40	9	66	251	709
android.location.Location.getLatitude	41	222	79	40	9	65	251	707
android.content.pm.PackageManager.queryIntentServices	75	78	0	25	0	31	126	335
java.net.HttpURLConnection.getInputStream	13	104	11	11	20	9	59	227
java.util.Locale.getCountry	0	9	88	0	10	10	33	150
android.content.pm.PackageManager.queryIntentActivities	4	14	3	5	2	10	17	55
android.accounts.AccountManager.getAccounts	0	0	0	0	0	15	36	51
android.content.pm.PackageManager.queryBroadcastReceivers	0	18	0	7	2	4	9	40
android.view.View findViewById	0	30	0	0	0	0	0	30
Others	25	16	4	2	10	13	25	95
Total	364	948	269	161	66	240	1216	3264

Sensitive information is **mostly leaked through the built-in Logcat functionality that developers typically employ for debugging**. Users' data may leak from the logs, as attackers may obtain sensitive user information by inspecting the logs on the corresponding device. The second most frequently detected sink is the Bundle object. It is used to transport data between activities, processes, and configuration updates. A large amount of information contained in Bundle objects is exploitable by attackers in both phone/TV apps.

The **average number of sources detected in each phone app (**4.3 **sources) is greater than its TV version (**3 **sources).** Given the larger codebase of mobile apps compared to TV apps, it comes as no surprise to find a greater number of potential

leaks identified in mobile apps. In general, phone apps have more sources than their TV counterparts, except for the pairs in the *Music & Audio* and *Game* categories. TV apps in *Music & Audio category* have an average of 3.2 sources, compared to only 0.9 sources in their phone counterpart. Table 4 lists the sources detected by FlowDroid in the phone/TV app pairs. The sources detected with fewer than 20 occurrences are grouped in the *Others* category. *Android.content.pm.PackageManager. queryIntentServices* is the most leaked source (i.e., 1,259 occurrence) among TV apps, contributing to more than half of the sources of the leaks (i.e., 2,282 occurrence). Many apps utilize *queryInterntServices* to get services that may match a certain intent and then wake these services up depending on the return values of *queryIntentServices*, which are solely used for inter-app communication (Xu et al. 2017). The *getString* method in the dataset package is the most leaked source in phone apps and the second most leaked source in TV apps, which is unsurprising as the database often involves vast amounts of data, hence having more chance to leak the data. The geographic location of the device is also a common source of data leaks. Although most TV devices do not have a GPS module, WiFi or network information can be used to approximate the device's location. In contrast to phone devices, the geographical locations of Android TVs are often fixed and expose users' home addresses. Attackers may utilize this information to get a wealth of additional information on users or even to pose physical threats (Tileria and Blasco 2022; Wijesekera et al. 2015).

Answer to RQ2:

- *Security vulnerabilities are frequently encountered in varying forms between the TV and phone versions of the same app.*
- *Potential privacy leak severity differs depending on device type and app category.*
- *The built-in Logcat functionality is the main leaking source for both phone/TV apps.*

5 Threats To Validity

The dataset of phone/TV app pairs might not fully represent the entire population. When searching for these app pairs in the Google Play store, we simulated specific devices, influenced by factors such as device type, OS versions, and hardware architectures. This approach may have limited our ability to include all relevant apps, potentially excluding some from our collection. However, we used AndroZoo, one of the most comprehensive datasets available, to gather package names for these app pairs, highlighting the extent of our dataset. We chose to focus exclusively on free apps because they are accessible to a broader user base, whereas paid apps would involve costs. Moreover, other widely recognized datasets for Android apps, such as AndroZoo and F-droid, also concentrate solely on free apps.

Tools and analyzers involved in the comparative study could generate false positives. Our comparative study employed a range of sophisticated tools to analyze app similarity, security, and privacy aspects. While these tools are advanced, it's crucial to acknowledge their inherent limitations, including the potential for false positives. Although a comprehensive verification of our findings would address these issues,

it falls outside the scope of this study. However, we mitigated the impact of these limitations on our conclusions by consistently applying these tools to both TV and phone apps. This approach ensured that any inherent limitations or inaccuracies in the tools affected both app types equally, thereby maintaining the validity of our comparative analysis. Future research could build upon our work by incorporating more extensive verification processes to further refine the accuracy of these findings.

6 Discussion

Possibility of automated migration for user interface. As unveiled in our empirical findings, 44.6% of the resource files and 50.6% of components in TV apps are shared with their counterparts on mobile devices. Given that 70.8% of all reused files are PNG files and that the sorts of interaction events in apps for both platforms are comparable, the User Interface elements of phone apps and TV apps can be exploited further to enable or assist conversion of Android phone apps to Android TV apps. Given the disparity in market share between phone/TV apps, there is a high demand for the TV versions of phone apps. The creation of an automated process to transfer GUIs of phone apps to TVs would decrease the workload for developers and enrich the Android ecosystem. Hu et al. (2023), has synthesized a list of rules for grouping and classifying phone Graphical User Interfaces (GUIs), converting them to TV GUIs. Those rules are based on GUIs extracted from 298 Android phone/TV app pairs where those pairs share the identical artifact. It is worth noting that while the dataset from Hu et al. (2023) (SHA256) does not overlap with ours, there is a complete overlap in app package names. This suggests that the datasets include different versions of the same apps, reflecting various release iterations. However, our work prepares market-scale app pairs with clear setups, rather than pairs with identical artifacts, and insights were gleaned from our quantitative and qualitative analyses, making our results generalizable to pairs within the Android market. By leveraging our dataset, rules can be synthesized from the comprehensive collection of datasets or specifically from certain categories of app pairs for automatically adapting mobile GUIs to TV GUIs.

Investigation of security/privacy issues on TV-specific apps. Liu et al. (2021) and Tileria and Blasco (2022) investigated security risks in TV apps.However, these studies overlooked the implications of Android App Bundles and Multiple APK support in their datasets. As a result, their research was compromised by the inclusion of apps containing artifacts from both phone and TV versions, which led to a blurring of platform-specific characteristics. This oversight potentially diminished the accuracy and specificity of their findings regarding TV app ecosystems. It is crucial to conduct TV-specific security/privacy app analysis. Our comparison of phone/TV app pairs revealed that TV apps have a smaller size, with an average of 4.5 Mb less than their phone counterparts, and simpler functionality, including hardware-related features, interaction events, and screens. Although Android mobile apps and TV apps share some common Software Development Kit (SDK), there are differences originating from their device nature, and the analyzer is mainly designed for Android mobile

apps instead of TV apps, potentially neglecting the TV-specific SDK. However, in certain categories of TV apps, the number of security and privacy concerns is comparable to, or even more pronounced than, those in phone apps, highlighting potential issues in the TV versions of Android apps. Future work should focus on conducting TV-specific analyses or investigating category-specific TV apps to enhance the security of the TV ecosystem. Understanding the differences between mobile and TV SDKs is crucial for designing TV app analyzers or adapting mobile app analyzers for TV apps. For instance, taint analysis relies heavily on source and sink labels extracted from the SDK of Android mobile apps, necessitating the exploration of TV-specific SDKs to migrate current mobile app analyzers to TV apps. Additionally, future research should investigate the causes of security and privacy issues in TV apps and develop targeted solutions and best practices to improve their overall security and privacy framework.
Classification of artifacts for different platform. The validity of platform-sensitive findings is undermined when performing static-analysis experiments on APKs containing artifacts for multiple platforms. Prior studies have failed to account for the presence of artifacts from other platforms in the apps they analyzed, which can result in platform-sensitive conclusions drawn from static analysis that are less precise and even misleading to researchers and prectioners. For example, static analysis outcomes for app compatibility and security concerns on a specific platform can be influenced by code intended for non-targeted platforms. Therefore, it is essential to develop a solution that can automatically differentiate or identify artifacts specific to distinct platforms within a single APK.

7 Related Work

Large-Scale Android App Analysis. Previous research has extensively examined various aspects of Android apps, particularly focusing on phone applications. Studies have analyzed app metadata, security, and privacy issues on a large scale (Wang et al. 2018; Fan et al. 2018; Cai et al. 2019; Liu et al. 2019; Hu et al. 2020; Lin et al. 2020; Mauthe et al. 2021; Dong et al. 2018). For instance, Mauthe et al. (2021) performed a large-scale empirical study of the decompilation success rate for Android apps and the authors discovered that code obfuscation is quite rarely encountered, even in malicious applications. Wang et al. (2018) conducted an extensive study on 6 million Android apps downloaded from 17 different app markets to understand catalog similarity across app stores. Chen et al. (2021) collected 223 pairs of Android Smartphone and Smartwatch apps and examined them from both non-code and code aspects to understand the relationship between them. Recent research has begun to explore TV app ecosystems, expanding beyond the traditional focus on mobile applications. Notably,Aafer et al. (2021) employed dynamic fuzzing to identify vulnerabilities in Android TV boxes, while Liu et al. (2021) and Tileria and Blasco (2022) examined security risks specific to TV apps. However, these studies have notable limitations. They typically concentrated solely on TV apps, neglecting to consider their relationship with phone counterparts. Moreover, previous works failed to account for the impact of Android App Bundles and Multiple APK support in their datasets. This oversight resulted in research that inadvertently included apps with artifacts from both phone and TV versions, thus

conflating platform-specific characteristics. Our study addresses these shortcomings by clearly differentiating between apps with TV and phone artifacts, providing empirical evidence specifically for device-specific releases. This refined approach enables us to more accurately identify and analyze the unique features and security challenges inherent to each platform.

Android security. Android is one of the most popular open-source operating systems, particularly on smartphones (Liu et al. 2021). With this widespread usage, an increasing number of Android applications are developed. This rising popularity, on the other hand, draws malware developers and exposes Android users to a growing number of security threats (Faruki et al. 2014). Malware app developers exploit platform vulnerabilities and steal private user data for profit. The empirical study conducted by Linares-Vásquez et al. (2017) examined 660 Android-related vulnerabilities such as memory corruption and data handling and discovered that Android vulnerabilities survive for a long period of time (at least 724 days, on average) in the codebase. Thus, vulnerability detection and analysis have garnered considerable attention from the academic community, and numerous relevant research works have been proposed to improve software security (Palomba et al. 2017; Wang et al. 2020; Zhang et al. 2019; Chen et al. 2019; Li et al. 2021). For example, Zhan et al. (2021) proposed an obfuscation-resilient third-party library detection tool that can precisely locate vulnerable third-party libraries used in Android applications. Apart from that, Android malware detection and defense have also attracted increasing research attention (Qiu et al. 2020). To better analyze apps' features and behaviors, industry and academia have proposed a variety of security and privacy analyzers for Android, including static (Arzt et al. 2014; Li et al. 2015b), dynamic (Tam et al. 2015; Mao et al. 2016) and hybrid (Lindorfer et al. 2014). Zhou *et al.* Zhou et al. (2012) developed a rule-based static malware detection system (i.e., permission-based behavioral footprinting and heuristics-based filtering) to detect new malware samples. In recent years, machine learning and deep learning techniques have been widely introduced to detect Android malware and demonstrated promising performance (Arp et al. 2014; Mariconti et al. 2017; McLaughlin et al. 2017; Zhao et al. 2021). For instance, McLaughlin et al. (2017) employed a deep convolutional neural network (CNN) to construct an Android malware detector and their experiments demonstrated that the proposed model can achieve a 98% detection accuracy. To the best of our knowledge, previous works mainly target the security of smartphone apps. We investigate the security and privacy risks in Android phone/TV pairs. As our future work, we plan to go beyond this work by inventing better approaches for improving the security and reliability of Android TV apps.

Android Control-flow modeling Static analysis of Android applications faces inherent challenges due to the platform's event- and callback-driven architecture, which complicates traditional control-flow modeling techniques Cao et al. (2015); Samhi et al. (2024). FlowDroid addresses these challenges through a dual-layered approach: it combines Soot's program analysis capabilities with Android-specific optimizations for lifecycle, callback, and asynchronous execution modeling, establishing itself as the de facto standard for call graph (CG) construction and taint analysis Li et al. (2015a); Samhi et al. (2021). A series of research has expanded FlowDroid's

capabilities across domains such as reflection (Barros et al. (2015); Li et al. (2016)), cross-language interactions (Samhi et al. (2022)), and framework-specific behaviors (e.g., React Native (Liu et al. (2023))), demonstrating incremental CG completeness improvements of 5–50% in specialized contexts. Collaborative integration of these advancements into FlowDroid's open-source ecosystem has solidified its role as a foundational tool for control- and data-flow analysis (Samhi et al. (2024)). While FlowDroid's generalized modeling suffices for mobile apps, Android TV-specific paradigms—such as Leanback library lifecycles, ambient mode transitions, and remote input handling—remain unaddressed. Extending prior work on external framework modeling (Liu et al. (2023)) and cross-language interoperability (Samhi et al. (2022)) could enhance CG precision for TV apps, particularly for background recommendation updates and multi-device casting workflows.

Android GUI modeling involves representing and analyzing the flow of an application's user interface (UI), specifically focusing on how different GUI components transition from one to another. The evolution of this domain has progressed through several significant theoretical and practical advancements: initially, Azim and Neamtiu's Activity Transition Graph (ATG) (Azim and Neamtiu (2013)) established the foundation for static GUI modeling, albeit limited to activity-level transitions; subsequently, Gator's Window Transition Graph (WTG) (Yang et al. (2018)) enhanced this framework by incorporating menu items and widgets, though excluding fragments, drawers, services, and broadcast receivers. Further research contributions (Wang and Rountev (2016); Wu et al. (2016); Zhang et al. (2018)) augmented WTG through the integration of API invocation for performance and energy consumption analysis. Further refinements are marked by StoryDroid (Chen et al. (2019, 2022); Zhang et al. (2023)), which extended ATG with ICC message and fragment support with the feature of static UI rendering capabilities. Moreover, the Window Transition Graph - Element (WTG-E) (Lai and Rubin (2019)), expanded coverage to drawers and broadcast receivers. However, these models' treatment of UI components as separate nodes and their insufficient detail for executable path extraction limited their applicability in goal-driven exploration scenarios. GOALEXPLORER (Lai and Rubin (2019)) addressed these limitations by introducing the Screen Transition Graph (STG), innovating through screen-centric composition as node representation to facilitate the generation of executable scripts for user-defined functionality testing. Throughout the above progression, each iteration refined both UI component representation granularity and transition modeling sophistication while maintaining a graph-based structural paradigm where UI serves as nodes connected by transition edges. Android TV's unique UI paradigms—hierarchical channel grids, D-pad navigation, and 10-foot display constraints—necessitate novel graph-based models that extend STG/ATG frameworks. Integrating TV-specific UI states (e.g., media carousels) with control-flow advancements in ICC (Samhi et al. (2021)) and asynchronous execution could enable static analysis of casting workflows and Leanback fragment transitions, while tools like StoryDroid (Chen et al. (2019)) could adapt to simulate TV layout safety zones.

8 Conclusion

In this study, we conducted the first comprehensive comparative analysis of Android phone and TV app pairs to understand their relationship. We curated and examined market-level Android phone/TV app pairs, analyzing their similarities from both non-code and code perspectives. Our evaluation of interaction events revealed comparable user engagement patterns across phone/TV pairs, suggesting potential for prioritizing app migration from phones to TVs. Furthermore, we investigated the security and privacy implications of these app pairs. Our findings indicate that TV apps face largely overlooked security and privacy issues. Notably, despite phone apps generally containing more code, certain TV app categories exhibit comparable or even higher numbers of security and privacy concerns. This research aims to assist developers and researchers in examining domain-specific issues for TV apps and provide novel insights for future work in this area.

9 Statements and Declarations

Yonghui Liu and John Grundy are supported by ARC Laureate Fellowship FL190100035.

References

Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: Collecting millions of android apps for the research community. In: MSR (2016)

Allix, K.: AndroZoo. (2021). https://androzoo.uni.lu/

Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of android apps. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, pp. 641–660 (2013)

APKMirror (2021). https://www.apkmirror.com/

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices **49**(6), 259–269 (2014)

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of android malware in your pocket. In: Ndss, vol. 14, pp. 23–26 (2014)

Aafer, Y., You, W., Sun, Y., Shi, Y., Zhang, X., Yin, H.: Android {SmartTVs} vulnerability discovery via {Log-Guided} fuzzing. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 2759–2776 (2021)

Barros, P., Just, R., Millstein, S., Vines, P., Dietl, W., d'Amorim, M., Ernst, M.D.: Static analysis of implicit control flow: Resolving java reflection and android intents (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 669–679 (2015). IEEE

Chen, X., Chen, W., Liu, K., Chen, C., Li, L.: A comparative study of smartphone and smartwatch apps. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing, pp. 1484–1493 (2021)

Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., Chen, Y.: Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In: NDSS (2015)

Chen, S., Fan, L., Chen, C., Su, T., Li, W., Liu, Y., Xu, L.: Storydroid: Automated generation of storyboard for android apps. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 596–607 (2019). IEEE

Chen, S., Fan, L., Chen, C., Liu, Y.: Automatically distilling storyboard with rich features for android apps. IEEE Transactions on Software Engineering **49**(2), 667–683 (2022)

Chen, X., Li, C., Wang, D., Wen, S., Zhang, J., Nepal, S., Xiang, Y., Ren, K.: Android hiv: A study of repackaging malware for evading machine-learning detection. TIFS (2019)

Cai, H., Zhang, Z., Li, L., Fu, X.: A large-scale study of application incompatibilities in android. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 216–227 (2019)

Desnos, A.: Androguard. [Online]. Available: https://github.com/androguard/androguard. Last checked 02 March 2022 (2021)

Dong, S., Li, M., Diao, W., Liu, X., Liu, J., Li, Z., Xu, F., Chen, K., Wang, X., Zhang, K.: 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, pp. 172–192 (2018). Springer

Enumeration, C.W.: Use of implicit intent for sensitive communication (2017)

Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M.S., Conti, M., Rajarajan, M.: Android security: a survey of issues, malware penetration, and defenses. IEEE communications surveys & tutorials **17**(2), 998–1022 (2014)

Fan, L., Su, T., Chen, S., Meng, G., Liu, Y., Xu, L., Pu, G., Su, Z.: Large-scale analysis of framework-specific exceptions in android apps. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 408–419 (2018).

IEEE

Gao, J., Li, L., Kong, P., Bissyandé, T.F., Klein, J.: Understanding the evolution of android app vulnerabilities. IEEE Transactions on Reliability (2019)

Google: Google Play Python API. (2021). https://github.com/NoMore201/googleplay-api

Google, D.: Android Developer Guide. (2021). https://developer.android.com/reference/android/Manifest.permission

Google, D.: Security SSL. (2021). https://developer.android.com/training/articles/security-ssl

Google, D.: About Android App Bundles. (2022). https://developer.android.com/guide/app-bundle

Google, D.: About Android App Bundles. (2022). https://developer.android.com/google/play/publishing/multiple-apks

Google, D.: Android.view. (2022). https://developer.android.com/reference/android/view/package-summary

Google, D.: Drawables Overview. (2022). https://developer.android.com/develop/ui/views/graphics/drawables

Google, D.: Handle TV Hardware. (2022). https://developer.android.com/training/tv/start/hardware

Google, D.: Input Events Overview. (2022). https://developer.android.com/guide/topics/ui/ui-events

Google: Get Started with TV Apps. (2023). https://developer.android.com/training/tv/start/start#dev-project

Google, D.: MotionEvent. (2023). https://developer.android.com/reference/android/view/MotionEvent

Group, I.: Smart TV Market: Global Industry Trends, Share, Size, Growth, Opportunity and Forecast (2021). https://www.researchandmarkets.com/reports/5311939/smart-tv-market-global-industry-trends-share?w=4

Hu, H., Dong, R., Grundy, J., Nguyen, T.M., Liu, H., Chen, C.: Automated mapping of adaptive app guis from phones to tvs. ACM Transactions on Software Engineering and Methodology **33**(2), 1–31 (2023)

Hu, Y., Wang, H., He, R., Li, L., Tyson, G., Castro, I., Guo, Y., Wu, L., Xu, G.: Mobile app squatting. In: Proceedings of The Web Conference 2020, pp. 1727–1738 (2020)

Johnson, J.: Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Guidelines. Morgan Kaufmann, ??? (2020)

Laricchia, F.: Android TV Continues Its Growth W/ 7,000 Apps. (2022). https://9to5google.com/2020/08/10/android-tv-growth-apps-users-operators/

Laricchia, F.: Number of Smart TV Users in the United States from 2016 to 2022 (in Millions)*. (2022). https://www.statista.com/statistics/718737/number-of-smart-tv-users-in-the-us/

Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: Iccta: Detecting inter-component privacy leaks in android apps. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 280–291 (2015). IEEE

Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: Iccta: Detecting inter-component privacy leaks in android apps. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 280–291 (2015). https://doi.org/10.1109/ICSE.2015.48

Li, L., Bissyandé, T.F., Klein, J.: Simidroid: Identifying and explaining similarities in android apps. In: The 16th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom 2017) (2017)

Li, L., Bissyandé, T.F., Klein, J.: Rebooting research on detecting repackaged android apps: Literature review and benchmark. IEEE Transactions on Software Engineering (TSE) (2019)

Li, L., Bissyandé, T.F., Octeau, D., Klein, J.: Droidra: Taming reflection to support whole-program analysis of android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 318–329 (2016)

Liu, Y., Chen, X., Liu, P., Grundy, J., Chen, C., Li, L.: Reunify: A step towards whole program analysis for react native android apps. In: 2023 IEEE/ACM International Conference on Automated Software Engineering (2023)

Li, C., Chen, X., Wang, D., Wen, S., Ahmed, M.E., Camtepe, S., Xiang, Y.: Backdoor attack on machine learning based android malware detectors. IEEE Transactions on Dependable and Secure Computing (2021)

Lin, Y.-C.: Androbugs framework: An android application security vulnerability scanner. Blackhat Europe **2015** (2015)

Li, L., Li, D., Bissyandé, T.F., Klein, J., Le Traon, Y., Lo, D., Cavallaro, L.: Understanding android app piggybacking: A systematic study of malicious code grafting. TIFS (2017)

Liu, Y., Li, L., Kong, P., Sun, X., Bissyandé, T.F.: A first look at security risks of android tv apps. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), pp. 59–64 (2021). IEEE

Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Van Der Veen, V., Platzer, C.: Andrubis–1,000,000 apps later: A view on current android malware behaviors. In: 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), pp. 3–17 (2014). IEEE

Lai, D., Rubin, J.: Goal-driven exploration for android applications. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 115–127 (2019). IEEE

Lin, J.-W., Salehnamadi, N., Malek, S.: Test automation in open-source android apps: A large-scale empirical study. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 1078–1089 (2020)

Liu, Y., Tantithamthavorn, C., Li, L., Liu, Y.: Deep learning for android malware defenses: a systematic literature review. arXiv preprint arXiv:2103.05292 (2021)

Linares-Vásquez, M., Bavota, G., Escobar-Velásquez, C.: An empirical study on android-related vulnerabilities. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 2–13 (2017). IEEE

Liu, T., Wang, H., Li, L., Bai, G., Guo, Y., Xu, G.: Dapanda: Detecting aggressive push notifications in android apps. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 66–78 (2019). IEEE

Mao, K., Harman, M., Jia, Y.: Sapienz: Multi-objective automated testing for android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 94–105 (2016)

Mauthe, N., Kargén, U., Shahmehri, N.: A large-scale empirical study of android app decompilation. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 400–410 (2021). IEEE

McLaughlin, N., Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupé, A., *et al.*: Deep android malware detection. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp. 301–308 (2017)

Mariconti, E., Onwuzurike, L., Andriotis, P., Cristofaro, E.D., Ross, G.J., Stringhini, G.: Mamadroid: Detecting android malware by building markov chains of behavioral models. In: 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017. The Internet Society, ??? (2017)

Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Le Traon, Y.: Effective {Inter-Component} communication mapping in android: An essential step towards holistic security analysis. In: 22nd USENIX Security Symposium (USENIX Security 13), pp. 543–558 (2013)

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A.: Lightweight detection of android-specific code smells: The adoctor project. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 487–491 (2017). IEEE

View and restrict your app's compatible devices (2021). https://support.google.com/googleplay/android-developer/answer/7353455

Qiu, J., Zhang, J., Luo, W., Pan, L., Nepal, S., Xiang, Y.: A survey of android malware detection with deep neural models. ACM Computing Surveys (CSUR) **53**(6), 1–36 (2020)

Samhi, J., Bartel, A., Bissyandé, T.F., Klein, J.: Raicc: Revealing atypical inter-component communication in android apps. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1398–1409 (2021). IEEE

Samhi, J., Gao, J., Daoudi, N., Graux, P., Hoyez, H., Sun, X., Allix, K., Bissyandé, T.F., Klein, J.: Jucify: a step towards android code unification for enhanced static analysis. In: Proceedings of the 44th International Conference on Software Engineering, pp. 1232–1244 (2022)

Samhi, J., Just, R., Bissyandé, T.F., Ernst, M.D., Klein, J.: Call graph soundness in android static analysis. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 945–957 (2024)

Tileria, M., Blasco, J.: Watch over your tv: A security and privacy analysis of the android tv ecosystem. Proceedings on Privacy Enhancing Technologies **3**, 692–710 (2022)

Thomas, D.R., Beresford, A.R., Coudray, T., Sutcliffe, T., Taylor, A.: The lifetime of android api vulnerabilities: case study on the javascript-to-java interface. In: Cambridge International Workshop on Security Protocols, pp. 126–138 (2015). Springer

Tam, K., Fattori, A., Khan, S., Cavallaro, L.: Copperdroid: Automatic reconstruction of android malware behaviors. In: NDSS Symposium 2015, pp. 1–15 (2015)

Google Play TV apps (2024). https://play.google.com/store/apps?device=tv&hl=en&gl=US

Wijesekera, P., Baokar, A., Hosseini, A., Egelman, S., Wagner, D., Beznosov, K.: Android permissions remystified: A field study on contextual integrity. In: 24th

USENIX Security Symposium (USENIX Security 15), pp. 499–514 (2015)

Wang, H., Liu, Z., Liang, J., Vallina-Rodriguez, N., Guo, Y., Li, L., Tapiador, J., Cao, J., Xu, G.: Beyond google play: A large-scale comparative study of chinese android app markets. In: Proceedings of the Internet Measurement Conference 2018, pp. 293–307 (2018)

Wang, Y., Rountev, A.: Profiling the responsiveness of android applications via automated resource amplification. In: Proceedings of the International Conference on Mobile Software Engineering and Systems, pp. 48–58 (2016)

Wang, Y., Xu, G., Liu, X., Mao, W., Si, C., Pedrycz, W., Wang, W.: Identifying vulnerabilities of ssl/tls certificate verification in android apps with static and dynamic analysis. Journal of Systems and Software **167**, 110609 (2020)

Wu, H., Yang, S., Rountev, A.: Static detection of energy defect patterns in android applications. In: Proceedings of the 25th International Conference on Compiler Construction, pp. 185–195 (2016)

Xu, M.e., othersMa, Y., Liu, X., Lin, F.X., Liu, Y.: Appholmes: Detecting and characterizing app collusion among third-party android markets. In: Proceedings of the 26th International Conference on World Wide Web, pp. 143–152 (2017)

Yang, S., Wu, H., Zhang, H., Wang, Y., Swaminathan, C., Yan, D., Rountev, A.: Static window transition graphs for android. Automated Software Engineering **25**, 833–873 (2018)

Zhang, R., Chen, X., Wen, S., Zheng, J.: Who activated my voice assistant? a stealthy attack on android phones without users' awareness. In: International Conference on Machine Learning for Cyber Security, pp. 378–396 (2019). Springer

Zhan, X., Fan, L., Chen, S., Wu, F., Liu, T., Luo, X., Liu, Y.: Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1695–1707 (2021). IEEE

Zhang, X., Fan, L., Chen, S., Su, Y., Li, B.: Scene-driven exploration and gui modeling for android apps. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1251–1262 (2023). IEEE

Zhao, Y., Li, L., Wang, H., Cai, H., Bissyandé, T.F., Klein, J., Grundy, J.: On the impact of sample duplication in machine-learning-based android malware detection. TOSEM **30**(3), 1–38 (2021)

Zhang, Y., Sui, Y., Xue, J.: Launch-mode-aware context-sensitive activity transition analysis. In: Proceedings of the 40th International Conference on Software Engineering, pp. 598–608 (2018)

Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: NDSS, vol. 25, pp. 50–52 (2012)