Published: 12 July 2024
Accepted: 16 April 2024
Received: 28 September 2023

RESEARCH-ARTICLE

Investigating Documented Privacy Changes in Android OS

CHUAN YAN, The University of Queensland, Brisbane, QLD, Australia

MARK HUASONG MENG, National University of Singapore, Singapore City, Singapore

FUMAN XIE, The University of Queensland, Brisbane, QLD, Australia

GUANGDONG BAI, The University of Queensland, Brisbane, QLD, Australia

Proceedings of the ACM on Software Engineering, Volume 1, Issue FSE (July 2024)
https://doi.org/10.1145/3660826
EISSN: 2994-970X

Investigating Documented Privacy Changes in Android OS

CHUAN YAN, The University of Queensland, Australia
MARK HUASONG MENG, National University of Singapore, Singapore and Institute for Infocomm Research at A*STAR, Singapore
FUMAN XIE, The University of Queensland, Australia
GUANGDONG BAI*, The University of Queensland, Australia

Android has empowered third-party apps to access data and services on mobile devices since its genesis. This involves a wide spectrum of user privacy-sensitive data, such as the device ID and location. In recent years, Android has taken proactive measures to adapt its access control policies for such data, in response to the increasingly strict privacy protection regulations around the world. When each new Android version is released, its privacy changes induced by the version evolution are transparently disclosed, and we refer to them as *documented privacy changes* (DPCs). Implementing DPCs in Android OS is a non-trivial task, due to not only the dispersed nature of those access control points within the OS, but also the challenges posed by backward compatibility. As a result, whether the actual access control enforcement in the OS implementations aligns with the disclosed DPCs becomes a critical concern.

In this work, we conduct the first systematic study on the consistency between the *operational behaviors* of the OS at runtime and the *officially disclosed DPCs*. We propose DopCheck, an automatic DPC-driven testing framework equipped with a large language model (LLM) pipeline. It features a serial of analysis to extract the ontology from the privacy change documents written in natural language, and then harnesses the few-shot capability of LLMs to construct test cases for the detection of *DPC-compliance issues* in OS implementations. We apply DopCheck with the latest versions (10 to 13) of Android Open Source Project (AOSP). Our evaluation involving 79 privacy-sensitive APIs demonstrates that DopCheck can effectively recognize DPCs from Android documentation and generate rigorous test cases. Our study reveals that the *status quo* of the DPC-compliance issues is concerning, evidenced by 19 bugs identified by DopCheck. Notably, 12 of them are discovered in Android 13 and 6 in Android 10 for the first time, posing more than 35% Android users to the risk of privacy leakage. Our findings should raise an alert to Android users and app developers on the DPC compliance issues when using or developing an app, and would also underscore the necessity for Google to comprehensively validate the actual implementation against its privacy documentation prior to the OS release.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Android, privacy, testing, documentation

ACM Reference Format:
Chuan Yan, Mark Huasong Meng, Fuman Xie, and Guangdong Bai. 2024. Investigating Documented Privacy Changes in Android OS. *Proc. ACM Softw. Eng.* 1, FSE, Article 119 (July 2024), 24 pages. https://doi.org/10.1145/3660826

*Corresponding author.

Authors' Contact Information: Chuan Yan, The University of Queensland, Australia, uqcyan3@uq.edu.au; Mark Huasong Meng, National University of Singapore, Singapore and Institute for Infocomm Research at A*STAR, Singapore, huasong.meng@u.nus.edu; Fuman Xie, The University of Queensland, Australia, fuman.xie@uqconnect.edu.au; Guangdong Bai, The University of Queensland, Australia, g.bai@uq.edu.au.

ACM 2994-970X/2024/7-ART119
https://doi.org/10.1145/3660826

1 INTRODUCTION

Android has reserved over 70% market share of smartphone devices, according to a recent report by Statista [62]. This remarkable success can be attributed in part to its rapid evolution. It has been renowned for its continuous innovation, constantly introducing new features and technology advancements, positioning it as one of the most cutting-edge mobile operation systems (OSes). It boasts a flourishing ecosystem of over 2.6 million third-party applications (apps) that greatly enhance productivity and enrich users' daily lives [7]. These apps are enabled to access data on users' devices to fulfill their rich functionalities. However, many of them have been reported to excessively collect personally identifiable information (PII), ranging from email addresses to physical locations, purportedly for user experience enhancement and personalization [40, 52, 57, 75].

In response to this challenge, updating the access control mechanisms over privacy-sensitive data is a key component in Android's evolution journey. Over the years, Android has implemented a series of measures to enhance user privacy. Typical examples include the transition from granting permissions during installation to runtime permissions request in Android 6 [32], and the introduction of one-time permissions in Android 11 [31]. In recent years, Android keeps strengthening its privacy framework by enforcing new privacy features to restrict apps' access to user data. This aligns with the current global user privacy landscape, as many countries have enacted legislation to regulate the collection and use of personal data [13, 56, 61, 65], such as the well-known General Data Protection Regulation (GDPR) of European Union [65]. In 2018 when the GDPR was introduced, Android 10 ceased all third-party apps' access to non-resettable device identifiers [18]. Since then, Google has adopted a proactive and transparent approach, providing comprehensive documentation on security and privacy updates with each new Android OS release [19, 21, 24, 27]. We refer to them as *documented privacy changes* (DPCs). This initiative aims to raise developers' and users' awareness of privacy protection on Android.

The research community has conducted a comprehensive investigation into privacy protection on Android. Numerous efforts have been taken to scrutinize the OS-level safeguard mechanisms [48, 50, 58, 60] and analyze data harvesting behaviors of third-party apps [15, 57, 77]. However, these provide only a partial perspective of the privacy protection within the Android ecosystem, leaving the complementary question of *whether the evolution of Android privacy mechanisms complies with the officially disclosed DPCs* still unsolved.

Sustaining such compliance presents a formidable challenge for mechanisms based on existing code review and static analysis techniques [49, 63], given that the Android evolution over a decade has made it an extraordinarily large and complex system. Indeed, *evolution-induced* incompatibility issues have been reported prevalent among general APIs [34, 42, 69]. As another example when it comes to user privacy, a recent study [50] reports that a hardware identifier ICCID (i.e., the serial number of the SIM card) remains accessible by apps in Android 10, despite Android's official documentation indicating that since that version, device IDs are regulated with new privileged permission of `READ_PRIVILEGED_PHONE_STATE` that is exclusively granted to privileged system apps [22]. Such non-compliance issues may expose Android to the risk of violating the *precise and transparent disclosure* requirement of GDPR (see "*right to be informed*" [64]), and could result in large penalties, e.g., "*a fine of up to €20 million, or 4% of the firm's worldwide annual revenue*" set by GDPR [68].

Our work. In this work, we propose DopCheck (Documented Privacy Changes Checker), which automatically identifies DPCs from Android documentation and explores *evolution-induced DPC non-compliance issues*, denoted simply as *DPC issues*. It extracts and analyzes privacy-related texts from the documents associated with each new release, and distills the DPC disclosures from them. Then, it generates test cases specifically tailored to each DPC description, which can comprehensively validate whether the desired changes have been properly implemented. Considering that most DPCs can be reflected in the APIs exposed to apps, DopCheck has to craft test cases as an app in Java with correct syntax and semantics. DopCheck runs the test cases on actual devices, assesses whether the claimed changes operate as expected, and reports an alert when any DPC issue has been observed. To this end, three primary challenges need to be addressed, as summarized below.

Challenge #1. Unstructured diverse textual data. The DPCs are disclosed as part of the Android documentation, which is written in unstructured natural language varying in length, structure, and writing quality across different Android versions.

Challenge #2. Validity of complex test cases. The test cases generated by DopCheck are encapsulated as third-party apps that must be runnable on real devices. Ensuring the validity of these apps is crucial, as any malformed app would fail trivially. A key challenge arising is the invocations of target APIs, which require correct API calls (e.g., the invocation sequence and non-null arguments) and a valid Android context (e.g., intents and broadcast receivers).

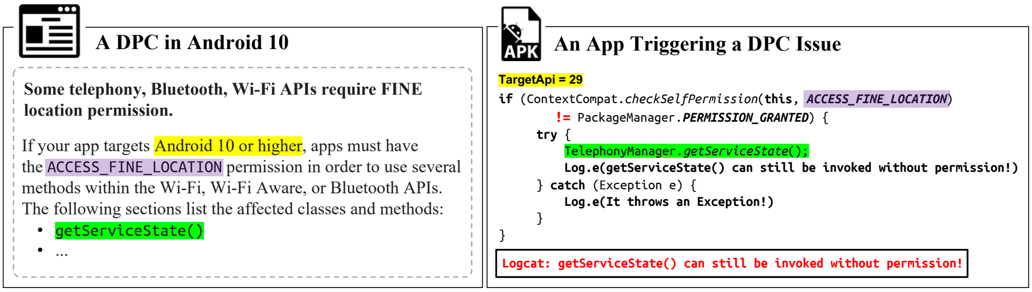
Challenge #3. Various types of DPCs. DPCs involve a wide spectrum of changes, including not only API changes that rarely request user interaction but also UI-related changes that prompt warnings or refrain from displaying sensitive information. For example, Android 13 introduces sensitive content hiding, which prevents sensitive contents from appearing in the content preview [26]. This requires DopCheck to have the capability to handle various oracles during its testing.

To address ***Challenge #1***, we conduct a thorough empirical study on the composition of DPCs, and design an ontology composed of nine entity types and five subsumptive relationships. The ontology serves as the backbone of DopCheck's DPC distillation, which extracts the information of each DPC using natural language processing (NLP) techniques, and represents it in a unified format to facilitate the test case generation. To address ***Challenge #2***, we resort to the latest large language models (LLMs) for test case generation, considering that LLMs are capable of assimilating and memorizing correct syntax and semantics of the code bases they have seen in their massive training data. We adopt a GPT-4 model [54], and conduct *in-context learning* to guide it with domain knowledge. This involves designing a series of prompts based on manually crafted sample test cases that have been validated with DPCs of Android 8 and 9. For ***Challenge #3***, we tailor DopCheck's testing into a category-wise manner, and leverage a neural network-based optical character recognition (OCR) technique [10] to handle the GUI-related DPCs.

We apply DopCheck with the latest versions (10 to 13) of the Android Open Source Project (AOSP). DopCheck recognizes 66 DPCs from their release documentation. Based on them, it generates 132 apps (in the form of *apk* files) as test cases for 79 privacy-sensitive APIs, 35 permissions, and 6 attributes. By executing the test cases on real devices, DopCheck manages to identify 19 DPC issues, among which, 12 are newly discovered in Android 13 and 6 in Android 10. According to the latest market share of the tested devices, these issues may pose more than 35% Android users the risk of unexpected privacy leakage.

Contributions. In summary, we make the following contributions in this paper.

- **Understanding DPCs in Android OS**. We conduct the first systematic study on evolution-induced DPCs in Android OS. We propose an ontology to facilitate the understanding of DPCs, and provide a characterization of DPCs to facilitate their testing.
- **An automatic approach to explore DPC non-compliance issues**. We design and implement DopCheck, an automatic solution that extracts DPCs from Android documentation, effectively generates valid test code, and tailors testing strategies for DPCs in different types to minimize human interaction during testing. Our evaluation shows that DopCheck is capable of identifying DPCs from Android documentation.



A DPC in Android 10

Some telephony, Bluetooth, Wi-Fi APIs require FINE location permission.

If your app targets Android 10 or higher, apps must have the `ACCESS_FINE_LOCATION` permission in order to use several methods within the Wi-Fi, Wi-Fi Aware, or Bluetooth APIs. The following sections list the affected classes and methods:

- `getServiceState()`
- ...

An App Triggering a DPC Issue

```
TargetApi = 29
if (ContextCompat.checkSelfPermission(this, ACCESS_FINE_LOCATION)
        != PackageManager.PERMISSION_GRANTED) {
    try {
        TelephonyManager.getServiceState();
        Log.e(getServiceState() can still be invoked without permission!)
    } catch (Exception e) {
        Log.e(It throws an Exception!)
    }
}
Logcat: getServiceState() can still be invoked without permission!
```

Fig. 1. An example of the DPC issue. Android 10's documentation states that the API `getServiceState()` requires a particular permission (left), whereas an app that does not request the permission can still invoke it without throwing any exception (right).

- **Revealing the status quo of the privacy mechanism evolution.** Our study presents the landscape of the privacy mechanism evolution in the latest releases of Android OS. We discuss the root causes of the DPC issues and share our insights on their prevention in future releases. Our work should raise an alert to Android to strengthen its testing mechanism prior to releasing new versions, and encourage the researchers in relevant fields to initiate the software quality assurance to prevent evolution-induced issues.

2 A MOTIVATING EXAMPLE AND APPROACH OVERVIEW

In this section, we use an example shown in Figure 1 to introduce the DPC issues in Android (Section 2.1), and then present an overview of our approach (Section 2.2).

2.1 DPC Issues

Android OS has undergone continuous changes in its privacy mechanisms during its evolution, aiming to strengthen user data protection and ensure compliance with regulations. These privacy changes are documented in natural language and released as a part of OS release documentation. One notable instance of such privacy enhancements can be found in Android 10 regarding Telephony, Bluetooth, and Wi-Fi APIs. Android 10 describes this change as part of its documentation, as shown in Figure 1 (left). It mandates that any app intending to use Wi-Fi, Wi-Fi Aware, or Bluetooth APIs must request the `ACCESS_FINE_LOCATION` permission. However, enforcing such changes in the actual OS implementations is challenging, given the complexity of Android OS. An inconsistency between *what is claimed in documentation* and *what the updated Android OS actually behaves* may exist. As shown in Figure 1 (right), in the absence of `ACCESS_FINE_LOCATION`, `getServiceState()` can still be invoked without throwing any exception, contradicting the documentation. Such *DPC issues* are the target problems that this work aims to explore.

2.2 Overview of DopCheck

DopCheck aims to 1) extract DPCs from official Android documentation into ontologies, 2) generate test cases as apps for each DPC, and 3) explore DPC issues by executing the generated test apps. Its approach consists of two main phases as outlined in Figure 2. It initiates by extracting DPC from Android documentation (Phase 1 in Figure 2). We define nine types of DPC entities that are essential for generating test cases, e.g., a test of invoking a privacy-sensitive API, and five subsumptive relationships to connect those entities, e.g., the appropriate permissions for an API, and the expected return value of an API. Based on this ontology, DopCheck extracts the information

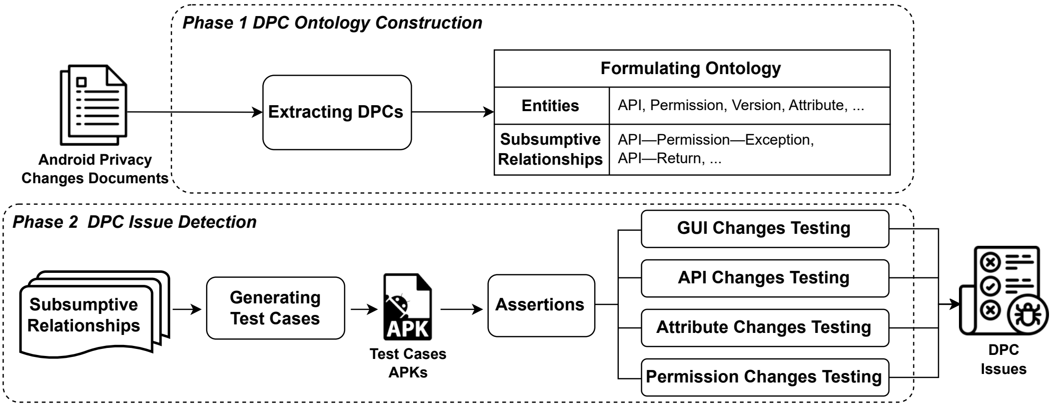

Phase 1 DPC Ontology Construction
Android Privacy Changes Documents
Extracting DPCs
Formulating Ontology
Entities
API, Permission, Version, Attribute, ...
Subsumptive Relationships
API—Permission—Exception, API—Return, ...
Phase 2 DPC Issue Detection
Subsumptive Relationships
Generating Test Cases
APK
Test Cases APKs
Assertions
GUI Changes Testing
API Changes Testing
Attribute Changes Testing
Permission Changes Testing
DPC Issues

Fig. 2. An overview of the workflow of DopCheck

of each DPC using NLP techniques, and represents it in a unified format to facilitate the test case generation (***Challenge #1***). We detail this process in Section 3.

With the extracted ontologies, DopCheck utilizes an LLM (i.e., GPT-4 [54]) to generate test cases (Phase 2 in Figure 2). This is to leverage the capability of generating syntactically and semantically correct code that has been well demonstrated in previous studies [66, 72, 76] (***Challenge #2***). To further enhance the quality of the generated test cases, DopCheck incorporates the concept of in-context learning, which uses domain knowledge to guide the LLM in crafting DPC-specific test cases. Regarding the test oracles, DopCheck devises specific assertions for each type of DPCs (***Challenge #3***). They serve the role of checking whether the test cases trigger expected behaviors or not. In those cases involving GUI changes, it uses machine learning-based Optical Character Recognition (OCR) techniques to recognize whether sensitive data appears on the screen during the execution of test cases. The entire phase of test case generation and execution for DPC issue detection is detailed in Section 4.

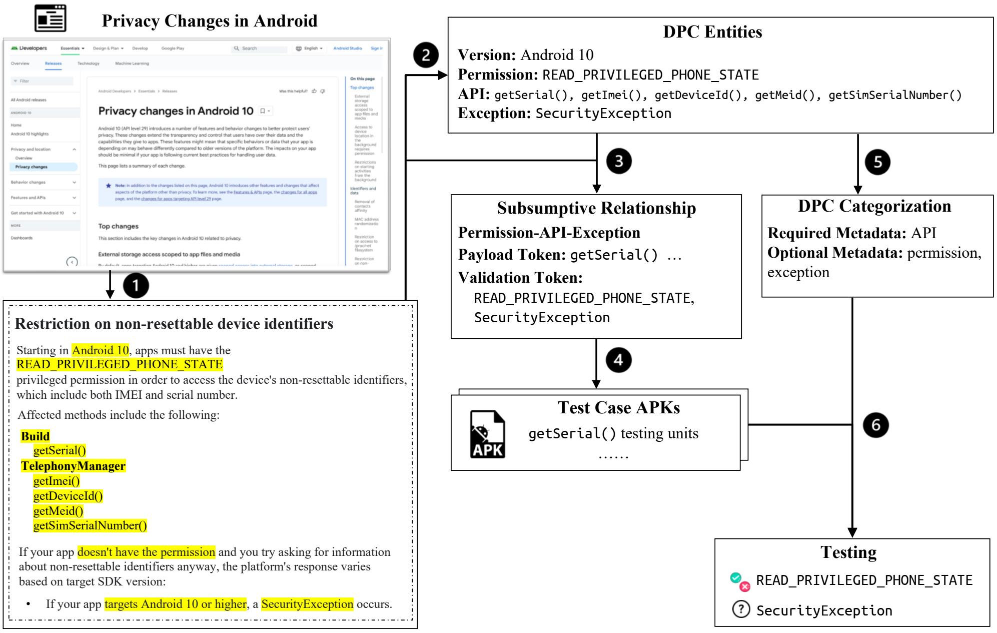
In Section 3 and Section 4, we use an end-to-end running example in Figure 3 to show the detailed steps in DopCheck.

3 DPC ONTOLOGY CONSTRUCTION

Given Android documents written in natural language, DopCheck identifies DPCs (Section 3.1), and recognizes the entities and subsumptive relationships to formulate DPC ontologies (Section 3.2) that is essential for test case generation.

3.1 Extracting DPCs

3.1.1 Sources. DopCheck takes as input the Android documentation accompanying each new OS version release. These documents are notably comprehensive and span more than 40 chapters distributed across 12 webpages. They state the changes made in various domains and topics, as exemplified in Figure 3 (left). To pinpoint privacy-related texts, DopCheck searches through the entire corpus of documents, and includes the chapters whose titles or contents contain specific keywords that we summarize from related literature [3, 16, 50] such as "*privacy*", "*security*", "*privacy/security changes*", "*personal*", "*identifier/identifiable/ID*", "*data protection*", "*permission management*" and "*user tracking*".



Restriction on non-resettable device identifiers

Starting in Android 10, apps must have the READ_PRIVILEGED_PHONE_STATE privileged permission in order to access the device's non-resettable identifiers, which include both IMEI and serial number.

Affected methods include the following:

Build
getSerial()
TelephonyManager
getImei()
getDeviceId()
getMeid()
getSimSerialNumber()

If your app doesn't have the permission and you try asking for information about non-resettable identifiers anyway, the platform's response varies based on target SDK version:

- If your app targets Android 10 or higher, a SecurityException occurs.

DPC Entities
Version: Android 10
Permission: `READ_PRIVILEGED_PHONE_STATE`
API: `getSerial(), getImei(), getDeviceId(), getMeid(), getSimSerialNumber()`
Exception: `SecurityException`

Subsumptive Relationship
Permission-API-Exception
Payload Token: `getSerial()` …
Validation Token:
`READ_PRIVILEGED_PHONE_STATE,`
`SecurityException`

DPC Categorization
Required Metadata: API
Optional Metadata: permission, exception

Test Case APKs
`getSerial()` testing units
……

Testing
`READ_PRIVILEGED_PHONE_STATE`
? `SecurityException`

Fig. 3. An end-to-end running example of a DPC on non-resettable device identifiers (Android 10) to show the steps of DopCheck

We conduct a manual confirmation on the collection process using Android 10 documentation, and affirm that our keywords effectively encompass all chapters related to privacy changes. We find that in Android 12 and 13, Android merges security and privacy changes lists, such that security-related alterations such as vulnerability fixes and security patches are included in the extracted chapters. Since they are in the general realm of security, we consider them as out of the scope of our work. We thus exclude those chapters with terms like "*security improvements*", "*vulnerability fixes*", and "*system maintenance*".

Scope. Our document collection covers Android 10-13, the most recent four releases as of mid-2023, considering their prevalence and our device availability. They can be considered representative, as every single one of them reserves more than 10% of the market share. According to Google's statistics of the distribution of active Android devices by OS versions [6], Android 11 (released in 2020) holds the largest market share (23.1%) as of August 2023. Android 13 (launched in 2022) accounts for approximately 21%, followed by Android 12 (17.7%) and Android 10 (15.5%), which were released in 2021 and 2019, respectively.

3.1.2 Extraction. Despite discrepancy across different versions, the extracted privacy change documents mostly follow the format of a *subheading* followed by the specific text describing the privacy changes. For example, in Figure 3 (left), the "*Restriction on non-resettable device identifiers*" is a subheading of a privacy change, and the contents below provide details about a specific change. With this observation, we utilize this format to extract DPCs. DopCheck first uses Python Beautiful Soup library [1] to parse the HTML files that contain the documents. It identifies the subheadings by locating the *H3* tag with the class *devsite-heading* and *aria-level* attribute set to 3. From this subheading, it continues parsing the HTML file until it encounters the next *H* tag, and the content within that tag is annotated as the specific text for the DPC (step ❶ in Figure 3).

Table 1. Nine types of DPC Entities

Entity Category	**Entity**	**Pattern/Archors**	**Regex**	**Semantics**
Aggregate	version	[Tag:*p*] [Start: "Android", End: 9-13]	-	Lower/Higher
	application	[Tag:*p*] ["app" + verb]	-	Hyperlink
Subject	API	[Tag:*code*] [End: ()]	\b[a-zA-Z_0-9]+\(\)	Hyperlink, Class, S/I,[1] expected return value
Property	permission	[Tag:*code*] [All capital letters joined with _]	([A-Z]+_)+[A-Z]+	Hyperlink, P/N[2]
	attribute	[Tag:*code*] [<>]	-	Hyperlink, P/N
Result	exception	[Tag:*code*] ["throw", "occur", End: "Exception"]	-	Condition[3]
	return	[Tag:*p*] ["return", End: "Noun/Num Entity"[4]]	-	API
	figure	[Tag:*figure*]	-	-
	effect	[Tag:*p*][Infinitive clause,[5] sentence contains API]	-	API

[1] S: The API is accessed through a static method of the class. I: The API is accessed through an instance method of the class.

[2] P: The DPC describes positive of (needs) this permission/attribute. N: The DPC describes the negative of (unnecessary) this permission/attribute.

[3] Condition: condition under which an exception occurs.

[4] Noun/Num Entity: Pair of nouns or number entity such as "empty list", "-1".

[5] Infinitive clause: An infinitive clause is an independent sentence, typically started by "to", followed by a verb or verb phrase.

3.2 Formulating Ontology

After extracting DPCs from Android documents, the next step is to construct DPC ontologies. Named-entity recognition (NER) is a common technique to recognize entities from a sentence and has been widely adopted in relevant research, such as PolicyLint [5], and Pico [70]. However, NER relies on labeled data for training, making it not directly applicable in the context of DPCs due to the limited size of the available corpus. Besides that, establishing the subsumptive relationships among entities is necessary to formulate ontologies of DPC, but this is out of the capacity of NER models. Alternatively, a recent study [4] resorts to the state-of-the-art LLMs for interpreting texts in the Android domain. Nevertheless, the power of LLMs lies in capturing syntactic and semantic information, rather than identifying new entities that never appear in its training datasets. For example, our pilot study (detailed soon in Section 3.2.1) shows that GPT-4 [54], one of the latest and most powerful LLMs, fails to recognize permissions introduced after the GPT-4 model has been trained, e.g., `POST_NOTIFICATIONS` in Android 13. Some system broadcasts named in all capital format, such as `BOOT_COMPLETED` and `LOCKED_BOOT_COMPLETED`, are mistakenly identified as permissions. For those reasons, we develop a domain-specific approach that constructs the DPC's ontologies by identifying entities and establishing subsumptive relationships.

3.2.1 Entities. Considering that DopCheck's test cases mostly center around APIs, we formulate the name of an API as the *subject entity*, and define another eight DPC entities to construct the context of an API invocation. This leads to overall nine entities of four categories listed in Table 1. The information associated with these entities, shown in the last column, plays key roles in the test case generation. For instance, following the API hyperlinks, DopCheck navigates to the documentation of the API, where details to correctly invoke the API are contained, including its class, arguments, methods, and version-specific information.

DopCheck uses a pattern-based manner to recognize entities, as listed in columns 3 and 4 in Table 1. It starts with identifying the entities of the *API* and *permission* types, given that these two types of entities have formats that can be easily recognized with regular expression. From the texts in the same paragraph, it uses anchors to find the sentences that contain the other seven entities. The entities of *version*, *attribute* and *figure* can be recognized using anchors or tags in

Table 2. The performance of DopCheck and LLM in extracting entities from DPCs

	Aggregate			**Subject**			**Property**			**Result**			**Overall**	
Groundtruth	17			38			19			11			85	
	TP	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*	*Precision*	*Recall*
LLM	15	1	2	36	3	2	17	4	2	9	1	2	89.53%	90.59%
DopCheck	16	0	1	37	0	1	19	1	0	10	1	1	97.62%	96.47%

† *TP*: True Positive, *FP*: False Positive, *FN*: False Negative.

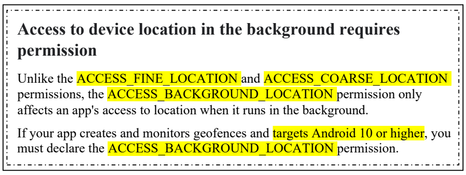
HTML. By locating the API and identifying infinitive clauses in its sentences, we determine the *effect* entity. DopCheck leverages Spacy [59], an open-source NLP library for tokenization and entity recognition, to identify and extract *application*, *exception* and *return* entities by detecting nouns or noun pairs in proximity to the keywords "*app*", "*application*", "*throw*", "*occur*", and "*return*". Step ❷ in Figure 3 presents the entities extracted, including *version*, *API*, *permission*, and *exception*.

The entities are organized into four categories, as shown in the first column in Table 1, to reflect the rationale for defining these entities. Besides the subject entity, the *version* and *application* entities are considered as *aggregate entities* as each DPC declares the target version and constraints of the entire app. The *permission* and *attribute* entities are categorized as *property entities* because they describe the setting to invoke an API. The *exception*, *return*, *effect*, and *figure* are treated as *result entities* as they indicate the response of the API calls.

We randomly select ten DPCs and manually labeled all entities within them as the ground truth. We then test DopCheck on the ground truth to benchmark its entity identification, and it achieves a precision of 97.62%. As a comparison, we invoke the LLM to complete the same task, by feeding each DPC text into it and requesting it to identify the entities within the text. The results of entity extraction by LLMs are listed in Table 2, and we also release the interaction with LLMs for this tasks in DopCheck repository [2].

3.2.2 *Subsumptive Relationships.*

After DopCheck has recognized entities, its next objective is to discover subsumptive relationships that establish connections among entities, and thereby facilitate the test case generation process. For example, an *API–permission* relationship depicts the proper permissions needed to invoke an API in the test app. Besides that, we have also defined three subsumptive relationships that are based on the connection between the subject entity *API* and other entity categories, i.e., *API–permission–exception*, *API–effect* and *API–return*, which are tailored to the requirement of testing API calls. In addition, we find some DPCs concern permission requests for certain app functionalities rather than specific APIs (see Figure 4), and therefore we define another relationship named *application–permission* All these five relationships are listed in Table 3.



Access to device location in the background requires permission

Unlike the ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION permissions, the ACCESS_BACKGROUND_LOCATION permission only affects an app's access to location when it runs in the background.

If your app creates and monitors geofences and targets Android 10 or higher, you must declare the ACCESS_BACKGROUND_LOCATION permission.

Fig. 4. An example indicating that multiple adjacent permission entities do not formulate a valid relationship

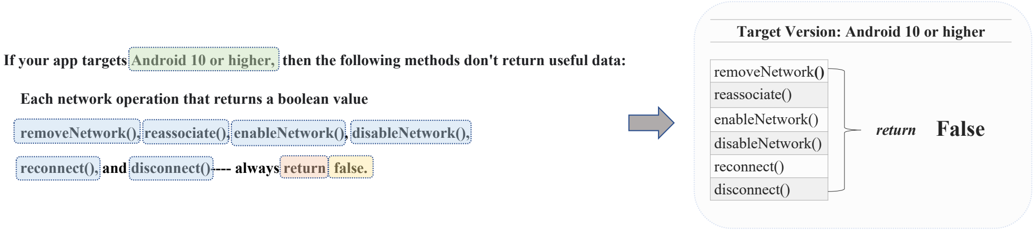
Aggregate entities are not explicitly included in any relationship, as they are implicit in all relationships. Each relationship can assume either a conditional or unconditional nature (column 2). For example, "*if your app targets Android 10 or higher, the [API] needs [permission]*" presents a conditional *API–permission* relationship. Following this, we depict the Hearst Patterns [35] of all relationships in Table 3.

To infer these relationships, a naive way is to group the entities that appear within the same sentence or paragraph. However, the adjacency of entities in the same or the neighboring sentences

Table 3. A list of subsumptive relationships

Subsumptive Relationships	**Condition Scenario**	**Pattern**	**Payload Tokens**[1]	**Validation Tokens**[2]	**Example**
Application -Permission	Conditional	<Temporal Conjunction ⌒ Conditional Clause, *must* ⌒ Permission>	Conditional Clause	Permission	If your app creates and monitors geofences and targets Android 10 (API level 29) or higher, you must declare the `ACCESS_BACKGROUND_LOCATION` permission.
	Unconditional	-	-	-	-
API-Permission	Conditional	<Temporal Conjunction ⌒ Permission ⌒API>	API Conditional Clause	Permission	If your app targets Android 13 or higher, you must declare the `NEARBY_WIFI_DEVICES` permission to call `publish()`, `attach()`, ...
	Unconditional	<*To* ⌒ API, ⌒ Permission>	API	Permission	To read number from `onCallStateChanged()`, you need the `READ_CALL_LOG` permission only.
API-Permission -Exception	Conditional	<Temporal Conjunction ⌒ Permission ⌒ API ⌒ Exception>	API Permission	Exception	Refer to the example in Figure 3.
	Unconditional	-	-	-	-
API-Effect	Conditional	<Temporal Conjunction ⌒ API ⌒ Conditional Clause, ⌒ Effect>	API Conditional Clause	Effect	When an app calls `getPrimaryClip()` to access clip data from a different app for the first time, a toast message notifies the user of this clipboard access.
	Unconditional	<*To* ⌒ Effect ⌒ API >	API	Effect	To determine the permission group into which the system has placed a platform-defined permission, call `getGroupOfPlatformPermission()`.
API-Return	Conditional	<Temporal Conjunction ⌒ Conditional Clause,⌒ API ⌒ *return* ⌒ Value>	API Conditional Clause	Value	If your app targets Android 9 (API level 28) or lower, `getSerial()` returns null.
	Unconditional	<API ⌒ *return* ⌒ Value>	API	Value	Apps targeting Android 10 or higher cannot enable or disable Wi-Fi. The `setWifiEnabled()` method always returns false.

[1] Payload tokens are the sub-sentence that is used to construct the test app.
[2] Validation tokens are the sub-sentence that is used to construct assertions.



If your app targets Android 10 or higher, then the following methods don't return useful data:

Each network operation that returns a boolean value

removeNetwork(), reassociate(), enableNetwork(), disableNetwork(), reconnect(), and disconnect()---- always return false.

Fig. 5. An example of the *API–return* subsumptive relationship

does not necessarily imply a meaningful relationship among them. Taking the DPC *Access to device location in the background requires permission* in Figure 4 as an example, the two entities `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION` are intended merely for comparison purpose, and `ACCESS_BACKGROUND_LOCATION` is the sole permission required for an app to create and monitor geofences. Therefore, DopCheck uses a semantics-aware method to construct relationships (step ❸ in Figure 3).

The key components in subsumptive relationships are the *payload tokens* and *validation tokens*, which are informative for the test case generation. Both of them are used as the key information in our test generation (to discuss in Section 4.1). The former specifies the app settings and values of the entities, and the latter defines the assertions to indicate whether a test behaves as expected or not (i.e., the testing oracle). DopCheck uses the `en_core_web_sm`, an NLP model of Spacy [59] to identify these two components. For the conditional patterns, Spacy identifies those conditional and temporal conjunctions, e.g., *if* and *when*, and marks their *dep* attributes (i.e., the syntactic dependency). Using them as anchors, DopCheck can obtain the two components from the sub-sentences. Figure 5 presents an example of the *API–return* relationship, where payload tokens, APIs and expected return values are included.

During the above recognition procedure, DopCheck may encounter complex sentences due to the diverse formatting and descriptions within the documents. To facilitate Spacy's identification, DopCheck processes two specific cases as follows.

Multiple Condition Statement. The documentation often includes nested conditional relationships such as *if A: {if B, then C; if D, then E}*. Figure 3 shows an example of this format ("*If your app does not... If your app targets...*"). DopCheck merges these conditions to facilitate Spacy's recognition, combining *A* and *B* into *A & B*, *A* and *D* into *A & D*, i.e., *if A & B, then C; if A & D, then E*.

Special Pronouns. Some nouns may be used to refer back to previously mentioned permissions or APIs. For example, the sentence "*If your app does not have the permission and ...*" in Figure 3, the pronoun "*permission*" refers to `READ_PRIVILEGED_PHONE_STATE` mentioned earlier. We use Spacy to identify pronouns for permissions and APIs in DPC text and then replace them.

4 DPC ISSUE DETECTION

With the extracted ontologies, DopCheck generates test cases to test the OS implementations for DPC issues. This involves generating valid API invocations (Section 4.1) and test assertions (Section 4.2).

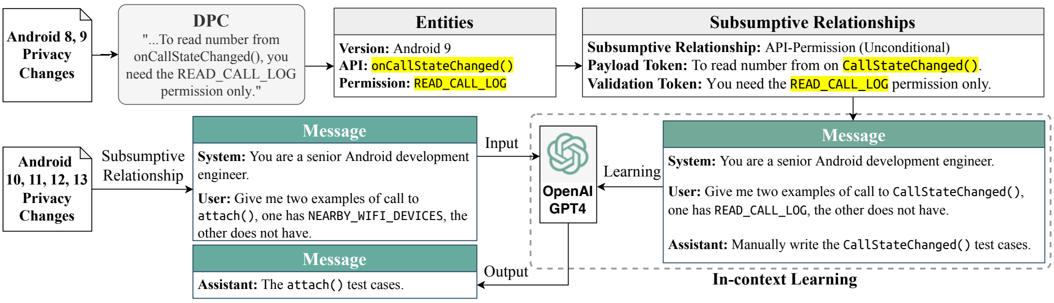
4.1 Generating Test Cases

DopCheck initiates its test case generation by creating an empty app, which includes the manifest file, an empty Activity, a basic GUI page with a button linked to an empty handler function, and the `onCreate()` method. The test code generated is then inserted into this empty app. For example, changes regarding the API are placed inside the `onCreate()` method, changes requiring user interaction are incorporated into the button's `onClick` handler, and the permission requests are added to the manifest file. The main challenge it must address is to generate valid code for invoking APIs. Android API invocations are characterized by parameter variations, requirements, and usage patterns, such as the need for single or multiple instances and the requirement for specific callback functions to be passed as arguments. It is impractical to create invocations for each DPC manually. For that reason, we leverage LLMs to overcome the practicability challenge and ensure the generated test cases are valid and contextually appropriate. Specifically, we adopt GPT-4 [54], a state-of-the-art language model built on the transformer architecture. GPT-4 has extensive exposure to diverse training data, enabling it to generate syntactically accurate code. Furthermore, DopCheck harnesses the concept of *in-context learning* [55], which allows us to guide the model in producing results that align with our specific requirements in a few-shot. In this form of learning, three key inputs are considered.

- **System**: The system input sets the model's behavior, role, and context, giving a general direction, tone, and style to the conversation. It assists the model in better understanding the user's intent, guiding the conversation, or providing contextual information.

Table 4. Examples of Constructing LLM prompts with payload tokens and validation tokens

Payload Token	LLM User Prompt
Conditional Clause	Write an easy Android app targeting [*version*], requiring [*Conditional Clause*].
	Give me an easy Android app example of [*Conditional Clause*], which targeting [*version*]
	Provide a simple Android app illustration for [*Conditional Clause*], designed for [*version*].
	Present a beginner-friendly Android app instance demonstrates [*Conditional Clause*] and targeting [*version*].
API	Write an example of invoking [*API*], targeting [*version*].
	Provide an instance of invoking [*API*] for [*version*].
	Demonstrate how to use [*API*] for [*version*].
	Show how to make a call to [*API*] with a target of [*version*].
API, Permission	Write two examples of calls to [*API*], one with [*Permission*] and the other without.
	Create two scenario of calls to [*API*], one with [*Permission*] and the other without.
DPC Context	Provide an example of [*DPC Context*].



"...To read number from onCallStateChanged(), you need the READ_CALL_LOG permission only."

Version: Android 9
API: `onCallStateChanged()`
Permission: `READ_CALL_LOG`

Subsumptive Relationship: API-Permission (Unconditional)
Payload Token: To read number from on `CallStateChanged()`.
Validation Token: You need the `READ_CALL_LOG` permission only.

System: You are a senior Android development engineer.
User: Give me two examples of call to `CallStateChanged()`, one has `READ_CALL_LOG`, the other does not have.
Assistant: Manually write the `CallStateChanged()` test cases.

System: You are a senior Android development engineer.
User: Give me two examples of call to `attach()`, one has `NEARBY_WIFI_DEVICES`, the other does not have.

Assistant: The `attach()` test cases.

Fig. 6. An example of test case generation with in-context learning

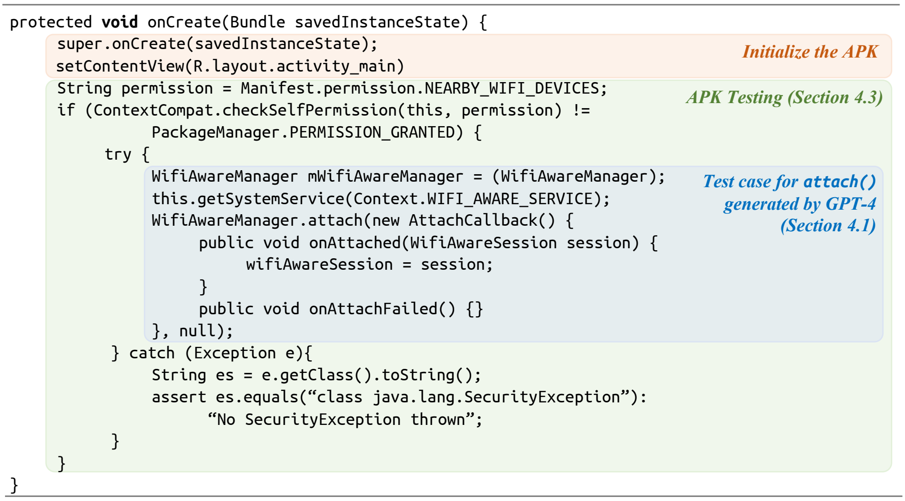
- **User**: The user's role is to pose questions, request information, provide input, or instruct the model to perform specific tasks. The user's input can be in the form of questions, commands, statements, or any information that requires a response from the model.
- **Assistant**: The assistant can see expectations for the model's responses. By configuring the assistant, we convey to the model the desired outcome we are aiming for, allowing us to generate test cases that align with our expectations.

DopCheck first sets the system as "*You are a senior Android development engineer*", to guide GPT-4 to respond with Android domain-specific answers. A recent study [67] demonstrates that the learning ability of LLMs improves with the increase in the number of in-context learning examples. We thus use the maximal in-context learning samples allowed within the token limit of GPT-4 (approximately 5 DPCs). In particular, DopCheck uses the subsumptive relationships of five DPCs from Android 8 and Android 9 and passes their payload tokens as user inputs for GPT-4, as shown in Table 4. We manually write corresponding test cases as assistant inputs to enhance its capability for the in-context training. Figure 6 illustrates an example of our in-context learning process with GPT-4 (step ❹ in Figure 3). After the in-context training, prompts constructed from the ontologies (shown in Table 4) are fed into GPT-4. To demonstrate the quality of the generated test cases, Figure 7 presents an example which assesses whether a `SecurityException` is thrown when the *attach()* API does not have the `NEARBY_WIFI_DEVICE` permission.

Table 5. Four categories of DPCs to facilitate assertion construction

DPC Category	**Entity Criteria**		**Testing Mode**
	Required Entities[1]	**Optional Entities**[2]	
API changes	*API*	*permission, attribute, exception, return, effect*	Return value validation
Permission changes	*permission*	*API, attribute, exception, return*	GUI/Prompt validation
GUI changes	*figure*	*API, exception, return*	GUI validation
Attribute changes	*attribute*	*API permission, , exception, return*	Property testing

[1] Required Entities: Entities that must be included to be classified into this category.
[2] Optional Entities: Entities that may be included to be classified into this category.



```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main)
    String permission = Manifest.permission.NEARBY_WIFI_DEVICES;
    if (ContextCompat.checkSelfPermission(this, permission) !=
              PackageManager.PERMISSION_GRANTED) {
        try {
              WifiAwareManager mWifiAwareManager = (WifiAwareManager);
              this.getSystemService(Context.WIFI_AWARE_SERVICE);
              WifiAwareManager.attach(new AttachCallback() {
                  public void onAttached(WifiAwareSession session) {
                      wifiAwareSession = session;
                  }
                  public void onAttachFailed() {}
              }, null);
        } catch (Exception e){
              String es = e.getClass().toString();
              assert es.equals(“class java.lang.SecurityException”):
                   “No SecurityException thrown”;
        }
    }
}
```

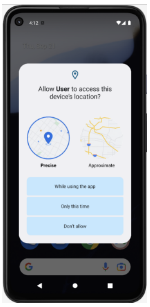
Fig. 7. A test case of `attach()` generated by GPT-4

4.2 Assertions

During the execution of the test cases, DopCheck has to check whether the invocation of the APIs leads to expected behaviors or not (step ❻ in Figure 3), for example, to check whether the return value matches the descriptions in the document, or to check whether an expected security exception is thrown. To define assertions for this purpose, we take a category-wise strategy (step ❺ in Figure 3). Table 5 defines four categories of DPCs and our categorization criteria. Below we brief the designed assertions for each category.

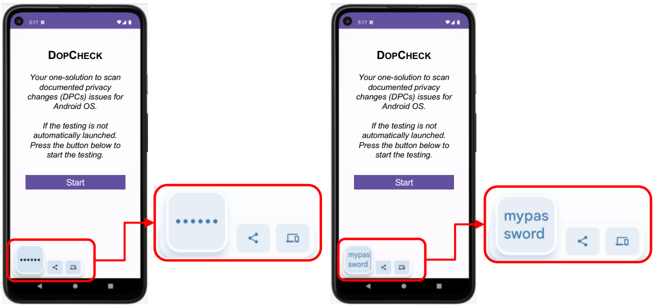
API changes. For the category of API changes, DopCheck compares the actual return values obtained from the test case with the validation token in their subsumptive relationship. The test passes if they are found matched. Otherwise, a DPC issue is reported.

Permission changes. For the category of permission changes, DopCheck applies two testing methods. First, after the test case dynamically requests the permission, DopCheck conducts a checking whether the GUI prompts the users for authorization, by *ActivityManager*. Figure 8 shows an example when testing two permissions `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION` on Android 12. The other method is designed specifically for the Permission DPCs that have an API entity. DopCheck tests whether the app can function properly by altering the permissions in the manifest file. For instance, if the subsumptive relationship is in the form of Permission-API-Exception (see Table 3), we check whether the system throws an exception when the required permission is missing in the manifest file.



Allow **User** to access this device's location?

Fig. 8. Dynamically requesting a dangerous permission



DopCheck

Your one-solution to scan documented privacy changes (DPCs) issues for Android OS.

If the testing is not automatically launched. Press the button below to start the testing.

DopCheck

Your one-solution to scan documented privacy changes (DPCs) issues for Android OS.

If the testing is not automatically launched. Press the button below to start the testing.

Fig. 9. An example of the GUI DPC – "*Hide Sensitive Content From Clipboard in Android 13*". Clipboard contents labeled as sensitive (left) and non-sensitive (right) are shown, respectively.

GUI changes. For the category of GUI changes, DopCheck lets the test case display the value it obtains on the screen, takes screenshots, and utilizes PaddleOCR [10] to recognize whether sensitive data appears on the screenshots.

Attribute changes. For the category of attribute changes, DopCheck tests whether adding or removing attributes as indicated in the documentation causes any exceptions.

We remark that not all DPCs can be classified into the four aforementioned categories and we call those DPCs ***miscellaneous changes***. We list all the miscellaneous DPCs identified during our categorization in Section 5.1.

5 EVALUATION AND LANDSCAPE

We apply DopCheck on Android versions 10 to 13, to understand the landscape of DPC issues in the latest Android implementations. Two Android phones, i.e., Pixel 3a (compatible with Android 9 to Android 12) and Pixel 6a (compatible with Android 12 and 13), are used for our experiment. We focus on Android AOSP versions, as they are typically the first batch to receive the official updates and security patches, which is crucial for testing new features. As DopCheck is designed to identify DPCs and explore DPC issues from the Android implementation, our evaluation aims to answer the following three research questions (RQs).

RQ1. What is DopCheck's performance in identifying DPCs? Is it able to accurately extract ontologies from each DPC?

RQ2. How effective is DopCheck in generating valid test cases?

RQ3. Based on DopCheck's findings, which DPCs exhibit discrepancies in the actual Android implementations and potentially pose privacy and security concerns for both app developers and the Android team?

5.1 RQ1: Ontology Extraction and Real-World DPCs Categorization

DopCheck identifies 66 DPCs across the four Android versions, involving 79 APIs, 35 permissions, 6 attributes and 12 effect entities. To validate DopCheck's performance, we invite three members from our institute to conduct an independent manual assessment. All of them have experience in Android app analysis and development, and one of them also possesses a legal background and has experience of drafting the privacy policy document for an app. They all have no knowledge of DopCheck before the confirmation process. They are provided with the introduction of entities with examples to interpret the tasks. We then randomly select 11 documentation excerpts from Android 10-13, accounting for 10% of the documentation in each version, for them to manually identify DPCs and entities. After that, they have a discussion on the identified items to reach

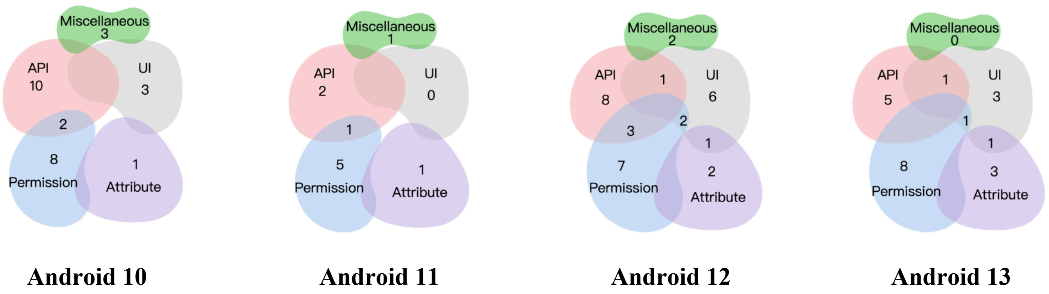


Fig. 10. Distribution of DPC categories for each Android version

an agreement. This process leads to 31 DPCs and 62 entities being recognized from the given documents. Among them, 31 DPCs and 59 entities are identified by DopCheck. The missing two entities include one API entity and one attribute entity, with the reason being that DopCheck failed to locate the anchor for the entity, e.g., the *MediaProjection* API does not end with "()" in a DPC.

We categorize the identified DPCs based on the criteria listed in Table 5. As shown in Figure 10, for each Android version, it is apparent that the most alterations occur within permissions (8, 5, 5, and 7 DPCs in Android version 10, 11, 12, and 13, respectively). This prominence is mainly due to the fact that permission control is a pivotal strategy for safeguarding user privacy. Furthermore, permissions standardize the procedure by which developers access designated user data, guaranteeing that applications can solely access sensitive information under reasonable and indispensable conditions [30]. We also observe that there are significant privacy changes for Android 10 and Android 12 (15 and 6 unique DPCs, respectively)[1]. The difference lies in Android 10 focusing on updates to APIs (9 DPCs) and permissions (8 DPCs), while Android 12 has predominantly emphasized privacy enhancements in the GUI changes (6 DPCs). In addition, we find three DPCs simultaneously involve both GUI and permissions in both Android versions 12 and 13. This reflects that in recent years, Android's changes to the GUI are not only stemming from the Human-Computer Interaction (HCI) enhancement, but also encompassing a focus on permission-related aspects [51]. For instance, in Android 12, a change allows apps to request only coarse-grained user location instead of precise latitude and longitude coordinates [25]. This shift has established a precedent where users can choose the precision level of the information they are comfortable sharing, consequently enhancing compliance with the principles of *transparency* and *user right* in regulations like GDPR.

Table 6 presents the DPCs that do not fit into any category, which we classify as *miscellaneous DPCs*. These DPCs are often related to users' behaviors, preferences, or time dimensions, so even if GPT-4 can generate valid test apps, it can still be challenging to conduct accurate testing. For instance, Android 10 removed the contact affinity information, which implies that the platform no longer sorts search results for contacts based on interaction frequency. As another example, if a user does not use the app for an extended period, Android 11 will reset all permissions for this app, and the user needs to reauthorize all permissions to the app.

5.2 RQ2: Test Case Generation

After excluding the 6 miscellaneous DPCs, DopCheck manages to generate 132 test cases for 54 out of the remaining 60 DPCs. The generated test cases consist of 79 API testing units, 35 permission testing units, 12 UI testing units, and 6 attribute testing units. Our test case generation results are summarized in Table 7.

[1] Recall that there is overlap among categories (see Table 5).

Table 6. List of six miscellaneous DPCs

Android Version	DPC Title	Reason
Android 10	Removal of contacts affinity	User preferences and behavior
	MAC address randomization	Enterprise use case
	Restrictions on starting activities from the background	System-level changes
Android 11	Permissions auto-reset	Time dimension
Android 12	Motion sensors are rate-limited	User preferences and behavior
	App hibernation	Time dimension

Table 7. Test case generation results

DPC Category	Subcategory (if any)	Generated Test Cases	Executable Valid Test Cases	Non-executable Test Cases	Off-target Test Cases	Success Rate	Time Cost (s)
API	Static Method	2	2	0	0	100.00%	26.91
	Abstract Method	6	6	0	0	100.00%	25.37
	Single Instance Method	35	30	4	1	85.71%	28.97
	Multi-Instance Method	36	27	5	4	75.00%	32.36
Permission	Dangerous Permission	27	23	3	1	85.19%	23.43
	Signature Permission	6	4	1	1	66.67%	22.38
	Normal Permission	2	2	0	0	100.00%	21.91
UI	–	12	10	2	0	83.33%	18.71
Attribute	–	6	3	2	1	50.00%	20.03
Total		**132**	**107**	**17**	**8**	**82.88%**	**24.45**

Efficacy of test case generation. To confirm the validity of the generated test cases, we manually inspect all 132 test cases, checking whether they correctly hit the intended constraints of entities and relationships. There are 8 test cases that invoke incorrect APIs (referred to as *off-target* cases), mainly due to Java method overloading, which allows multiple APIs of the same name to be defined in the same class. We then execute the remaining test cases, and find that 107 out of all 132 (82.88%) test cases are executable. The 17 non-executable test cases are due to erroneous API invocation generated by the LLM, such as missing parameters and erroneous object instantiation. In consideration of testing coverage, we manually fix all test cases to make them syntactically and semantically correct.

Time cost. On average, generating a test case takes around 25 seconds, as shown in Table 7. The loading and execution time for each case is within 10 seconds on the real device. Overall, the API category takes longer than others, mainly because DPCs in this category often involve more entities and complex subsumptive relationships. That leads to a higher demand with regard to prompt tokens, and as a result, the LLM may need a longer duration to produce a response.

Test cases in the API change category. Within the API change category, DopCheck can successfully invoke test units for all APIs that are defined as *static methods* and *abstract methods*. This is because these methods typically have relatively simple parameter structures, such as `getSerial()`, a static method from the `Build` class, which can be invoked without any parameters.

For APIs that are *single instance non-static methods*, DopCheck can generate valid invokable test units for 85.71% of them. These methods typically require constructing an instance, such as `TelephonyManager`, and then the API is invoked through the instance. The failures in generating test cases for APIs of this type (i.e., the `getAvailableNetworks()` in `TelephonyManager` class) are due to missing details in Android documentation. Although Android 10 privacy change document clearly states that `getAvailableNetworks()` belongs to the `TelephonyManager`, we could not find any explanation for the getAvailableNetworks() in the official documentation of the TelephonyManager [33], neither from the API library of Android 10.

For APIs that are *multi-instance non-static methods*, DopCheck only manages to generate valid test units for 75% of them. This is because the multi-instance methods require more than one instance to be invokable. For example, in the WifiManager class, an API named updateNetwork() takes a WifiConfiguration instance as a parameter. For that reason, DopCheck needs to construct both WifiManager and WifiConfiguration instances to call updateNetwork(). Nonetheless, our adopted LLM fails to identify semantic connections in generating complex multi-instance test units.

Test cases in the permission change category. The majority of permissions involved in DPCs within this category are at a dangerous level (77.14%). We observe that some of the failed test cases are due to specific permission combinations that require interactions to trigger or some new permissions that GPT-4 cannot correctly identify. Among the 29 executable test cases, DopCheck discovers 12 APIs that can be invoked without being granted the requested NEARBY_WIFI_DEVICE permission, and 4 APIs that can be invoked without requesting ACCESS_FINE_LOCATION. We discuss more details about our findings in Section 5.3.

Considering that permission group may contain issues of permission violations [26], we also screen each DPC to check whether they involve any group permissions. Two such permissions are found, namely "*Request location access at runtime*" in Android 12 [20] (involving ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION) and "*Granular media permissions*" in Android 13 [23] (involving READ_MEDIA_IMAGES and READ_MEDIA_VIDEO). We then update relevant test cases to assess these two groups. Taking the former as an example, we first grant the ACCESS_COARSE_LOCATION permission in the old version ($\mathcal{A}$) of the test app, and then grant ACCESS_FINE_LOCATION in the updated version ($\mathcal{B}$). The results show that in version $\mathcal{B}$, the system still prompts for fine location permission, indicating Android has fixed the recently reported permission issue [11].

Test cases in the GUI and attribute categories. DPCs within the GUI change category predominantly focus on the presentation of user-sensitive data. For instance, Android 13 allows apps to copy sensitive content to the clipboard and provides the option to add flags to prevent sensitive content from appearing in content previews [26]. Out of the 12 GUI-related DPC test cases, 10 of them are executable (83.33%), causing GUI-related DPCs do not involve complex API invocations or permission requests.

For the 6 attribute DPCs, the LLM can only generate 3 test cases that compile successfully due to the personalization of attribute values. Take one DPC in Android 12 titled "*Known signers permission protection flag*" as an example, the knownCerts attribute mentioned in that DPC requests actual digital certificates for testing purposes. However, the LLM adopted by DopCheck cannot generate a certificate on its own. As a result, DopCheck fails to generate a valid test case for this DPC. We discuss the limitation of this aspect in Section 6.2.

5.3 RQ3: DPC Issues Landscape

Through executing the test apps, DopCheck manages to find 19 independent DPC issues from three DPCs, among which two DPCs are on Android 10 and one on Android 13. Besides that, there are a total of 18 APIs and 3 permissions involved in our findings. Among the 19 DPC issues, one issue replicates a known vulnerability of Android 10, and the remaining 18 issues are discovered by us for the first time. We present more details of our findings in Table 8.

First, DopCheck finds that the API getIccid() still returns the actual ICCID in Android 10, which contradicts Android's privacy policy stipulating that the third-party apps' access to the ICCID is prohibited [18]. That issue was first unveiled in a recent study [50].

The second DPC in our findings is titled "*Some telephony, Bluetooth, Wi-Fi APIs require FINE location permission*". It stipulates that the invocation of certain APIs requests the ACCESS_FINE_LOCATION

Table 8. Inconsistency between DPCs and actual Android OS behavior

DPC Title	API	Permission	Version	Status
Restriction on non-resettable device identifiers	`geticcid()`	`READ_PRIVILEGED_PHONE_STATE`	10	Replicate
Some telephony, Bluetooth, Wi-Fi APIs require FINE location permission	`getAvailableNetworks()`	-	10	New
	`startScan()`[W]	`ACCESS_FINE_LOCATION`	10	New
	`startScan()`[B]	`ACCESS_FINE_LOCATION`	10	New
	`startDiscovery()`	`ACCESS_FINE_LOCATION`	10	New
	`startLeScan()`	`ACCESS_FINE_LOCATION`	10	New
	`getServiceState()`	`ACCESS_FINE_LOCATION`	10	New
New runtime permission for nearby Wi-Fi devices	`attach()`	`NEARBY_WIFI_DEVICES`	13	New
	`publish()`	`NEARBY_WIFI_DEVICES`	13	New
	`subscribe()`	`NEARBY_WIFI_DEVICES`	13	New
	`addLocalService()`	`NEARBY_WIFI_DEVICES`	13	New
	`connect()`	`NEARBY_WIFI_DEVICES`	13	New
	`createGroup()`	`NEARBY_WIFI_DEVICES`	13	New
	`discoverPeers()`	`NEARBY_WIFI_DEVICES`	13	New
	`requestDeviceInfo()`	`NEARBY_WIFI_DEVICES`	13	New
	`requestGroupInfo()`	`NEARBY_WIFI_DEVICES`	13	New
	`discoverServices()`	`NEARBY_WIFI_DEVICES`	13	New
	`requestPeers()`	`NEARBY_WIFI_DEVICES`	13	New
	`startScan()`[W]	`ACCESS_FINE_LOCATION`	13	New

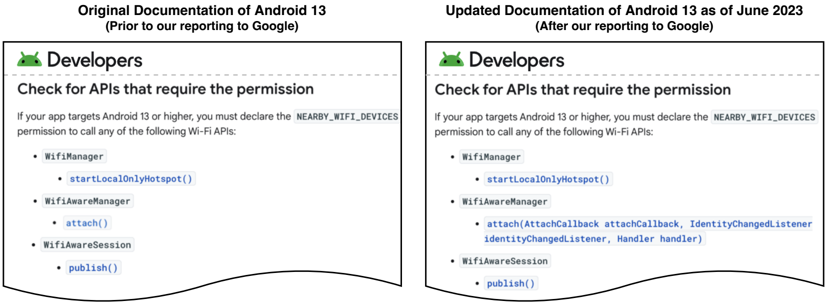
[W]: The API is defined in class `WifiManager`.

[B]: The API is defined in class `BluetoothLeScanner`.

permission since Android 10. However, our testing discovered that one API listed in that DPC is not included in Android API for third-party app developers, and another five APIs can still be invoked without the aforementioned permission. We later manually confirmed these six DPC issues after checking the AOSP's source code. Notably, one of the involved APIs, i.e., `startScan()` of the class `WifiManager`, was again mentioned to request the `ACCESS_FINE_LOCATION` permission three years later in another DPC in Android 13. Unfortunately, this DPC issue still remains in the Android 13 release, making it a problem open for more than four years until detected by our work.

The last DPC issue is about the `NEARBY_WIFI_DEVICES` permission, which is primarily responsible for connecting to nearby devices via Wi-Fi. DopCheck finds a DPC titled "*New runtime permission for nearby Wi-Fi devices*" mandates that an app must declare the use of the `NEARBY_WIFI_DEVICES` for accessing 13 relevant APIs if its developer sets the build target to Android 13 or higher. However, among the 13 APIs, 11 APIs (see in Table 8) can still be successfully invoked without being granted the aforementioned permission. Our manual inspection of the AOSP source code has confirmed our findings. At the same time, from the AOSP implementation, we find that the source code level documentation of the 13 problematic APIs (i.e., the Javadoc located before the actual API source code) has clearly stated that the `NEARBY_WIFI_DEVICES` permission is required, but unfortunately the immediately-following definitions of these APIs lack the annotations requesting the `NEARBY_WIFI_DEVICES` permission.

Responsible disclosure and ethical consideration. We have reported all our newly discovered DPC issues to Google via the Issue Tracker Platform, and actively assist in the fix process. Fixing DPC issues turns out to be an unexpectedly non-trivial process. Take the API named `attach`, i.e., the third DPC in Table 8, as an example. In fact, there are two APIs named `attach` in the class `WifiAwareManager`. According to the documentation of this class, both APIs can be used to initialize the Wi-Fi Aware service. DopCheck manages to find that one of the APIs, namely



Original Documentation of Android 13
(Prior to our reporting to Google)

Developers

Check for APIs that require the permission

If your app targets Android 13 or higher, you must declare the `NEARBY_WIFI_DEVICES` permission to call any of the following Wi-Fi APIs:

- `WifiManager`
    - `startLocalOnlyHotspot()`
- `WifiAwareManager`
    - `attach()`
- `WifiAwareSession`
    - `publish()`

Updated Documentation of Android 13 as of June 2023
(After our reporting to Google)

Developers

Check for APIs that require the permission

If your app targets Android 13 or higher, you must declare the `NEARBY_WIFI_DEVICES` permission to call any of the following Wi-Fi APIs:

- `WifiManager`
    - `startLocalOnlyHotspot()`
- `WifiAwareManager`
    - `attach(AttachCallback attachCallback, IdentityChangedListener identityChangedListener, Handler handler)`
- `WifiAwareSession`
    - `publish()`

Fig. 11. Change in Android documentation before and after our reporting

`attach(AttachCallback, Handler)`, does not have the permission `NEARBY_WIFI_DEVICES` included in its `RequiresPermission` annotation. Therefore, it can be invoked even if the test apps are not granted that permission. In contrast, the annotation for the other API, i.e., `attach(AttachCallback, IdentityChangedListener, Handler)` is correct in AOSP. In other words, an app must have the `NEARBY_WIFI_DEVICES` permission to invoke the API with three parameters.

An ideal fixation is to add the missed permission to the two-parameter version of the `attach` API, making the two APIs being controlled consistently, since this is a case of Java method overloading. However, the Android team updated the privacy change document by replacing the general name of the involved API, i.e., `attach()`, with the specific three-parameters one, namely `attach(AttachCallback, IdentityChangedListener, Handler)`, without changing the implementation. This change is illustrated in Figure 11. We speculate that the code remains unchanged because of compatibility requirement, but remark that this fixation may not be a secure one, as it just hides the buggy API from the developer documentation. We are still in discussion with the Android team through the issue tracker system.

6 DISCUSSIONS

In this section, we first propose our recommendations to different parties in the Android ecosystem who may be involved by DPC issues (Section 6.1). We also discuss the limitations of our work. In the end, we outlook the future landscape of studying evolution-induced DPC issues in Android OS.

6.1 Recommendations

Android. The DPCs that Android has clear entities specified in its documentation could have been validated through a rigorous assurance mechanism. For such DPCs, Android should establish a regression testing process to check the consistency between documentation and code, ensuring all the implementations comply with the high-level requirements and policies stated in the documentation. Android should pay special attention to the interfaces that lead to the same data or service as the documented APIs, and ensure that the access points to the same data or service (see the `attach` example in Section 5.3) should be enforced with consistent permissions. Regarding the fixation of DPC issues, their root causes should be pinpointed and addressed, rather than opting for merely modifying the documents with undesired compliance.

App developers. Developers should strictly adhere to the guidelines and requirements outlined in the Android documentation while developing their applications. If any inconsistencies between the documentation and the actual code behavior are discovered, they should promptly report it to Android to ensure the legality and compliance of their applications. Additionally, developers should stay vigilant about the release of each new Android version, paying close attention to the usage and permission requirements of updated APIs. This would ensure that their applications remain compatible with the regulations and requirements of multiple Android versions.

6.2 Limitations

To the best of our knowledge, DopCheck is the first work that conducts a systematic analysis on DPC issues. However, our work still carries several limitations that could be addressed in future work. First, due to our scope of privacy changes, the ontologies we have proposed (Table 1 and Table 3) still have some omissions, such as the presence of tabular information in a few DPCs. Our ontology can be further extended to capture rich relationships to comprehensively profile privacy changes, and ideally, general changes. Such an ontology can facilitate change assurance. Second, DopCheck has not considered some specific changes (Table 6), e.g., the "*app hibernation*" which revokes the app's granted permissions if it is not used for an extended period, and the "*motion sensors are rate-limited*" which involves extensive user interactions. Third, this work primarily investigates the consistency between updates to Android documentation and their implementation, and we thus only assess APIs and permissions explicitly detailed in the documentation. The invocations of interfaces or permissions made in undocumented ways, e.g., Java reflection, can be considered in future work. Finally, although LLM can generate corresponding test cases for most DPCs, some still fail to invoke the tested APIs effectively (see Table 7). This includes situations where the API's owning class cannot be instantiated correctly, or attribute assignments are incorrect. Further fine-tuning of the LLM model with Android domain knowledge may enhance test case generation.

6.3 DopCheck's Applicability to Upcoming Android Versions

From Android 10 to Android 13, we notice that there have been slight formatting changes in the privacy change documentation with each update. For instance, Android 12 and Android 13 combined security and privacy changes into a single document. As of the submission of this paper (September 2023), Android has published the change list for Android 14 [29], although its official release has yet to come. In Android 14, privacy changes are more distributed throughout the entire document of the OS release. Take the DPC "*Schedule exact alarms are denied by default*" [28] as an example. It explicitly requires that the three APIs, i.e., `setExact()`, `setExactAndAllowWhileIdle()`, and `setAlarmClock()`, must be called with `SCHEDULE_EXACT_ALARM` permission. Although it is an apparent DPC, it is discussed as part of the functionality description of these APIs. DopCheck's DPC extraction can capture these sentences and distill the DPCs. It can be used to test Android 14 once the OS and device are available.

7 RELATED WORK

Evolution-induced issues in Android. The evolution of Android APIs has consistently been a matter of concern in the process of application development and maintenance [34, 38, 39, 45, 47, 53]. Xia et al. [69] employ a combination of static analysis and machine learning techniques to ascertain whether API compatibility issues during Android version iterations have been effectively addressed. He et al. [34] identify compatibility issues in apps by studying the API differences between different versions of Android. Li et al. [43] investigate the security threats that may arise from the evolution of Android APIs during version iterations by studying the popularity, documentation, deprecation, removal, replacement messages, and developer reactions associated with deprecated APIs. DopCheck focuses on specific privacy-related changes, such as permissions, attributes, UI and their connections. This enables it to reveal in-depth domain-specific test cases and oracles, leading to deep findings of privacy-related issues.

Documentation entity extraction and consistency check with implementation. Extracting various styles of entities from documentation has always been an essential preliminary task in research on compliance work. Bacchelli et al. [8] develop an island parsing document strategy focusing on identifying specific and meaningful information blocks (“islands”), while ignoring other irrelevant or unimportant information (“sea”). Andow et al. [5] propose PolicyLint based on NLP techniques, to automatically generate ontologies from a large corpus of privacy policies. It analyzes these policies at the sentence level to detect both positive and negative statements regarding data collection and sharing. Yan et al. [73] introduce QuPer, which structures privacy policies based on subsections, thereby analyzing their content quality from multiple perspectives.

On the other hand, with the introduction of various data protection regulations (e.g. GDPR, California Consumer Privacy Act (CCPA) [61]), there has been extensive research in multiple domains on the alignment of documentation declaration with actual application implementation [9, 41, 71, 74]. Yang et al. [74] introduce a tool named ManiScope, which utilizes NLP techniques to identify the syntactic structure of sentences, thereby being used to detect security configuration issues presented in Android manifest files. Bai et al. [9] propose ArgusDroid, which leverages the official Android documentation to establish a knowledge graph of the relationships between APIs and permissions. This knowledge graph is used for the analysis and detection of malware family variants. In other domains, Xie et al. [71] develop Skipper to identify inconsistencies between the behaviors of Alexa skills and their declared profiles in the virtual personal assistant domain.

Different from existing studies, DopCheck proposes identifying entities and subsumptive relationships for the DPC domain to construct domain-specific ontologies, which is utilized by LLMs to generate test cases. DopCheck can reduce the reliance on a vast amount of training data while ensuring accuracy for domain-specific documents.

LLMs for app development and testing. With the success of LLMs in multiple fields such as natural language processing, language translation, information retrieval and more, researchers have initiated investigations into its applicability in software engineering. Several studies have started using LLMs for automated code generation [12, 14, 36, 37, 72]. Liu et al. [44] conduct an analysis of the quality of code generated by ChatGPT and propose optimization solutions. In testing tasks, Feng et al. [17] develop AdbGPT, which uses LLMs to quickly reproduce Android system bugs. The practicality of this tool is demonstrated through a user study. Liu et al. [46] apply LLMs in GUI testing. They introduce GPTDroid, through a continuous dialogue, information retrieved from the GUI pages is relayed to the LLM, iterating through the entire process.

8 CONCLUSION

In this work, we present the first comprehensive analysis of consistency between the operational behaviors of the OS at runtime and the officially disclosed DPCs. We design and implement DopCheck, which uses a set of NLP techniques combined with an LLM to identify DPCs from Android documentation and generate test cases for DPC issues. DopCheck have identified a total of 19 bugs, with 13 of them discovered in Android 13 and 6 in Android 10 for the first time. Our work reveals that the inconsistency between the documentation claimed and Android OS actually behaves still exists. Our findings emphasize the importance of further research and action to address discrepancies in DPCs, aiming to better align documented capabilities with their actual behavior.

9 DATA AVAILABILITY

Availability. The source code of our work and relevant artifacts are available on Github [2].

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their constructive comments. This work is partially supported by Australian Research Council Discovery Projects (DP230101196, DP240103068).

REFERENCES

[1] 2023. *Beautiful Soup Documentations.* https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[2] 2024. *Investigating Documented Privacy Changes in Android OS (Source Code).* https://github.com/DopCHECK/DopCHECK

[3] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2016. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP).* IEEE, 289–305. https://doi.org/10.1109/SP.2016.25

[4] Marco Alecci, Jordan Samhi, Tegawendé F. Bissyandé, and Jacques Klein. 2024. Revisiting Android App Categorization. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE).* https://doi.org/10.48550/arXiv.2310.07290

[5] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. 2019. PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play. In *USENIX Security Symposium (USENIX security).*

[6] AppBrain. 2023. The most common Android OS versions currently installed on Android devices (phones and tablets) used by AppBrain SDK users. https://www.appbrain.com/stats/top-android-sdk-versions

[7] AppBrain. 2023. Number of Android apps on Google Play. https://www.appbrain.com/stats/number-of-android-apps

[8] Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. 2011. Extracting structured data from natural language documents with island parsing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 476–479. https://doi.org/10.1109/ASE.2011.6100103

[9] Yude Bai, Sen Chen, Zhenchang Xing, and Xiaohong Li. 2023. ArgusDroid: detecting Android malware variants by mining permission-API knowledge graph. *Science China Information Sciences* 66, 9 (2023), 1–19. https://doi.org/10.1007/s11432-021-3414-7

[10] Baidu. 2022. *Awesome multilingual OCR toolkits based on PaddlePaddle.* https://github.com/PaddlePaddle/PaddleOCR

[11] Paolo Calciati, Konstantin Kuznetsov, Alessandra Gorla, and Andreas Zeller. 2020. Automatically granted permissions in Android apps: An empirical study on their prevalence and on the potential threats for privacy. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR).* 114–124. https://doi.org/10.1145/3379597.3387469

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. (2021). https://doi.org/10.48550/arXiv.2107.03374

[13] Chinese people's congress. 2021. Personal Information Protection Law of the People's Republic of China. (2021). https://digichina.stanford.edu/work/translation-personal-information-protection-law-of-the-peoples-republic-of-china-effective-nov-1-2021/

[14] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis (ISSTA).* 423–435. https://doi.org/10.5281/zenodo.7980923

[15] Zikan Dong, Liu Wang, Hao Xie, Guoai Xu, and Haoyu Wang. 2022. Privacy Analysis of Period Tracking Mobile Apps in the Post-Roe v. Wade Era. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1–6. https://doi.org/10.5281/zenodo.7980923

[16] Zheran Fang, Weili Han, and Yingjiu Li. 2014. Permission based Android security: Issues and countermeasures. *computers & security* 43 (2014), 205–218. https://doi.org/10.1016/j.cose.2014.02.007

[17] Sidong Feng and Chunyang Chen. 2024. Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE).* https://doi.org/10.1145/3597503.3608137

[18] Google. 2019. Android 10 privacy change. https://developer.android.com/about/versions/10/privacy/changes

[19] Google. 2020. Android 11 privacy change. https://developer.android.com/about/versions/11/privacy

[20] Google. 2021. Android 12 change: Request location access at runtime. https://developer.android.com/develop/sensors-and-location/location/permissions#request-location-access-runtime

[21] Google. 2021. Android 12 privacy change. https://developer.android.com/about/versions/12/summary

[22] Google. 2021. Device Identifiers. https://source.android.com/docs/core/connect/device-identifiers?hl=en

[23] Google. 2022. Android 13 change: Granular media permissions. https://developer.android.com/about/versions/13/behavior-changes-13#granular-media-permissions

[24] Google. 2022. Android 13 privacy change. https://developer.android.com/about/versions/13/summary

[25] Google. 2023. Android 12 change: Approximate location. When an app requests precise location permissions, users can now opt to grant only approximate location permissions instead. https://developer.android.com/training/location/permissions#accuracy

[26] Google. 2023. Android 13 change: Hide sensitive content from clipboard. https://developer.android.com/about/versions/13/behavior-changes-all#copy-sensitive-content

[27] Google. 2023. Android 14 Beta. https://developer.android.com/about/versions/14

[28] Google. 2023. Android 14 change: Schedule exact alarms are denied by defaul. https://developer.android.com/about/versions/14/changes/schedule-exact-alarms

[29] Google. 2023. Android 14 features and changes list. https://developer.android.com/about/versions/14/summary

[30] Google. 2023. Determine sensitive data access needs. https://developer.android.com/games/develop/permissions?hl=en

[31] Google. 2023. Permissions updates in Android 11. https://developer.android.com/about/versions/11/privacy/permissions

[32] Google. 2023. Request runtime permissions. https://developer.android.com/training/permissions/requesting

[33] Google. 2023. TelephonyManager class summary. https://developer.android.com/reference/android/telephony/TelephonyManager

[34] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and detecting evolution-induced compatibility issues in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1145/3238147.3238185

[35] Marti A Hearst. 1992. Automatic acquisition of hyponyms from large text corpora. In *COLING 1992 Volume 2: The 14th International Conference on Computational Linguistics*.

[36] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2022. Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1–13. https://doi.org/10.1145/3551349.3556912

[37] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 1219–1231. https://doi.org/10.1145/3510003.3510203

[38] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2021. A systematic review of API evolution literature. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–36. https://doi.org/10.1145/3470133

[39] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting android api migrations using code examples. *IEEE Transactions on Software Engineering* 48, 2 (2020), 417–431. https://doi.org/10.1109/TSE.2020.2988396

[40] Ada Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. 2016. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security)*.

[41] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. 2018. Improving api caveats accessibility by mining api caveats knowledge graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 183–193. https://doi.org/10.1109/ICSME.2018.00028

[42] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 153–163. https://doi.org/10.1145/3213846.3213857

[43] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2020. Cda: Characterising deprecated android apis. *Empirical Software Engineering* 25 (2020), 2058–2098. https://doi.org/10.1007/s10664-019-09764-z

[44] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2023. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3643674

[45] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep learning for android malware defenses: a systematic literature review. *Comput. Surveys* 55, 8 (2022), 1–36. https://doi.org/10.1145/3544968

[46] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Chatting with GPT-3 for Zero-Shot Human-Like Mobile Automated GUI Testing. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.48550/arXiv.2305.09434

[47] Tarek Mahmud, Meiru Che, and Guowei Yang. 2023. Detecting Android API Compatibility Issues With API Differences. *IEEE Transactions on Software Engineering* (2023). https://doi.org/10.1109/TSE.2023.3274153

[48] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. 2012. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 51–60. https://doi.org/10.1145/2420950.2420958

[49] Alejandro Mazuera-Rozo, Camilo Escobar-Velásquez, Juan Espitia-Acero, Mario Linares-Vásquez, and Gabriele Bavota. 2023. CONAN: Statically Detecting Connectivity Issues in Android Applications. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. 2182–2186. https://doi.org/10.1145/3611643.3613097

[50] Mark Huasong Meng, Qing Zhang, Guangshuai Xia, Yuwei Zheng, Yanjun Zhang, Guangdong Bai, Zhi Liu, Sin G Teo, and Jin Song Dong. 2023. Post-GDPR threat hunting on android phones: dissecting OS-level safeguards of user-unresettable identifiers. In *The Network and Distributed System Security Symposium (NDSS)*.

[51] Kristopher Micinski, Daniel Votipka, Rock Stevens, Nikolaos Kofinas, Michelle L Mazurek, and Jeffrey S Foster. 2017. User interactions and permission use on android. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 362–373. https://doi.org/10.1145/3025453.3025706

[52] Suman Nath. 2015. Madscope: Characterizing mobile in-app targeted ads. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 59–73. https://doi.org/10.1145/2742647.2742653

[53] Juliana Oliveira, Deise Borges, Thaisa Silva, Nelio Cacho, and Fernando Castor. 2018. Do android developers neglect error handling? a maintenance-Centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software* 136 (2018), 1–18. https://doi.org/10.1016/j.jss.2017.10.032

[54] OpenAI. 2023. GPT-4 Technical Report.

[55] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with gpt-4. *arXiv preprint arXiv:2304.03277* (2023). https://doi.org/10.48550/arXiv.2304.03277

[56] Personal Data Protection Commission. 2012. PERSONAL DATA PROTECTION ACT 2012. (2012). https://sso.agc.gov.sg/Act/PDPA2012

[57] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, Phillipa Gill, et al. 2018. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. In *The 25th Annual Network and Distributed System Security Symposium (NDSS)*.

[58] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th USENIX security symposium (USENIX security)*.

[59] Spacy. 2020. *SpaCy Documentations*. https://spacy.io

[60] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. 2017. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE communications surveys & tutorials* 20, 1 (2017), 465–488. https://doi.org/10.1109/COMST.2017.2779824

[61] State of California Department. 2018. California Consumer Privacy Act (CCPA). (2018). https://oag.ca.gov/privacy/ccpa

[62] Statista. 2023. Global market share held by mobile operating systems from 1st quarter 2009 to 2nd quarter 2023. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[63] Thomas Sutter and Bernhard Tellenbach. 2023. FirmwareDroid: Towards Automated Static Analysis of Pre-Installed Android Apps. In *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 12–22. https://doi.org/10.1109/MOBILSoft59058.2023.00009

[64] The European Parliament. 2016. GDPR-Right to be Informed. https://gdpr-info.eu/issues/right-to-be-informed/

[65] The European Parliament. 2016. General Data Protection Regulation. *Official Journal of the European Union* (2016).

[66] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Can Large Language Models Write Good Property-Based Tests? *arXiv preprint arXiv:2307.04346* (2023). https://doi.org/10.48550/arXiv.2307.04346

[67] Jerry Wei, Jason Wei, Yi Tay, Dustin Tran, Albert Webson, Yifeng Lu, Xinyun Chen, Hanxiao Liu, Da Huang, Denny Zhou, et al. 2023. Larger language models do in-context learning differently. *arXiv preprint arXiv:2303.03846* (2023). https://doi.org/10.48550/arXiv.2303.03846

[68] B. Wolford. 2022. What are the GDPR fines? https://gdpr.eu/fines/

[69] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, et al. 2020. How Android developers handle evolution-induced API compatibility issues: a large-scale study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 886–898. https://doi.org/10.1145/3377811.3380357

[70] Fuman Xie, Chuan Yan, Mark Huasong Meng, Shaoming Teng, Yanjun Zhang, and Guangdong Bai. 2024. Are Your Requests Your True Needs? Checking Excessive Data Collection in VPA Apps. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3597503.3639107

[71] Fuman Xie, Yanjun Zhang, Chuan Yan, Suwan Li, Lei Bu, Kai Chen, Zi Huang, and Guangdong Bai. 2022. Scrutinizing privacy policy compliance of virtual personal assistant apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1–13. https://doi.org/10.1145/3551349.3560416

[72] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10. https://doi.org/10.1145/3520312.3534862

[73] Chuan Yan, Fuman Xie, Mark Huasong Meng, Yanjun Zhang, and Guangdong Bai. 2024. On the Quality of Privacy Policy Documents of Virtual Personal Assistant Applications. *Proceedings on Privacy Enhancing Technologies* (2024). https://doi.org/10.56553/popets-2024-0028

[74] Yuqing Yang, Mohamed Elsabagh, Chaoshun Zuo, Ryan Johnson, Angelos Stavrou, and Zhiqiang Lin. 2022. Detecting and Measuring Misconfigured Manifests in Android Apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3063–3077. https://doi.org/10.1145/3548606.3560607

[75] Zhiju Yang and Chuan Yue. 2020. A comparative measurement study of web tracking on mobile and desktop environments. *Proceedings on Privacy Enhancing Technologies* 2020, 2 (2020). https://doi.org/10.2478/popets-2020-0016

[76] Jiahao Yu, Xingwei Lin, and Xinyu Xing. 2023. GPTFUZZER: Red Teaming Large Language Models with Auto-Generated Jailbreak Prompts. *arXiv preprint arXiv:2309.10253* (2023). https://doi.org/10.48550/arXiv.2309.10253

[77] Kaifa Zhao, Xian Zhan, Le Yu, Shiyao Zhou, Hao Zhou, Xiapu Luo, Haoyu Wang, and Yepang Liu. 2023. Demystifying privacy policy of third-party libraries in mobile apps. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE48619.2023.00137

Received 2023-09-28; accepted 2024-04-16