Content for Everyone: Detecting Misconfigurations of Android Content Providers

Christopher Lenk[1(✉)], Tim Lange[2], and Johannes Kinder[2]

[1] Zentrale Stelle für Informationstechnik im Sicherheitsbereich (ZITiS), Munich, Germany
christopher.lenk@ifi.lmu.de
[2] Ludwig-Maximilians-Universität München (LMU Munich), Munich, Germany
{tim.lange,johannes.kinder}@lmu.de

Abstract. Mobile apps record a plethora of personal and privacy-sensitive data. Within Android, apps can intentionally share data with each other through *content providers*, with access governed by settings and custom permissions. Misconfiguration of a content provider by a developer can allow an attacker to leak or tamper with private application data. In this paper, we study how content providers protect or do not protect private data in a systematic study on 321,340 Android apps. We identify potentially vulnerable apps using a staged approach employing static and dynamic analysis and are able to validate content provider leaks in 302 apps. In all cases, we successfully access information that is either directly privacy-relevant or would support further attacks. We select ten applications for detailed case studies that demonstrate the effects of data leakage from content providers in practice.

Keywords: Android · content provider · data leakage · static analysis · privacy.

1 Introduction

Mobile devices such as smartphones store a large amount of personal data, from chat logs over shopping preferences to financial and health records [8]. In addition to system-wide storage locations for documents or images, much personal data is found in internal data structures of applications (apps) [15]. Mobile operating systems implement security mechanisms designed to block uncontrolled access to data, function calls and app components as a protection from data exfiltration or manipulation. The most widely used mobile operating system, Android [28], implements isolating sandboxes to increase security. System components and applications are separated from each other by default and all interaction is moderated by the operating system. If an application needs to communicate with another application or a sensitive system component, e.g., to obtain contact information, it has to construct an *intent* to launch that app or directly access the data via a *content provider*.

Published in 21st International Conference on Availability, Reliability and Security, Linköping, Sweden, August 24 – 27, 2026

Inter-app communication is a well-known attack surface for Android applications [9,32]. If a victim application does not sufficiently protect its endpoints, then an attacker application can retrieve, intercept or manipulate information. In 2012, Zhou and Jiang [32] described *content provider leakage*, a then widespread issue where content providers were accessible by default, making potentially sensitive data available to other apps. Google responded in November 2012 with the release of Android version 4.2 (API level 17) by introducing a new flag `exported` for content providers that had to be explicitly set for making them available. However, despite the reduced attack surface, there were several examples of successful attacks against content providers in the following years [20,22,27]. In current versions of Android, content providers with potentially sensitive information are meant to be protected both by the `exported` setting and *custom permissions*. Custom permissions can be defined by app developers, and a third party app that wishes to access a content provider must obtain the corresponding permission first. Like regular system permissions, these custom permissions are granted to an application either at installation time or interactively at runtime, depending on the *protection level* defined for the permission. While there are some recommendations on designing permissions in the Android documentation, no specifications are enforced and developers have full control on what to expose of their applications and how.

In this paper, we focus our attention on apps implementing a content provider that is set to be `exported` and protected by *normal* permissions (protection level zero). We argue that this combination of settings should be considered a misconfiguration. Declaring a permission required for accessing the content provider suggests that the developer intended to protect the information within; but using a normal permission has no discernible effect since the Android system will grant it upon installation without alerting the user. In fact, such permissions are hidden by default in the design of the app information in the Google Play Store. As shown in Figure 1, the details are nested in a structure of two linked pages including a variety of information. Furthermore, the descriptions of custom permissions that have to be provided by the developer are often inaccurate or missing entirely. Therefore, even if a user decides to inspect the list of permissions, they may not understand that a given custom permission mediates access to their data in another app [10]. If a content provider is meant to be openly accessible, it should not be protected by any permission. Otherwise, it should be protected with either (i) a *signature permission*, limiting access to apps by the same developer, or (ii) a *dangerous permission* (also called *runtime permission*) that requires an informed user decision. This is a basic principle of the Android security concept, whereby an application should request and define as many permissions as are necessary for its functions and security [18].

In our work, we demonstrate that such misconfigurations are prevalent and content providers are often inadequately protected. As a result, the issue of content provider leakage persists at the application level and can be exploited in currently available Android apps more than ten years after system-wide countermeasures were implemented. We examine the configurations of 321,340 apps

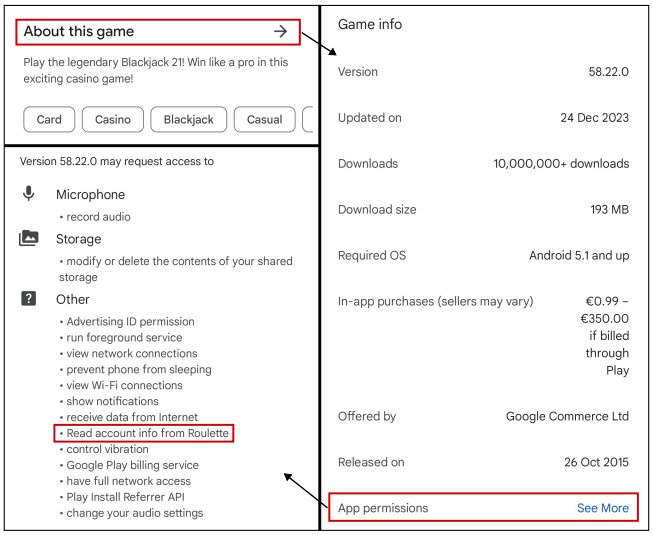

Fig. 1. Design of app information in the Google Play Store.

defining content providers with custom permissions and determine 380 apps where the providers are exported and equipped with insecure parameters. We validate the exploitability on 358 targets, checking the impact on data security and user privacy. In addition to general information and configuration data, we are also able to read user-entered data, including potentially sensitive personal information. In summary, we make the following contributions:

- We develop an automated analysis to search for configuration mistakes in the definition of content providers and custom permissions belonging to them. We conduct a study on the Google Play Store to check the data structures of current Android apps downloaded to up to millions of mobile devices for accessible information with impact on user security and privacy.
- We validate apps discovered in our study with an automated system implementing dynamic validation on a real Android device and static validation with data flow analysis.
- We discuss concepts for improving the data security of the Android system in the area of inter-app communication via content providers.

This paper is organized as follows: We begin by providing necessary background information (§2). We then describe the methodology and implementation of finding and exploiting content providers (§3), quantitatively analyze the results (§4), and present the individual case studies (§5). Finally, we discuss the results and potential mitigations (§6), review related work (§7), and conclude (§8).

2 Background

This section provides an overview of the security design and relevant properties of Android (§2.1), followed by a brief description of Android content providers (§2.2) and the tool FlowDroid used for static data flow analysis (§2.3).

2.1 Android Security Mechanisms

Android relies on Security-Enhanced Linux (SELinux) to implement mandatory access control across the system. In particular, applications are strictly sandboxed and reside in the system as separate users [12]. The Android system restricts access to data structures and actions outside the application sandbox with *permissions*. Permissions may be granted to an app through mechanisms depending on their associated *protection level*. Since the introduction of Android 6.0, permissions can be differentiated according to the way they are checked [11].

Install-time permissions are intended for access to resources and components with limited impact on security and privacy. At installation time, the user can see a list of permissions, which are then permanently granted to the application. This category encompasses *normal* and *signature* permissions. A normal permission (level zero) is considered low-risk and includes no additional checks. Signature permissions (level two) are granted only if the requesting app was signed with the same certificate as the app or the operating system that defined the permission.

Runtime permissions or *dangerous permissions* (level one) are intended for critical system resources or privacy-sensitive data [17]. In contrast to the first group, these are not automatically granted, but must be explicitly confirmed at runtime by the user via a selection dialog, which explains the actions to be carried out.

Other protection levels are available but used in limited contexts, such as for system applications. An overview can be found in the Android documentation[3]. While the Android system specifies permissions with fixed protection levels for device functions and hardware components, developers can define their own permissions for specific functions of their app and set individual protection levels. The recommended syntax consists of a reverse domain-style app identifier and a functional ending written in upper case, which can be for example `com.example.app.permission.ACCESS`. The Android documentation provides best practices and security recommendations, but there are no checks as to whether a permission is configured appropriately and proportionately [16].

2.2 Data Sharing with Content Providers

In the case that apps need to exchange data with each other and thus want to access memory protected by a sandbox, the developer can define a content provider. By default, a provider has two basic methods, `onCreate` and `getType`, as well as four main functions that reflect the access options of databases, `query`,

[3] https://developer.android.com/reference/android/R.attr#protectionLevel

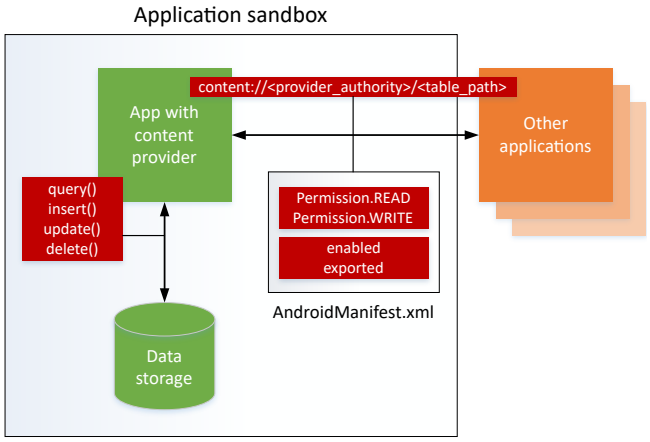

Fig. 2. Schematic overview of content providers [14].

`insert`, `update` and `delete`. The configuration of the provider takes place in the `AndroidManifest.xml`. It is possible to assign custom permissions for protection, either a single common permission or two separate permissions for read and write access. The two parameters `enabled` and `exported` determine whether the provider is active overall and can be called from outside the sandbox. When these parameters are set to `true` and another app has been granted the permissions, it can access the provider's functions and data, as shown in Figure 2. It does so via a content resolver and a corresponding Uniform Resource Identifier (URI), which consists of an authority and a path. The authority identifies the respective provider, the appended path is the name of the database table or the identifier of another structured set of data. The methods of the content resolver implement basic storage functions for creating, retrieving, updating, and deleting data (CRUD), with queries and arguments matching the SQL language. It is possible to access individual values using the assigned names and to use selections and sort orders [13].

2.3 Static Data Flow Analysis

FlowDroid [4] is a tool for static data flow analysis for Android and Java programs based on the Soot framework [29]. FlowDroid parses the Android manifest and configuration files containing the app layout of the Dalvik executable files (DEX) and analyzes the Jimple representation to iteratively build a dummy main method emulating the Android lifecycle and callbacks of the app components. The subsequent taint analysis tracks information flows from sources to sinks and reports leaks with a witness path. The analysis is based on the IFDS framework (*Interprocedural Finite Distributive Subset problems*) and is flow-, context-, field-, and object-sensitive. It uses a multi-solver IFDS setup to resolve aliases on-demand.

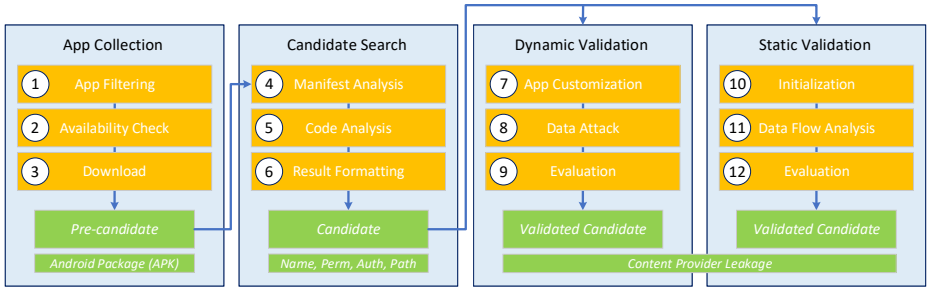

Fig. 3. Automated analysis workflow of ProseccoFlow.

3 Methodology

We now introduce our automated analysis tool `ProseccoFlow` (for **Pr**ivacy-**O**riented **S**canning of **E**xported **C**ontent providers by **Co**mbining **Flow**s) to find and verify insufficiently secured content providers. To set the right context for our work, we begin by defining the threat model (§3.1). We collect current app versions (§3.2) and take a closer look at the parameters of unprotected content providers and search for opportunities to exploit them (§3.3). Finally, we present a method to validate the vulnerabilities dynamically on a real Android device (§3.4) and statically with a data flow analysis (§3.5). Figure 3 and the included circled numbers show the workflow of `ProseccoFlow`. We make all source code and data available as open source under a GitHub link[4].

3.1 Threat Model

The *target app* is some application processing and storing potentially sensitive information implementing a content provider. We assume that an attacker has access to the Android Package Kit (APK) of the target app, e.g., by downloading it from the Google Play Store. When the attacker finds a misconfiguration, they extract the necessary information for accessing the data from the app. The attacker then creates or updates an *attack app* under their control. The attack app may be entirely created by the attacker or contain just fragments of attacker code, e.g., through a library on the software supply chain.

A potential victim has the target app installed; through regular use, the target app has recorded some sensitive information. For the attack to happen, the victim must install (or update, if already installed) the attack app. The app can be distributed via official channels such as the Google Play Store. No further user interaction is needed; the system grants the permissions at install time. The attack app can access the content provider of the target app on the device and, depending on the range of functions, systematically read and change data. The data is transferred from the device via the network.

[4] https://github.com/lmu-plai/ProseccoFlow

3.2 App Collection

We conduct an analysis on all Android applications implementing content providers and custom permissions registered on the Google Play Store by May 2025 ①. To simplify and speed up the crawling process, we rely on the existing Android application collection AndroZoo [2]. AndroZoo regularly crawls several application stores and malware collections. We process the Play Store to draw from them only those apps that are currently available and actively downloaded by users ②. For each app, we obtain from AndroZoo and the Play Store the latest APK, the version code, the download numbers, and user ratings ③. If the app is implemented as an Android App Bundle (AAB) splitting code and configurations into multiple APKs, we use the alternative app store APKPure to get the complete Extended Android Package Kit (XAPK).

3.3 Candidate Search and Preanalysis

We continuously process the Android apps, extracting and storing data from the `AndroidManifest.xml` on the fly ④. From the manifest, we obtain information about the presence of content providers, their security parameters and the custom permissions with protection levels. If an application implements at least one exported content provider with a normal permission, it allows any third-party app to interact with the content provider once it is installed on the device.

To validate whether this results in data leakage from the application, we now perform an initial static code analysis ⑤. An attacking app accesses data objects via the registered content URI of the provider with authority and path. We obtain the provider authority from the Android manifest of the target app. To get the paths, we statically analyze the byte code of the class implementing the content provider as well as external references and extending classes. We extract any string literals in calls to methods like `addURI` and `appendPath`, which register a URI pointing to a specific database table or data structure when the associated provider is created. Alternatively we search the code for strings containing `SELECT` queries and complete content URIs that are not included in method calls. We consider any such string literal a candidate path to be validated. Our static analysis is implemented on top of Androguard and its integrated decompiler DAD. This completes the candidate search and preanalysis, and we prepare the extracted information for processing in the validation stages as tuples of provider name, its permission(s), authority and path(s) ⑥.

In the next stages, we dynamically validate whether information can actually be *accessed* at runtime, and we statically validate that information in the content provider is potentially *sensitive*.

3.4 Dynamic Validation

We dynamically validate the identified attack path on current app versions using `ProseccoApp`, a customizable attack application that captures and outputs any available data from misconfigured content providers. `ProseccoFlow` processes

the target apps continuously and uses `apktool` and `apksigner` to create a unique version of the `ProseccoApp` customized for each target app by decoding the template APK of `ProseccoApp`, inserting the required custom permission(s), as well as reassembling and signing the APK ⑦. Finally, it installs and launches both apps automatically on the device via the Android Debug Bridge (ADB). The intent for initializing the `ProseccoApp` contains the provider authority and path(s) for attacking the target app.

`ProseccoApp` starts with registering the intent parameters ⑧. It merges the authority and a path of the content provider selected for the attack to create a complete URI. The app uses a content resolver with a URI to access the columns of the connected table or a similar data structure using the provider's query method. In this way, `ProseccoApp` can read and save content referenced with a validly registered URI to hidden files in the documents directory of the user. Our app repeats the queries for all URI paths that we extracted from the program code using static analysis ⑤.

If the write permission is also granted or is included in a central permission, we can use the methods to insert new data and delete or update existing entries if they are implemented in the tested app. Apart from denial of service and tampering with personal data, this can potentially enable further app-specific exploits by injecting specially crafted strings into an internal data structure.

We transfer the results from the device via ADB and store the information of whether the data attack was successful ⑨. If structured data is included in the results file, we have found a dynamically validated candidate for content provider leakage. To assess the impact, we carry out a usage analysis: we automatically check how many app versions in the AndroZoo dataset have actually applied for permission to access the validated content provider in their Android manifest.

3.5 Static Validation

In parallel to dynamic validation, we integrate FlowDroid in our analysis system to statically validate that an app leaks sensitive data via its content provider. This allows us to determine the source of the data and to examine the type of information. To perform the flow analysis, we use a list of sources of potentially sensitive information based on [23] and define the return statement(s) of the content provider's query method as sinks. A challenge for static information flow analysis arises from content providers frequently storing data in SQLite databases or property objects. This indirection is prone to stripping taint information, such that a direct information flow between sensitive source and the return statement cannot be established. Therefore, we additionally define the parameters of methods used to manipulate data in content providers such as `insert()` and `update()` of `SQLiteDatabase` as sinks ⑩ and the corresponding query methods as additional sources, so that we can connect the separate taint flows into and out of the content provider after analysis finishes.

We configure FlowDroid to use the access path-based library summaries from StubDroid [3] and added summaries for the classes `ContentValues` and

Table 1. Categories of vulnerable app candidates.

App category	Count	App category	Count	App category	Count
Education	72	Entertainment	8	Social	3
Tools	52	Food & Drink	7	Family	2
Business	49	Travel & Local	7	House & Home	2
Games	23	Auto & Vehicles	6	Shopping	2
Communication	21	Video Players	6	Art & Design	1
Productivity	20	Books & Reference	5	Comics	1
Lifestyle	16	Weather	5	Medical	1
Finance	14	Photography	4	News & Magazines	1
Health & Fitness	12	Maps & Navigation	3	Parenting	1
Personalization	10	Music & Audio	3	Sports	1

`MatrixCursor` that are required for tracking data across content provider implementations. FlowDroid then performs information flow analysis between the data sources and sinks (11). If the target app implements multiple DEX files, we merge these inside FlowDroid, to ensure that the search for leaks covers the whole application. We define timeouts of five minutes each for data flow analysis, the collection of callbacks and the witness generation.

The results of FlowDroid list the information flows leaking sensitive data specified with the path, name and affected parameters of the source and sink methods. We use the list of methods implementing provider functions we collected during static analysis (5) to automatically check if methods belonging to the identified content provider are the sink of a data flow registered as a leak (12). In this case we consider the content provider as statically validated.

4 Results

We now present the results of applying our methodology to all apps in the Google Play Store. We show the results of filtering for misconfigured candidate apps (§4.1), followed by dynamic (§4.2) and static validation (§4.3), and finally an overview of the types of data leaked (§4.4).

4.1 Discovering Vulnerable Content Providers

We applied our filtering approach to the entire dataset of applications on Google Play as of May 2025. 468,220 apps defining content providers and custom permissions are currently available in the Google Play Store. If only apps with at least one thousand downloads are considered, 321,340 APKs or XAPKs remain as pre-candidates. 380 apps have the parameter `exported` set to `true` and at least one provider permission with a normal protection level. After the static analysis and the search for URI paths, our system registers 358 applications as candidates for validation. The apps selected for the case studies have thousands to several million downloads and span a wide range of categories (see Table 1).

Table 2. Groups of exploitable data fields.

Data group	Data objects	Dyn.	Stat.	Both
Informational	Push notifications and log messages like warnings	61	70	59
Configuration	User settings, preferences and app parameters	258	245	199
Application	User input like texts, lists and search requests	63	109	24
Personal	Names, communication, location and health data	109	92	86

4.2 Dynamic Validation

We implemented the dynamic validation using four Samsung Galaxy S22 Ultra as real devices with Android version 15 (API level 35). `ProseccoFlow` installs the 358 candidate applications on the devices and tests them in an automated run. We are able to successfully access the content providers of 332 candidates. For 26 apps, we get no result files due to Android system errors during app installation or provider initialization. These apps do not support the current Android version, because they receive no updates anymore. Some of them have no configuration files for Samsung devices or try to load device- or vendor-specific libraries, that are not delivered with a Samsung Galaxy S22 Ultra. In 79 cases, we can not apply our strategy for attacking the components and extracting data for the following reasons: The apps or their providers are not functional or have empty data structures. Some apps implement special forms such as API or method providers or use strategies to hinder finding correct URI paths. This leaves 253 apps where 491 data fields shared with content providers are openly accessible.

4.3 Static Validation

We run FlowDroid in its current version 2.15 with four instances in parallel. The data flow analysis processes 349 candidates without errors. In 104 cases, the detection of the implemented content provider as a leaking sink for sensitive data is not successful for the following reasons: The apps have no sensitive sources or use methods that are app-specific and not included in our collection of sources. Obfuscated candidates and applications using native methods, data access objects or generated code also belong to this group. Some apps are not correctly implemented or use the methods of the content provider not as intended. We can successfully validate 516 data fields included in the content providers of 245 apps which means that data shared with them comes from sensitive sources.

4.4 Classes of Leaked Data

Table 2 shows the categories of data collected when successfully accessing the content providers of the apps. Note that some providers share multiple types of data. We determined which class the accessible information belongs to by semi-automatically inspecting the results from dynamic and static validation. We can prove provider access and data leakage for 368 data fields of 196 applications with

both validation methods. We provide some details on the classes of information being leaked in the following paragraphs.

Informational Data. This category includes IDs and message contents of external push notifications, as well as log messages that are released by providers for other apps. In some apps, notification providers enable access to verification data such as hash values and IDs. While this does necessarily contain no direct personal data, the timing and contents of push messages and other metadata for the application may leak sensitive information or can be used to deanonymize users. Given that separate system permissions exist for reading log data and accessing broadcast messages, making such information freely accessible via a content provider clearly violates the design of the Android permission system.

Configuration Data. Configuration data includes rules and parameters set by the user, such as screen formats, language settings and login preferences. Some apps share status and configuration data from user accounts, as well as version and hardware information. This also includes session and user IDs and assigned rights. Many of the examined apps store and share timestamps for the creation and updating of specific data entries. In terms of impact on user security and privacy, configurations provide more detail than informational data and can aid malicious actors in preparing further targeted attacks. In the case studies, we can find the largest number of apps with insufficiently protected providers sharing data from this category.

Application Data. This class covers content the user entered into the app at some point. This includes free-form text, numbers, task and calendar entries, favorites, or entered search terms. The scope varies between apps and ranges from individual data objects to the entire app content depending on the implemented functionality of the provider. Internal app content and search queries that are publicly available threaten user privacy. Depending on the function of the app and the personal value of the entered data, these can be misused with varying degrees of severity.

Personal Data. Personal data is the class with the highest relevance to privacy. In contrast to general application data, personal data has direct reference to a person, e.g., names, addresses and communication details, as well as financial and health information. The examined apps show that misconfigurations can have particularly critical consequences when a provider releases data structures that contain personal data. Through an unprotected content provider, any app can access and further leak privacy-sensitive information, effectively bypassing the built-in security mechanisms for protecting user data on Android.

5 Case Studies

In the following, we conduct case studies on ten applications detected by our previous study. Table 3 provides an overview of the studied apps with their download numbers on the Google Play Store in May 2025 and the analysis results sorted by the discovered use case. In all case studies, we disclosed our findings to the developers.

Table 3. Results of case studies

Category (Mitigation)	App	Downl.	Impact	Use	Reaction
Exchange within app families (Normal to signature-based or dangerous)	Diet app	100K+	Health	1	No fix
	Learning platform	1M+	Personal	1	
	Casino game	10M+	Login	14	Will fix
	Agricultural app	5K+	Economic	1	Fixed
	Blood sugar app	100K+	Health	1	
Public access (Normal to dangerous, Data scope and export necessity)	Shopping list	1M+	User-entered	7	Removed
	Flight tracker	10M+	User-entered	0	Fixed
	Logistics app	10K+	Login	0	Fixed
Access for widgets (Normal to dangerous, Data scope)	Task planner	5M+	User-entered	4	
	E-mail app	5M+	Personal	10	

Diet App. The app with the package name `com.ro.atoming.slim90days.free` supports users in weight loss by planning and controlling nutrition. A content provider in the app shares the user's current weight they have entered into the app. Such personal health information is usually specifically protected in built-in applications such as the "Health" app in the Apple ecosystem and requires explicit user consent for read or write access.

Shopping List. The app with the package name `org.openintents.shopping` manages shopping lists including price and priority. The content provider enables access to eight tables for items, lists, and notes, covering most of the user-generated content. We are able to read and write the app's data and to delete existing entries; writing invalid entries crashes the application and makes it unable to start.

Learning Platform for Civil Service. The app with the package name `com.edurev.upsc` is one of a group of 71 used to prepare students for post-graduate exams for the public sector of India. All apps use a content provider for settings, app data and personal information such as name, place of work, taken courses, learning progress, and connected followers. The retrievable profile of the user covers all data categories with the exception of informational data.

Casino Game. The app with the package name `com.kamagames.blackjack` is the most downloaded variant of a family of 13 online gambling games. The apps include a content provider for exchanging information about the login process. This involves sensitive data such as the login type and the authentication token. An attacker can retrieve the login token from the content provider to steal a session if the user is logged in. Given that the game allows users to spend real money, this is clearly a real threat.

Agricultural App. The app with the package name `fr.obione.cownotes` belongs to a family of eleven agricultural apps and is designed to manage cattle. The content provider reveals personal information such as names, e-mail and physical addresses of contacts, as well as data about animals and equipment.

Attackers can read or manipulate all data, which is personally and commercially relevant information.

Task Planner. The app with the package name `com.ticktick.task` includes a list and calendar feature for task management. The content provider makes individual entries and their properties such as priority, sort order and due dates available. This can include personal or private information and enable insights into the user's plans and everyday life.

E-Mail App. The app with the package name `org.kman.AquaMail` can manage multiple e-mail accounts and exports two content providers. The first includes the name of the mail account and the number of total and newly added unread messages, indicating that it is meant to be used for notifications and widgets. The second provides the messages themselves, including full address, subject, time and body text. The data allows an attacker to read the contents of e-mails and the names of accounts, which reveals personal information.

Flight Tracker. The app with the package name `com.flightaware.android.liveFlightTracker` is used to check the status of and information about flight connections, airlines and airports. The content provider offers general information about airports and airlines, user search terms, and favorites. While the data is not directly personal, it may provide indirect information about travel plans.

Logistics App. The app with the package name `com.cemex.foreman.eu` tracks status and delivery of goods for the construction industry. The implemented provider reveals user account data and two access tokens, which can be sufficient to steal user sessions and thus obtain additional information about shipments.

Blood Sugar App. The app with the package name `com.philosys.gmateon` is the companion app for a physical blood sugar monitoring device. It exposes the user's glucose level and the measurement conditions including date and time through a content provider, which is sensitive medical data.

6 Discussion

We now discuss the results of our study (§6.1) and potential improvements to the Android system (§6.2), followed by the limitations of our study (§6.3).

6.1 Impact

In §4, we identified hundreds of currently available apps where the configuration of the content provider does not match the recommended security requirements of the shared data and also differs from other apps with a similar provider functionality, making privacy-relevant data openly accessible. The case studies in §5 provide examples of these potential attacks in detail. Through these misconfigurations, malicious actors can learn or modify important app-specific data. While health data, e-mail contents, and login information are obviously privacy-sensitive, other data such as configurations, user names, or logs, which may seem benign at first, can also enable the construction of detailed profiles over longer

periods of time. Daily habits, personal preferences, appointments, and tasks can all be stepping stones for further attacks, including social engineering.

6.2 Mitigations

The issue of misconfiguration points to a general problem in the concept of data sharing in the Android system. Although several security improvements have been implemented in response to attacks [9,32], it still offers the potential for exploiting inter-app communication. Application-specific security is primarily based on recommendations and less on standardized and mandatory restrictions. The developer can flexibly define and configure the components of the application. This includes security-relevant parameters in the Android manifest, which may then be inadequately defined. This is often related to the fact that the underlying concept and the effects of the parameters have not been fully understood or misjudgments arise in the important connection between the value of the included data and the available security levels. Since there are no checks whether a permission or a content provider has been declared sufficiently secure, the app components are implemented without protection against data attacks and allow the basic security mechanisms of the Android system to be circumvented. The purpose of the permission system for regulated access is undermined, since rights are automatically granted without additional checking by the user or the system.

It is necessary to improve security from both sides of the app development process, from the developer and the system side. On the one hand, this means that developers of Android apps should be made aware which impact mistakes in the app configuration can have and which influence the parameters within their app have on the entire Android system and the processed data. On the other hand, an improved concept for the effective protection of content providers is required. One possibility would be to improve the concept of data access in the provider implementation. For example, it is possible to place a component at database level or between the data structure and the provider, which checks the requests on the basis of defined rules and, if necessary, adapts them or blocks them completely [1,7,25]. In this case, the security of data access could be increased in addition to the declarations of the Android manifest. However, this component harbors further possibilities for misconfigurations if the definition of rules is left to the developer. This risk is less with access rules set by the system, but legitimate requests to the data structures could be rejected by restrictions that are not adapted enough to specific app functions.

At the app level, there is the option of systematically checking the Android manifest for incorrect or missing configuration entries. A developer could search for errors in the configuration files with methods of static analysis before the corresponding application is published [19,21]. Existing tools such as CodeQL[5] can also support this search. This is an effective method to prevent the configuration vulnerability, but the developer needs to be made more aware of the importance and the currently available debugging software.

[5] https://codeql.github.com/codeql-query-help/java/java-android-incomplete-provider-permissions/

Another approach in this context would be to standardize the parameters of content providers in the Android manifest, so that it would be specified how the parameter values should be designed for certain types of information. In connection with the approach by Yang et al. [30], the creation of a manifest scheme based on security recommendations in the Android documentation would be conceivable, so that the configurations created would be compared with a secure standard and adjusted accordingly.

There could also be groups of special providers with security standards according to the category of provided data. In the case of high-value data, the optional declaration of permissions should be suspended and made mandatory. It would be necessary to introduce a restrictive security check in the Google Play Store that only allows apps whose configuration meets the recommended security requirements. Currently, Android is relatively open with respect to the possibilities given to app developers. As the value of the app-specific data continues to increase, however, we have to ask whether a more restrictive system with fixed policies would be desirable and lead to higher user privacy.

6.3 Limitations

The developed methodology was not applicable on all candidate apps because parameters like the path of the content URI could not be determined due to obfuscation efforts. As a result, the information required for the attack could not be obtained from the provider's methods via the automatic extraction strategy. A manual examination of the program code of the test elements in question may reveal further examples suitable for the attack strategy.

Certain apps or their provider function were defective and not functional. In some cases, registration processes that could not be completed due to specific data such as company IDs or general bugs blocked the remaining functions of some apps, making it impossible to test their provider with the attack.

7 Related Work

We now review related work in inter-app vulnerability analysis (§7.1), including an experimental comparison to a commercial analyzer (§7.2), and studies of misconfigurations in the Android manifest (§7.3).

7.1 Inter-app Vulnerability Analysis

The development of methods and tools for the detection and analysis of vulnerabilities in Android's inter-app communication increased in 2012 with the discovery of attacks on app components. Table 4 provides an overview of related work in this area. We categorized the approaches based on the fact if they investigated the passive leak of data shared with content providers. We also checked if the authors implemented a systematic search for vulnerabilities in real applications and a dynamic test of data leaks in a realistic Android system environment.

Table 4. Overview of related work on inter-app vulnerability analysis

Year	Work	Dataset	Passive	System	Dynamic	Config	Impact	Usage	Mitigation	Context
2013	Zhou and Jiang [32]	62.5K	●	●	●	◑	●	○	●	Provider leakage
2014	Shahriar and Haddad [27]	1.2K	●	○	●	◑	○	○	○	Provider leakage
2015	Bagheri et al. [5]	0.5K	○	○	○	○	●	●	○	Permission leakage
2015	Li et al. [24]	16.3K	●	○	○	○	●	○	○	ICC taint analysis
2017	Hassanshahi and Yap [20]	0.9K	●	○	○	◑	●	○	●	DB vulnerabilities
2017	Bosu et al. [6]	110.5K	●	●	○	○	●	○	●	ICC taint analysis
2018	Johnson et al. [22]	18.9K	○	●	○	○	○	○	○	Component fuzzing
2021	Samhi et al. [26]	136.4K	●	○	○	○	○	○	○	ICC taint analysis
2023	Gamba et al. [10]	2.2M	●	●	●	○	●	○	●	Permission study
2025	Our work	321K	●	●	●	●	●	●	●	Provider leakage

We evaluated the case studies if they address the underlying misconfiguration and the usage by other apps as well as the impact and possible mitigation of content provider leakage. The filling of the circles shows how an approach meets the respective criterion.

Zhou and Jiang [32] conducted a systematic study of vulnerabilities in content providers and their existence in 62,519 Android applications from multiple markets. Candidate apps were selected using a three-stage system with tests for existing vulnerabilities at code level, their exploitability and the value of included data. The authors were able to read data fields and to change settings and configurations. This study includes a test strategy and a discussion of impact and possibilities for improvements, but only a general example of misconfiguration in the case studies. Details about the implementation of listed examples are not included and a usage analysis was not carried out.

Shahriar and Haddad [27] developed a methodology for detecting vulnerabilities in content providers. Elements of a secure provider implementation are compared with the app under test to uncover weaknesses in the implementation of the content provider on basis of three security principles. The authors worked with a small dataset that was created randomly, so no systematic study was carried out. The cause of content provider leakage is shown with a general example of misconfiguration. Test cases were analyzed without considering impact, provider usage and mitigation.

Bagheri et al. [5] presented a tool for analyzing inter-app vulnerabilities. The work primarily deals with permission leakage based on privilege escalation, which is an active attack to enable data leakage. The apps for the tests were collected based on download numbers from various sources, without systematically examining the presence of insecure configuration parameters. A dynamic test of

passive content provider leakage was not carried out. While impact and usage are being analyzed, no mitigation proposals have been made.

Li et al. [24] conducted a study on the use of inter-component communication in benign and malicious apps. They developed a methodology to check the communication of 16,260 applications for leaks of sensitive data based on static analysis and the creation of a control flow graph. The implemented tool collected 2,929 corrupt paths with the possibility of data leakage. The methodology is tested against several test cases that are known to contain vulnerabilities, but not against real apps. The impact of data leakage is named, while the configuration, provider usage and approaches for improvement are not evaluated.

Hassanshahi and Yap [20] introduced new classes of vulnerabilities in accessing Android app databases. They separated attacks on publicly made data structures via unprotected content providers and on private data structures via unsecured components such as intents. The work does not use concrete examples of misconfiguration. The apps under test were selected based on download numbers without a systematic search for apps with the identified vulnerabilities. There is neither a dynamic test nor an analysis of the use of the integrated database connections. The impact and mitigation of the attacks were analyzed.

Bosu et al. [6] developed a method based on 110,550 apps to detect and analyze vulnerabilities in sensitive inter-app communication streams and collusion attacks. The study identified six categories of threats, including data leakage and privilege escalation. Dynamic tests were not carried out. The case studies include effects and mitigation, but no details on configuration and use.

Johnson et al. [22] published a study with 32 Android devices and 59 different Android versions. The dataset contained the 18,583 most downloaded applications from 2016 and 2017 in several versions. The authors developed the fuzzing framework `Daze`, which extracts app components and tests all communication interfaces. In this way, a total of 14,413 vulnerabilities were found. The work does not address passive data leakage that is possible due to the vulnerabilities, neither the causes in the configuration nor the impact on the user. The specific use of the tested components and mitigation possibilities are also not discussed.

Samhi et al. [26] have examined atypical communication paths between Android components and apps. They presented a static approach for modeling previously undetected inter-component communication links to detect vulnerabilities or data leaks. The work investigated the detection of passive data leakage, but there is no systematic search with a dynamic test strategy. Furthermore, no detailed case studies were listed that include details about misconfigurations, impact, usage and mitigation.

Gamba et al. [10] conducted a security study of 2.2 million apps to examine the use of custom permissions in Android and their impact on security and privacy. Using two static analysis tools, they identified 1,209 apps with custom permissions that were defined with a normal protection level. In the case of one app, read and write access to a content provider could be carried out successfully. A systematic search was made for custom permissions that enable data leakage.

In this context, the effects and mitigation were also discussed. However, specific configuration details and usage data were not provided.

In our work, we carry out an automated search and analysis of misconfigurations and passive data leakage from content providers. With our systematic filtering approach and dynamic exploitability testing, we are able to effectively search for vulnerabilities in a large data collection of real apps. Our case studies examine the underlying mistake in the configuration, the impact on the user, the usage by other apps and possible mitigation, linked together and related to the respective app. To our knowledge, this is the first study that includes all of these aspects in the analysis.

7.2 Drozer: A Commercial Vulnerability Analyzer

We compared our tool `ProseccoFlow` with `Drozer`[6], a tool for dynamic analysis on Android, which supports checking the security of content providers. `Drozer` uses a `Client` including a console to submit commands to the `Agent`, an app installed on the Android device. We implemented a semi-automatic test run with `Drozer` replacing the candidate search and the dynamic validation of `ProseccoFlow`. Since the `Agent` in its classic version requests no custom permissions of apps under test, the analysis is only successful for content providers without permissions. We added the required permissions to a custom version of the `Agent` to enable the path search and query functions of `Drozer` for content providers with custom permissions. While `ProseccoFlow` searches for several functions that can contain content URIs, `Drozer` only uses path permissions in the app's manifest and complete URIs starting with `content://` from the binary to check the app under test for accessible content providers. With this strategy, `Drozer` can provide the complete content URIs for 29 of the 358 candidate apps of our study. This includes one app, where `ProseccoFlow` was not able to extract the correct paths. Finally, we used the query function of `Drozer`, which also implements a content resolver like the `ProseccoApp`, to dynamically validate the apps. We added our collection of possible paths and were able to save the provided data of 254 apps. This confirms our results for the 253 candidates successfully validated in both runs. The test shows that the strategy of `Drozer` is not suitable for checking content providers with permissions and extracting paths from all possibilities of declaring content URIs.

7.3 Android Manifest Misconfigurations

Incorrect configuration entries in the Android manifest enable many vulnerabilities within the Android system, especially in inter-app communication. Research work has therefore also dealt with the detection of misconfigurations and their consequences for the Android system.

Han et al. [19] presented an approach to analyze vulnerabilities triggered by misconfigurations in the Android manifest, which extracts information based on

[6] https://labs.reversec.com/tools/drozer

static analysis and applies it using logic programming to uncover vulnerabilities. Jha et al. [21] systematically searched for developer errors in manifests of 13,483 Android applications and were able to identify 59,547 configuration errors. Yang et al. [30] used natural language processing to generate a scheme based on Android documentation constraints and to check the manifest files of around 2.5 million Android apps against it. In the process, over 850,000 misconfigured apps were identified. Zhang et al. [31] made a systematic study on custom permissions and the mistakes of developers at implementing them. They used bug reports and forum questions to identify nine groups of issues. The authors developed a static analysis tool to detect them in the manifest files of 83,085 Android apps and were able to find developer mistakes in 21,664 apps.

8 Conclusion

We have developed an automated analysis to specifically search for configuration parameters within Android applications that allow access to sensitive data from content providers. We applied our method to a dataset of 321,340 apps. Through the strategic filter approach, we were able to carry out a detailed study with 380 current applications of different categories, which shows that errors in the configuration of components for data sharing are possible. In detailed case studies we proved the successful exploitation of the vulnerability that can have a significant impact on the privacy and the everyday life of Android users.

References

1. Ali-Gombe, A.I., III, G.G.R., Ahmed, I., Roussev, V.: Don't touch that column: Portable, fine-grained access control for Android's native content providers. In: Proc. ACM Conf. Security & Privacy in Wireless and Mobile Networks (WISEC). pp. 79–90. ACM (2016). https://doi.org/10.1145/2939918.2939927
2. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: collecting millions of Android apps for the research community. In: Proc. 13th Int. Conf. Mining Software Repositories (MSR). pp. 468–471. ACM (2016). https://doi.org/10.1145/2901739.2903508
3. Arzt, S., Bodden, E.: Stubdroid: automatic inference of precise data-flow summaries for the Android framework. In: Proc. 38th Int. Conf. Software Engineering (ICSE). pp. 725–735. ACM (2016). https://doi.org/10.1145/2884781.2884816
4. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.D.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: ACM SIGPLAN Conf. Programming Language Design and Impl. (PLDI). pp. 259–269. ACM (2014). https://doi.org/10.1145/2666356.2594299
5. Bagheri, H., Sadeghi, A., Garcia, J., Malek, S.: COVERT: compositional analysis of Android inter-app permission leakage. IEEE Trans. Software Eng. **41**(9), 866–886 (2015). https://doi.org/10.1109/TSE.2015.2419611
6. Bosu, A., Liu, F., Yao, D.D., Wang, G.: Collusive data leak and more: Large-scale threat analysis of inter-app communications. In: Proc. ACM Asia Conf. Computer

and Communications Security (AsiaCCS). pp. 71–85. ACM (2017). https://doi.org/10.1145/3052973.3053004
7. Bugiel, S., Heuser, S., Sadeghi, A.: Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In: King, S.T. (ed.) Proc. USENIX Security Symp. pp. 131–146. USENIX Association (2013)
8. Chang, K.C., Zaeem, R.N., Barber, K.S.: Is your phone you? How privacy policies of mobile apps allow the use of your personally identifiable information. In: IEEE Int. Conf. Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA). pp. 256–262. IEEE (2020). https://doi.org/10.1109/TPS-ISA50397.2020.00041
9. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.A.: Analyzing inter-application communication in Android. In: Proc. Int. Conf. Mobile Systems, Applications, and Services (MobiSys). pp. 239–252. ACM (2011). https://doi.org/10.1145/1999995.2000018
10. Gamba, J., Feal, Á., Blázquez, E., Bandara, V., Razaghpanah, A., Tapiador, J., Vallina-Rodriguez, N.: Mules and permission laundering in Android: Dissecting custom permissions in the wild. IEEE Trans. Dependable Sec. Comput. **21**(4), 1801–1816 (2023). https://doi.org/10.1109/TDSC.2023.3288981
11. Google: Android 6.0 changes – Android Developers (2025), https://developer.android.com/about/versions/marshmallow/android-6.0-changes, accessed: April 9, 2025
12. Google: Application sandbox – Android open source project (2025), https://source.android.com/docs/security/app-sandbox, accessed: April 9, 2025
13. Google: Content provider basics – app data and files – Android Developers (2025), https://developer.android.com/guide/topics/providers/content-provider-basics, accessed: April 9, 2025
14. Google: Content providers – app data and files – Android Developers (2025), https://developer.android.com/guide/topics/providers/content-providers, accessed: April 9, 2025
15. Google: Declare your app's data use – privacy – Android Developers (2025), https://developer.android.com/guide/topics/data/collect-share, accessed: April 9, 2025
16. Google: Define a custom app permission – privacy – Android Developers (2025), https://developer.android.com/guide/topics/permissions/defining, accessed: April 9, 2025
17. Google: Permissions on Android – privacy – Android Developers (2025), https://developer.android.com/guide/topics/permissions/overview, accessed: April 9, 2025
18. Google: Security checklist – Android Developers (2025), https://developer.android.com/privacy-and-security/security-tips, accessed: April 9, 2025
19. Han, Z., Cheng, L., Zhang, Y., Zeng, S., Deng, Y., Sun, X.: Systematic analysis and detection of misconfiguration vulnerabilities in Android smartphones. In: Proc. IEEE Int. Conf. Trust, Security and Privacy in Computing and Communications (TrustCom). pp. 432–439. IEEE Computer Society (2014). https://doi.org/10.1109/TRUSTCOM.2014.56
20. Hassanshahi, B., Yap, R.H.C.: Android database attacks revisited. In: Proc. ACM Asia Conf. Computer and Communications Security (AsiaCCS). pp. 625–639. ACM (2017). https://doi.org/10.1145/3052973.3052994
21. Jha, A.K., Lee, S., Lee, W.J.: Developer mistakes in writing Android manifests: an empirical study of configuration errors. In: Proc. Int. Conf. Mining Software Repositories (MSR). pp. 25–36. IEEE Computer Society (2017). https://doi.org/10.1109/MSR.2017.41

22. Johnson, R., Elsabagh, M., Stavrou, A., Offutt, J.: Dazed droids: A longitudinal study of Android inter-app vulnerabilities. In: Proc. ACM Asia Conf. Computer and Communications Security (AsiaCCS). pp. 777–791. ACM (2018). https://doi.org/10.1145/3196494.3196549
23. Kober, M., Samhi, J., Arzt, S., Bissyandé, T.F., Klein, J.: Sensitive and personal data: What exactly are you talking about? In: Proc. IEEE/ACM Int. Conf. Mobile Software Engineering and Systems (MOBILESoft). pp. 70–74. IEEE (2023). https://doi.org/10.1109/MOBILSOFT59058.2023.00016
24. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Traon, Y.L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.D.: IccTA: Detecting inter-component privacy leaks in Android apps. In: Bertolino, A., Canfora, G., Elbaum, S.G. (eds.) Proc. IEEE/ACM Int. Conf. Software Engineering (ICSE). pp. 280–291. IEEE Computer Society (2015). https://doi.org/10.1109/ICSE.2015.48
25. Mutti, S., Bacis, E., Paraboschi, S.: SeSQLite: Security enhanced SQLite: Mandatory access control for Android databases. In: Proc. Annu. Computer Security Applications Conf. (ACSAC). pp. 411–420. ACM (2015). https://doi.org/10.1145/2818000.2818041
26. Samhi, J., Bartel, A., Bissyandé, T.F., Klein, J.: RAICC: revealing atypical inter-component communication in Android apps. In: Proc. IEEE/ACM Int. Conf. Software Engineering (ICSE). pp. 1398–1409. IEEE (2021). https://doi.org/10.1109/ICSE43902.2021.00126
27. Shahriar, H., Haddad, H.M.: Content provider leakage vulnerability detection in Android applications. In: Poet, R., Rajarajan, M. (eds.) Proc. Int. Conf. Security of Information and Networks (SIN). p. 359. ACM (2014). https://doi.org/10.1145/2659651.2659716
28. StatCounter: Mobile OS market share worldwide 2009-2025 – Statista (2025), https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/, accessed: April 9, 2025
29. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot – a Java bytecode optimization framework. In: Proc. Conf. Centre for Advanced Studies on Collaborative Research (CASCON). p. 13 (1999). https://doi.org/10.1145/781995.782008
30. Yang, Y., Elsabagh, M., Zuo, C., Johnson, R., Stavrou, A., Lin, Z.: Detecting and measuring misconfigured manifests in Android apps. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS). pp. 3063–3077. ACM (2022). https://doi.org/10.1145/3548606.3560607
31. Zhang, X., Yu, Z., Li, X., Zhang, C., Sun, C., Zhang, N., Deng, R.H.: Understanding the bad development practices of Android custom permissions in the wild. IEEE Trans. Dependable Sec. Comput. **22**(4), 3208–3223 (2025). https://doi.org/10.1109/TDSC.2024.3525049
32. Zhou, Y., Jiang, X.: Detecting passive content leaks and pollution in Android applications. In: Proc. Annu. Network and Distributed System Security Symp. (NDSS). The Internet Society (2013)