Component Security Ten Years Later: An Empirical Study of Cross-Layer Threats in Real-World Mobile Applications

KEKE LIAN, Fudan University, China
LEI ZHANG, Fudan University, China
GUANGLIANG YANG, Fudan University, China
SHUO MAO, Fudan University, China
XINJIE WANG, Fudan University, China
YUAN ZHANG, Fudan University, China
MIN YANG, Fudan University, China

Nowadays, mobile apps have greatly facilitated our daily work and lives. They are often designed to work closely and interact with each other through app components for data and functionality sharing. The security of app components has been extensively studied and various component attacks have been proposed. Meanwhile, Android system vendors and app developers have introduced a series of defense measures to mitigate these security threats. However, we have discovered that as apps evolve and develop, existing app component defenses have become inadequate to address the emerging security requirements. This latency in adaptation has given rise to the feasibility of *cross-layer exploitation*, where attackers can indirectly manipulate app internal functionalities by polluting their dependent data. To assess the security risks of cross-layer exploitation in real-world apps, we design and implement a novel vulnerability analysis approach, called `CLDroid`, which addresses two non-trivial challenges. Our experiments revealed that 1,215 (8.8%) popular apps are potentially vulnerable to cross-layer exploitation, with a total of more than 18 billion installs. We verified that through cross-layer exploitation, an unprivileged app could achieve various severe security consequences, such as arbitrary code execution, click hijacking, content spoofing, and persistent DoS. We ethically reported verified vulnerabilities to the developers, who acknowledged and rewarded us with bug bounties. As a result, 56 CVE IDs have been assigned, with 22 of them rated as 'critical' or 'high' severity.

CCS Concepts: • **Security and privacy → Domain-specific security and privacy architectures**; **Software security engineering**.

Additional Key Words and Phrases: Android, APP Component, Cross-Layer Threat

ACM Reference Format:
Keke Lian, Lei Zhang, Guangliang Yang, Shuo Mao, Xinjie Wang, Yuan Zhang, and Min Yang. 2024. Component Security Ten Years Later: An Empirical Study of Cross-Layer Threats in Real-World Mobile Applications. *Proc. ACM Softw. Eng.* 1, FSE, Article 4 (July 2024), 22 pages. https://doi.org/10.1145/3643730

1 INTRODUCTION

Mobile applications (apps) have become an integral part of modern daily life, offering users a broad spectrum of services and functionalities. They are typically crafted to interact and collaborate with

Authors' addresses: Keke Lian, Fudan University, Shanghai, China, kklian20@fudan.edu.cn; Lei Zhang, Fudan University, Shanghai, China, zxl@fudan.edu.cn; Guangliang Yang, Fudan University, Shanghai, China, yanggl@fudan.edu.cn; Shuo Mao, Fudan University, Shanghai, China, smao20@fudan.edu.cn; Xinjie Wang, Fudan University, Shanghai, China, xinjiewang21@fudan.edu.cn; Yuan Zhang, Fudan University, Shanghai, China, yuanxzhang@fudan.edu.cn; Min Yang, Fudan University, Shanghai, China, m_yang@fudan.edu.cn.

ACM 2994-970X/2024/7-ART4
https://doi.org/10.1145/3643730

one another, creating a dynamic and integrated ecosystem. As the foundational infrastructure of inter-app communication, app components are required to be exposed and play a pivotal role. Over a prolonged period, the security of app components has garnered significant attention. Extensive security research [16, 19, 22, 25, 34, 36, 37] has identified vulnerabilities in app components that can lead to various *component hijacking attacks*. Specifically, app attackers can exploit exported components to execute critical operations (e.g., privileged system APIs) within the apps, and cause serious security consequences, including permission redelegation and leakage [18, 25], intent spoofing [7], and cross-app scripting [23].

After knowing these security vulnerabilities, app developers and system vendors have made diligent efforts to enhance the security of app components. They have proposed a series of code-layer measures to prevent the abuse of sensitive operations by external apps. For example, the Android security team has continuously improved and restricted access to app components since Android 4 (2011). Also, app developers deployed strict security checks (e.g. input validation and permission checks) against external requests in their app components. Certainly, these defensive protections prove efficacious in mitigating conventional component hijacking attacks. Nevertheless, with the expansion of app size and the proliferation of app functionalities and services, app development practices are also undergoing evolution. Consequently, a critical question arises: *Do existing defensive mechanisms still retain their robustness against component security threats?*

We first investigate modern app architectures and uncover an exploitation method that current defenses struggle to address. Specifically, following official recommendations [14], modern apps are progressively embracing a persistent data layer design for robustness and maintainability. This data layer employs data pools (e.g., databases, shared preferences, and files) to centrally manage data from various components. As a result, data with varying sensitivities may be stored within the same data pool, accessible by different components, including exported components and isolated internal ones. If the internal components retrieve data from the data pool for security-critical purposes, there may be a risk of app functionality abuse. As illustrated in Figure 1, app attackers can corrupt some app internal data through an exported component to indirectly manipulate the internal functionalities that use these polluted data, referred to as *cross-layer exploitation* in this paper. The data items within the data pool require different levels of protection, contingent upon their app-specific purposes, to strike a balance between usability and security. Nevertheless, existing app component safeguards encounter difficulties in addressing the fine-grained and diverse protection demands (detailed in §2.1).

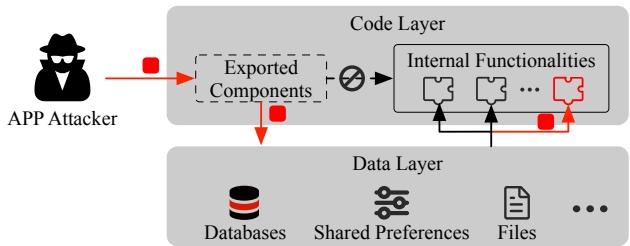

Fig. 1. APP Cross-Layer Exploitation.

Then we study the practicality of cross-layer exploitation by assessing the security risks in real-world Android apps. While numerous tools [26, 28, 30, 34, 35, 50, 51, 55, 58, 59] have been proposed for detecting app component vulnerabilities, they do not adequately consider the data flows crossing the code and data layers, making them hardly applied to cross-layer threat detection. Technically, there are two major challenges that should be carefully dealt with. (i) Fine-grained

data item-level flow tracking. A data pool often stores a multitude of data items but not all of them can be corrupted by external apps. Even worse, all these data items are accessed through a unified set of data pool access APIs, making it challenging to correlate a data injection point with its corresponding readout points. Treating the data pool as a black box without tracing the data item flow will cause unacceptably high false positives. (ii) App-specific critical data usage scenario identifying. The corrupted data can be used in various app internal functionalities, but it is uncertain which of these functionalities can be exploited to cause security hazards. In contrast to traditional component attacks that often target clearly defined privileged system APIs, the sensitivity of app internal functionalities depends on their app-specific business logic implementation, lacking a universal indicator. To address the above challenges, we design and implement a novel app cross-layer threat detection approach, named `CLDroid`. By applying `CLDroid` on 13,824 popular apps collected from Google Play, we find 1,714 (12.4%) apps have opened data sharing channels that result in the sharing of over 10,000 data pools and more than 200,000 data items. `CLDroid` assesses their security and identifies 1,215 (8.8%) apps as potentially vulnerable to cross-layer exploitation with a total of more than 18 billion installs.

Furthermore, we delve deeper into understanding the severe security hazards that cross-layer exploitation can cause in real-world scenarios. Specifically, we randomly select 60 vulnerable apps for manual verification and confirm that at least 32 of them can be successfully exploited. By exploiting these vulnerabilities, an unprivileged app can achieve various attack consequences, including arbitrary code execution, click hijacking, UI spoofing, and persistent DoS.[1] We have responsibly disclosed the verified vulnerabilities to respective developers and received several confirmations, along with bug bounties awarded by Alibaba and Tencent. A total of 56 CVE IDs have been assigned for these vulnerabilities, with 22 of them rated as 'critical' and 'high' severity.

Contributions. The contributions of our work are summarized below.

- We revisited Android app component security in the context of modern app architectures and discovered that state-of-the-art defenses have not caught up with the fast app development. Through a form of attack referred to as cross-layer exploitation, attackers can indirectly manipulate app internal functionalities by polluting their dependent app data.
- We studied the security risks of cross-layer exploitation in real-world apps. To achieve this, we design and implement a novel vulnerability detection tool named `CLDroid` which addresses two non-trivial challenges. Our experiments revealed that 1,215 (8.8%) popular apps may suffer from cross-layer exploitation, with a total of more than 18 billion installs.
- We verified that an unprivileged app attacker could achieve significant security consequences through cross-layer exploitation, including content spoofing, privilege escalation, and persistent DoS attacks. We responsibly disclosed the verified vulnerabilities to respective developers, who confirmed and rewarded bug bounties to us, with 56 CVE IDs assigned.

2 UNDERSTANDING APP CROSS-LAYER EXPLOITATION

In this section, we first analyze why cross-layer exploitation is practical against existing defenses. Then we introduce their security implications and threat model. Finally, we present a real-world example found by us to show how exploitation occurs and its security consequences.

2.1 Root Cause Analysis

Modern apps are becoming increasingly complex and larger in size, resulting in a wealth of data in apps. To enhance app robustness and maintainability, there is a growing trend towards coupling business functionality with app data. In other words, the management of app data is shifting from

[1]All attack consequences have been manually verified with PoCs.

decentralized handling across various components towards centralized management. Specifically, modern apps employ a persistent data layer including various data pools to manage data originating from different components within the app. However, there is no common guideline or best practices on how to manage the diverse app-specific data. Consequently, data with various purposes may be mixed and stored in the same data pool. When a need arises to share some of these data with external apps, it can potentially lead to excessive exposure of data used for other sensitive functionalities, thereby creating the potential for cross-layer exploitation.

To prevent app functionalities abuse through exported components, a series of app component safeguards have been proposed and implemented over the years [4, 5, 12, 15, 18, 28, 44, 55]. Due to prior research predominantly focusing on component attacks limited to code-layer exploitation, existing defenses primarily concentrate on addressing security threats at the code-layer level. The core principle of these defenses is to ensure that the privileged operations in apps can only be invoked by authorized requesters with legitimate inputs. Specifically, app permission and identity-based checks can help restrict functionalities of varying sensitivities to be accessible only by requesters that meet corresponding identity conditions. The Android system has categorized the sensitivity of system operations that apps can invoke and allows apps to define customized protections based on their own implementation. Furthermore, for authorized requesters, developers can enforce input validations to prevent privileged operations from being executed with untrusted inputs. For instance, Android provides parameterized database access APIs [15] to prevent external malicious inputs from being interpreted as executable SQL code, effectively mitigating the risk of SQL injection.

As shown in Table 1, while these defenses can effectively mitigate traditional component attacks at the code layer, such as capability leak and code injection, they are inadequate against cross-layer exploitation. On one hand, defenses around requesters are enforced at the component level, which is too coarse-grained to protect the diverse app-specific data. In particular, a singular component can access multiple data pools, maintaining an extensive collection of data items. These data items serve various functionalities and possess varying degrees of sensitivity, thus warranting different levels of protection. Component-level access control is insufficient to address the fine-grained data protection requirements, potentially resulting in unintended data exposure. On the other hand, input validations are also limited in their efficacy. Techniquely, they are designed to prevent critical functions from being executed in unexpected ways due to illegitimate external inputs. However, cross-layer exploitation injects poisonous data into data pools leveraging the intended app functionalities in the expected manner. Hence, the request inputs appear legitimate and lack identifiable problematic patterns. Furthermore, when the poisoned data is retrieved from the internal data pools and used, it loses its external input identity and is automatically trusted due to the inherent trust in the isolation provided by the underlying Android system.

Table 1. Existing defenses against app component attacks.

Defenses	Target	Code-Layer Exploitation				Cross-Layer Exploitation
		Capability Leak	Intent Proxy	Code Injection	Buffer Overflow	
APP Permissions	Requester	✔	✔	✘	✘	✘
APP Identity Whitelist	Requester	✔	✔	✘	✘	✘
Input Validation	Input	✘	✘	✔	✔	✘

2.2 Security Implications

In this subsection, we analyze the security implications when poisonous data enters the victim app's data layer. Broadly speaking, cross-layer exploitation shares similarities with second-order

web attacks [9, 42, 52, 53] and code reuse attacks [10, 49]. Therefore, we scrutinize two attack vectors: data loading and consuming.

2.2.1 Data Loading-Based Attack. The memory hierarchy in computer architecture implies that the size of local storage is typically greater than the size of memory. Thus, apps should be careful when loading data from local storage to memory. Specifically, apps should avoid loading vast amounts of data into memory simultaneously.

However, our analysis indicates that while loading local data to memory, apps usually do not perform due diligence to verify the data size. The reason for such oversight is a belief by developers that data loaded from the app's internal storage is controlled by themselves and always falls within expected data intervals. Thus, if an attacker injects sufficient data into the data pool of an app, an Out-of-Memory (OOM) error or Application-Not-Responding (ANR) error will be triggered when the app loads these injected data. (See more details in §4.2.)

2.2.2 Data Consuming-Based Attack. In general, apps can contain numerous strictly-protected critical functionalities. Cross-layer exploitation allows attackers to indirectly abuse these sensitive functionalities by tampering with their dependent data. Specifically, we find two ways to achieve the goals: 1) function parameter manipulation and 2) execution switch.

For the first scenario of function parameter manipulation, the poisoned data may flow to a critical API as its parameters. In such a case, an attacker can control the API's execution by feeding it malicious inputs. For example, a popular weather app (*.weather, 1,000,000+ downloads) reads URL data from its database and renders the content inside the app through WebView. By manipulating the URL parameter of *WebView.loadURL()*, the attacker can load arbitrary malicious content within the victim app's embedded web browser for phishing. Even worse, the attacker can further stealthily obtain private user data, such as geographical location, by calling the corresponding JavaScript interfaces customized and supported by the WebView component.

For the other attack scenario of execution switch, the poisoned data can be used as a function controller, determining whether a critical API is enabled or disabled for execution. This is because there exist control-flow dependencies between the poisoned data and critical functions. By tampering with these data items, attackers can disable some key features of the app. For example, to ensure the user experience of new users, an app (*.security, 10,000,000+ downloads) only starts advertising to users after a certain time of app installation. This depends on a time interval saved in a shared preference file. By exploiting the developer-configured time interval, the app's ads-pushing functionality can be disabled permanently, resulting in the app developer's economic loss.

2.3 Threat Model

Similar with existing security work [1, 3, 16, 19, 22, 25, 34, 57, 60], we consider the adversary is a mobile app attacker, whose goal is to attack benign apps on the victim's device. One important assumption in our threat model is that the attacker is unprivileged and does not require any sensitive permissions. Regularly, all mobile apps are sandboxed and isolated from each other. An app attacker is not allowed to directly touch the isolated data pools in other apps. Instead, it can send crafted inter-process communication (IPC) messages to the 'exported' (i.e., callable by other apps) components in the victim app. As shown in previous studies, the attacker can send local messages through controlled apps [25] or remote messages via WebView or third-party ads [34].

2.4 Real-World Example

Action Launcher (*.playstore) is a high-profile launcher app with more than 10 million downloads on Google Play and is ranked as one of the "Best Android launchers of 2022" [8]. This app helps users manage home screen (desktop) and apps, e.g., opening the corresponding app when an app

icon is clicked. We find this popular app suffers cross-layer exploitation, causing serious security hazards. We verify that by exploiting these vulnerabilities, an attacker can perform persistent device freezing (DoS) and UI spoofing attacks. We have reported these vulnerabilities to the developers, which have been confirmed and assigned with two CVE IDs.

Figure 2 illustrates its simplified code implementation. There are three important components (e.g., C_e, C_{s1}, and C_{s2}) in the victim app, which are linked together by a persistent data layer. The entry component C_e is designed for app-to-app communication. When receiving data in runtime, it saves poisonous data into a database, through the data-accessing API *insert()* at Line 11. Then, the sensitive component C_{s1} blindly trusts all data saved in the database and reads the poisonous data from the database through another data-accessing API *query()* at Line 20. It loads all poisonous data in memory (binding to app widgets) for further use, i.e., calling the function *bindBubbleViews()*, and starting another sensitive component C_{s2}. The attacker can exploit this victim app from two attack vectors. To exploit the data-loading attack vector, the attacker can inject excessive poisonous data into the database. We find that the victim app loads and stores all poisonous data in the memory to prepare the desktop, thus causing overflow and crashes. Since the launcher app is the first app to be started (similar to 'desktop'), this attack persistently freezes the victim device. For the data-consuming attack vector, the attacker can manipulate the intent string corresponding to a certain app (e.g., Facebook) in the database through another access API *update()*. When the app icon is clicked, the fake intent is consumed and the app-launching process is hijacked by starting a fake activity (*startActivity()* at Line 33 in C_{s2}), resulting in UI spoofing.

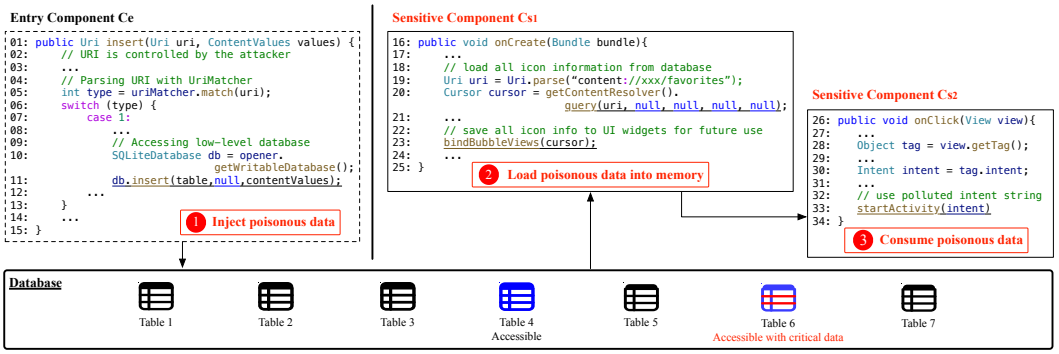

Fig. 2. Simplified code of a real-world case, which can be exploited for device freezing and UI Spoofing

3 CLDROID DESIGN

To assess the real-world security risks of cross-layer exploitation, we design an automated and effective security-vetting approach for Android apps. The high-level idea is to track the injected data items and figure out whether they are used in some security-critical scenarios. However, as discussed in § 1, two major challenges should be carefully dealt with.

1) Fine-grained Data Pool Aware Analysis: In practice, various different types of data items are often mixedly saved together in a data pool. As illustrated by the example shown in Figure 2, there are seven tables in the local database. However, not all of them are exploitable for external apps. Only two of them can be potentially touched by app attackers. Within these two insecure tables, only seven columns out of 28 columns correspond to critical internal functionalities, e.g., determining which app to be launched. Furthermore, these data items are managed and accessed through the same set of database APIs, e.g., *insert()* (Line 11) and *query()* (Line 20).

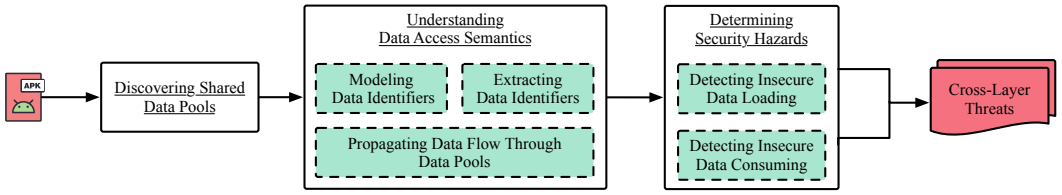

Fig. 3. The overall architecture of CLDroid.

Data flow analysis is suitable in such an analysis scenario. However, existing techniques are hardly applied or extended to handle the crucial data flow through data pools. For dynamic techniques [17, 38, 54, 56], they suffered from coverage issues, which may cause false negatives, i.e., missing some data access behaviors. For static data flow-based techniques, prior approaches [3, 21, 32, 57] did not consider data pools. Treating them as 'black boxes' and directly applying prior approaches on the entire data pool will cause high false positives.

2) Universal Security Hazard Vetting Approach: The security hazards caused by insecure app-specific data items are hardly determined, which stems from the fact that there is no documentation to tell us which data-use scenarios are security-critical. These data items are designed for customized app-specific functionalities, whose purpose and sensitivity depend on the app's business logic. It is inherently challenging to automate the understanding of an app's business logic, thus making it difficult to assess the potential security risks posed by tampering with these data items. In addition, these functionalities are typically internal and inaccessible to external apps, thus lacking indicators such as permission checks. Worse still, unlike data with explicit purposes, e.g., user privacy data, these app-specific data items lack sufficient semantic information to infer their sensitivity.

Our Solution: To address the above challenges, we propose a novel end-to-end vulnerability detection approach, called CLDroid, against app cross-layer threats. Figure 3 illustrates the overall architecture of CLDroid, which comprises three stages. Given a mobile app, CLDroid first identifies the data pools that may be injected by external apps through exported components. Second, CLDroid employs data identifier-based analysis to track the data flow of data items that traverse through the target data pool. Third, CLDroid learns app-specific data use semantics and universally assesses their security risks (from the perspectives of two attack vectors). Below we present more details for each stage.

3.1 Discovering Shared Data Pools

In this step, CLDroid locates data pools that are potentially used in app-to-app data sharing and can be injected by third-party apps, which are susceptible to cross-layer exploitation.

To gain insights into the utilization of data pools in real-world scenarios, we first conduct an empirical study on the 300 most popular Android apps from Google Play. We observe that apps usually create data pools and prepare initial data after their first startup but before users interact with the core functionalities. Thus, we install and run each app on a Oneplus 9 (Android 12) with Monkey [13] for two minutes. Then, we extract data pools from their private storage space (e.g., *Context.getFilesDir()*). As a result, we successfully obtained 16,348 unique files with 218 distinct types of file name extensions. Table 2 shows more details of the data pool type distribution.

Then, according to the study results, we conclude related instructions (i.e., APIs) for accessing data pools, e.g. opening data pool and reading/writing data items, based on the Android developer documentation. These APIs are further used for detecting shared data pools for which attack payloads can be injected. Here we focus on the official Android built-in APIs. As summarized by Table 3, we totally collected 244 APIs, including 55 opening, 107 reading, and 82 writing APIs.

Table 2. The distribution of data pools extracted from 300 popular apps.

File Type	# Unique File	Examples	File Type	# Unique File	Examples
SharedPreference	2186	.xml	Layout & Format	441	.html, .css
Picture	1742	.png, .webp	Code	305	.js, .so, dex
Database	1091	.db, .sqlite	Others	2993	.tmp, .crc, .pb
JSON	836	.json	***Total***	16,348	–

Table 3. Summary of 244 data pool access APIs.

Data Pool	# Open/Read/Write APIs	Examples
Shared Preference	4/7/6	SharedPreferences.getString(), SharedPreferences$Editor.putString()
Database	12/14/12	SQLiteDatabase.query(), SQLiteDatabase.insert()
JSON	31/60/35	JSONObject.getString(), JSONObject.put()
Others	39/26/29	FileReader.read(), FileWriter.write()

Upon the above API list, CLDroid applies static program analysis on the target app to identify the potentially injectable data pools. Specifically, CLDroid first parses the *AndroidManifest* file to learn the app-to-app communication channels, i.e., exported components, which are responsible for managing and processing the data injection process. Then, CLDroid models the life cycle of found components and sets up entry points for our further analysis. After that, we apply control flow and data flow analysis starting from the entry points. The control flow analysis is helpful to check if there is a path from an entry point to the data pool access API. The data flow analysis can provide information and tips for which data items may be shared. During analysis, CLDroid checks its required access permissions. Only the components without permission protection or just protected by normal-level and undefined permissions are picked by CLDroid. Finally, CLDroid obtains the potentially shared data pools and the data flow information related to the internal data items, which is further analyzed in the next stage.

3.2 Understanding Data Access Semantics

After discovering shared data pools, we aim to figure out the data flow paths traversing the data pools. It is essential to 1) learn data access semantics, especially data pool reading and writing behaviors, and 2) infer the internal organization of data pools and track each data item. Therefore, we model data pool access and conduct a data item-level analysis. Specifically, CLDroid first searches for all pieces of code containing data pool reading and writing operations, and precisely analyzes what data items are being accessed by each code unit. Then, CLDroid can link the corresponding data reading and writing code units that operate on the same data item. In this way, CLDroid facilitates the propagation of data flows traversing the data pools.

3.2.1 Modeling Data Identifiers. We model the access operation of data pools based on how related APIs are used in practice. When a data item is accessed through a unified access API (e.g., *ContentResolver.insert()*), the API needs to first learn the data pool to be accessed (e.g., database), and then check the detailed position (e.g., table and column information) where the data item is saved in the target data pool. The detailed position information where data is saved plays an important role in data operations. Thus, we can use the position information, i.e., data identifier, to model data item access. We define the data identifier as follows:

```
Data Pool Identifier (DPI) = <Type, URI | Name>
Data Item Identifier (DII) = <DPI, IP>
```

```
Type = DataBase | Shared Preference | JSON | Regular File | ...
URI  = String, Name = String

Internal position (IP) in data pool:
Type = DataBase   =>  IP = <Table, Column>,    Type = Shard Preference =>  IP = <Key, Value>
Type = JSON       =>  IP = <Key, Value>,       Type = Regular File     =>  IP = <Raw Data>
```

Around the above definition, we model data pool operations. More details are presented below.

DataBase Identifier. The identifier of a database data item can be abstracted as [<URI, Name>, <Table, Column>]. For database operations, e.g., 'query' and 'insert', their workflows are quite similar. As Figure 2-❶ shows, when a database API '*insert*' is called, it first parses the URI content with a parser '*UriMatcher*', a hashmap-like collection of resource information. Then, the resource *type* is retrieved at Line 5 and dispatched to the essential corresponding operation (Line 11). In this line, there is an important dependency chain: *uri* → *type* → *opener* → *db*.

To discover such a URI dependency relationship and the URI value space, `CLDroid` applies data flow analysis to track the URI object. When reaching a parser, `CLDroid` computes the inside of the parser to gather URI values. There are usually different cases for the parser. For the first case, URI is parsed by a prepared 'UriMatcher' object via *UriMatcher.match(Uri)*, and then the operation request is dispatched based on the parsing results. `CLDroid` obtains the URI value by backwardly tracking the UriMatcher initialization. The UriMatcher object stores various URIs with corresponding types as key-value pairs, and its *match()* method returns the stored type of the matched URI. The relationship between the URI (key) and type (value) is typically registered during the initialization of UriMatcher object through *UriMatcher.addURI(authority, path, code)*, which constructs the URI by concatenating the authority and path. Thus, `CLDroid` obtains the URI string by analyzing the authority and path parameters through string value analysis (detailed in § 3.2.2). The second situation is that concrete values in the URI are directly parsed to determine the resource to operate, e.g., the target table. Specially, the target app directly reads parameters from the URI class, e.g., path segment, and uses it as the table name to be accessed. In this case, based on the parsing process, `CLDroid` constructs the final URL string value by linking the obtained information together, e.g., content://@authority/@tableName. Note that the authority information is resolved from AndroidManifest and the possible table names can be retrieved later.

To obtain the database name, `CLDroid` continues data flow analysis and reaches the dispatched code (Line 9-11), which calls the database operating instruction, e.g., *db.insert()*. `CLDroid` backwardly tracks the essential initialization of the caller object *db* and searches the database *opener*, which may be an instance of *DBOpener*, a child class of *SQLiteOpenHelper*. After that, `CLDroid` continues backwardly pinpointing the constructor functions of *DBOpener*, where the name of the target database may be defined. We also find some cases that directly initialize the database name as the value of a field. Thus, `CLDroid` also checks the initialization of all fields.

Furthermore, for obtaining the table and column information, `CLDroid` can directly analyze database reading and writing APIs. For instance, the first parameter of *db.insert(table,null,ContentValues)* (Line 11) is the table name to be accessed, while the third parameter contains the columns to write. Typically, the columns are determined by requesters. `CLDroid` needs to extract all potential values. Specifically, `CLDroid` analyzes the table creation instructions and parses the SQL statements, which are usually defined in the constructors of *DBOpenner* objects.

Besides, some apps do not directly provide SQL wrapper functions. Instead, all database access requests are handled by directly calling the low-level function *SQLiteDatabase.execSQL(String sql)*. For these cases, `CLDroid` directly parses the SQL request string to understand the table and column information through regular expressions.

SharedPreference Identifier. A shared preference file is used to save primitive data in key-value pairs. For a data item (i.e., <key, value>), the key field provides enough information for its position. The following code snippet shows a classic example to access the shared preference file:

```
// Opening file
SharedPreferences sp = Context.getSharedPreferences(filename,mode);
SharedPreferences$Editor editor = sp.edit();
// Reading data
String value = sp.getString(key);
// Writing data
editor.putString(key,value);
```

To obtain the file name, `CLDroid` locates the open functions of shared preference files, e.g., *getSharedPreferences(filename,mode)*, and directly analyzes the file name parameter via string value analysis. Besides, there exist some special interfaces without parameters, e.g., *getDefaultSharedPreferences()*. The name of the opened file depends on the app context, i.e., the app package name. `CLDroid` computes its value with the help of information from AndrodManifest.

To obtain the key value, `CLDroid` directly analyzes the related reading and writing APIs. For instance, the first parameter of the reading APIs (e.g., *SharedPreferences.getString()*) and the writing APIs (e.g., *SharedPreferences$Editor.putString()*) is the key value of the data item to be accessed.

JSON Identifier. JSON file is designed to store structured data based on the JavaScript object syntax. Its internal structure is organized as (nested) key-value pairs.

To retrieve data from a JSON file, an app typically first opens the file and reads the content. Then, the app deserializes the read content to a JSONObject through JSON parsing APIs, such as *JSONObject.parse()*. For nested JSONObjects, the app can utilize a chain of key values to access underlying objects by recursively calling reading APIs, e.g., *JSONObject.getJSONObject()*. Last, the app accesses specific values of the underlying JSONObject via a key name, e.g., *JSONObject.getString()*. Note that in addition to system APIs, we collect several JSON libraries commonly used by Java programs, including org.json, Fastjson, GSON, and Jackson.

To obtain the file name, `CLDroid` starts analysis from the JSONObject serialization or deserialization APIs. Taking the deserialization API '*JSONObject.parse(string)*' for example, `CLDroid` backwardly tracks the *string* parameter to check its initialization, which is usually read from a *file* object. Then, `CLDroid` further analyzes the file initialization and obtains the file path by analyzing the parameter of file open instructions, e.g., *new File()*. For the key information, `CLDroid` forwardly tracks the deserialized JSONObject and records the access path (i.e., key chain) for each level of nested objects by analyzing the related reading APIs, e.g., *JSONObject.getJSONObject(key)*, until the access to some specific values, e.g., *JSONObject.put(key,value)*.

Unstructured Data Identifier. In addition to the above data pools with structured organization, there exist some data pools whose internal structures are hard to parse, e.g., text and media files. Android framework provides dedicated access APIs for them. In this case, `CLDroid` treats their raw data as a whole and connects their reading-and-write access based on the URI or filename.

3.2.2 Extracting Data Identifiers. As mentioned above, during the analysis of data identifiers, many parameters of data access APIs are not constants and are generated dynamically, which involves many string operations. It is hard to obtain them directly. To address this challenge, we apply a string value analysis to support the extraction of data item identifiers. Specifically, starting from a variable of our interest, e.g., *SQLiteDatabase.rawQuery(query)*, we backwardly traverse the instructions in the CFG. If there are any variables that contribute to the computation of the target variable (e.g., data flow dependency), we record the involved instructions and variables in a string computation stack. We keep iterating in the same way until the definitions of all dependent variables are found.

Then, we compute the final string value of our interest with the string computation stack. We execute the involved string operations in a forward simulation based on the string operation API summaries. For example, if the involved instruction is a string append API, we perform the string append operation. Note that the string-related APIs belong to system-defined classes, which are typically not obfuscated. The computed string values may still need to be further parsed to extract related data identifiers. For example, we parse the SQL query statement through regular expressions to extract the table and column information. It is worth noting that if some variables are entirely decided by the requester apps and cannot calculate exact values, we consider they can be any value. Finally, we parse the concrete values collected from a total of 244 APIs (Table 3).

3.2.3 Propagating Data Flow Through Data Pools. After understanding what data item (i.e., data identifier) is being tracked, CLDroid links the corresponding reading and writing code units together so as to track the data flow through data pools. Given the specific identifier of a data item, CLDroid first scans the code space to find the related reading and writing code units layer by layer. Take shared preference for example. Its data item identifier can be [<File Name>, <key>]. First, CLDroid locates all opening instructions for the specified data pool type. Then CLDroid extracts and compares the data pool identifiers, e.g., file name. For instructions with the same file name, CLDroid further tracks the file objects to find all their data reading instructions, e.g., *SharedPreferences.getString(key)*. Next, starting from the data reading APIs, CLDroid extracts the data item identifiers, i.e., key name, through string value analysis. If the data item identifier is the same as the target data item, CLDroid connects them and achieves the data item tracking through the data pool.

3.3 Determining Security Hazards

After understanding data propagation through data pools, we can check whether there is a vulnerable data flow from external apps to critical internal functionalities. Following such a data flow, an app attacker can launch cross-layer exploitation. However, it is still difficult to determine the security hazards that may be caused. This is because the app internal functionalities are diverse and app-specific. To mitigate this problem, we universally measure their security impacts from the perspectives of two attack vectors we found, i.e., data loading and consuming, which can compromise the target app's availability and integrity respectively. Below we first present the details of how to detect data loading and consuming-based vulnerabilities, and then we discuss how to determine caused security consequences.

3.3.1 Detecting Insecure Data Loading. To detect insecure data loading, CLDroid analyzes whether an external app can inject a large amount of data into the target data pool and if the injected data will be loaded into memory. First, CLDroid checks the injection possibility by analyzing the data flow from requesters to the writing APIs of the target data pool. Then, based on the semantics of the writing APIs, CLDroid can determine whether the volume of data pools can be unexpectedly increased (according to the tracked parameters). For example, if there is a data flow to the first parameter of *SharedPreferences$Editor.putString(key,value)*, it indicates that the attacker can inject large amounts of data by inputting different key names, which results in the file size increase. Last, CLDroid detects if the injected data can be loaded into memory. Specifically, we consider two scenarios: (i) loading all data from a data pool into memory and (ii) continuously reading data into a size-extensible variable (e.g., hashset). For the former case, some data pools, e.g., shared preference, need to load all data into memory before reading any value inside the file. Any instruction to read data from the data pool is considered a possible DoS risk. Thus, CLDroid checks the data identifiers corresponding to data pool reading operations by verifying the data pool type. For the latter case, some data pools, e.g., databases, use buffering to progressively load data, and memory overflow only occurs if the data is continuously stored in memory without being released. Thus, CLDroid

detects if there is a data flow from data pool reading APIs to the storing APIs of size-extensible objects. Furthermore, CLDroid detects if the reading is continuously executed. For example, if the data storing instructions are executed in a loop, while the loop termination condition is determined by the data read from the data pool, an insecure data loading is reported.

3.3.2 Detecting Insecure Data Consuming. For data consuming-based vulnerability, CLDroid detects if the poisoned data items can impact critical app functionalities. A question arises here: what functionalities are security-critical? In practice, we find app functionalities are quite diverse and many of them are specific to their corresponding apps. For example, the launcher app (shown in § 2.4) contains an important functionality of determining the app to be launched. We mitigate the problem based on the observation that the essential implementation of an app functionality is still delivered with Android system APIs or popular library APIs. For example, the launcher app uses Intent to open and manage apps. But different from prior work, which mainly focused on privileged system APIs, we consider a much boarder set of development APIs. These APIs may be insensitive for security, but more important for app business logic and the integrity of app behaviors. Following this, we build an extensive API list to help us understand data consuming-based attacks. For the API extraction, more details are shown in § 3.3.3 and Table 4.

Specifically, CLDroid considers two ways for abusing and manipulating internal functionalities: parameter manipulation and execution switch. For the former case, CLDroid conducts a data flow analysis to track the use of data items and check if there exist flows between the (potentially malicious) data item and parameters of the critical APIs. For execution switch, CLDroid first extracts related condition instructions that have data dependencies with the polluted data items. Then, for each condition instruction, CLDroid constructs sub-call graphs for its two branches and compares their invocations to critical APIs. The data item is considered critical if one branch calls the critical APIs while the other does not.

3.3.3 Understanding Security Hazards. To support the determination of security hazards, we construct an extensive list of critical APIs. These APIs help CLDroid understand the characteristics of data loading and consuming-based attacks. Although previous works [3, 32, 43] have provided an extensive set of sinks, they do not cover the sensitive APIs of the recent Android versions. Thus, we manually reconstruct a new list of sensitive APIs. Constructing such an API list involves much painstaking manual work, but the effort is one-time and can be reused and extended with ease. Specifically, we collect APIs from Android SDK and popular third-party libraries from AppBrain [2]. Finally, 1542 APIs are included (Table 4). These APIs are gathered from the perspectives of two attack vectors. For data loading, two attack manners about reading data are considered (i.e., L1 and L2): loading all data from data pools into memory and storing data into size-extensible variables. For data consuming, we mainly consider the functionality-critical APIs. Specifically, on account of the integrity of content shown to the user and communication with other entities, we summarize the APIs of C1-C4. Besides, considering the security of app-specific capabilities and resources, we collect APIs of C5-C6. Note that some APIs are commonly used in apps whose execution is not sensitive, e.g., *TextView.setText()*. We only care about the content they operate. Hence, CLDroid only checks if their parameters can be manipulated by attackers. While for other APIs whose execution can determine some app key features, e.g., *WindowManager.addView()*, we further analyze if their execution can be decided by injected data (execution switch).

4 EVALUATION AND SECURITY IMPACT

In this section, we apply CLDroid on a large set of popular apps to assess the security risks of cross-layer exploitation in the real world and then break down their security hazards.

Table 4. The summarized critical APIs that may cause security hazards if the dependent data are exposed.

Phase	Type	Critical API Description	# Num	Example
Loading	L1	Loading all data from data pools into memory.	47	SharedPreferences.getAll()
	L2	Storing data into size-extensible variables.	100	HashSet.add()
Consuming	C1	Determining the content displayed on the screen.	101	ImageView.setImageURI()
	C2	Determining sensor output to apps.	65	MediaPlayer.setDataSource()
	C3	Communicating with local app components.	39	Context.startActivity()
	C4	Communicating with servers and presenting content to users.	118	WebView.loadUrl()
	C5	Privileged system APIs that are used to access protected system functions and resources.	421	WindowManager.addView()
	C6	System APIs that do not require permissions but are only allowed to manipulate the caller app's own resources.	651	NotificationManager.cancel()

Data Set. We build the dataset by collecting the top 500 apps from 33 categories on Google Play. As a result, we successfully downloaded and gathered 14,349 unique apps as our experiment dataset.

Test Bed and Performance. We implement `CLDroid` in 12K-SLOCs of Java on the top of static analysis framework 'Soot' [29]. `CLDroid` analyzes the apps on a Ubuntu 18.04 LTS 64-bit server with 64 CPU cores (2.30GHz) and 212GB memory. The analysis is performed in parallel and has a timeout of 5 minutes for each app. On average, the analysis needs 44.4 seconds for each app. Finally, 13,824 apps have been successfully analyzed in total. The remaining apps either exceed the time limit or cannot be parsed by Soot.

4.1 Prevalence of Cross-Layer Threats

`CLDroid` successfully analyzes 13,824 apps and the detection results are shown in Table 5. Overall, `CLDroid` finds 10,839 data pools are shared and involved in 2,074 app-to-app data sharing channels, spanning over 1,714 (12.4% of 13,824) apps. In the discovered data pools, `CLDroid` successfully recovers 223,878 data item identifiers. In security-oriented experiments, `CLDroid` identifies 70.9% (1,215 of 1,714) apps are potentially vulnerable to cross-layer exploitation, with more than 18 billion installs in total. To be specific, 925 apps suffer from data loading-based attacks involving 3,507 data pools, and 947 apps are vulnerable to data consuming-based attacks involving 1,409 data items. We randomly select 60 potentially vulnerable apps and manually verify their detection results.

Table 5. Overall results of cross-layer threats discovered by `CLDroid`

Data Pool Type	Involved APPs	Exported Channels	Shared Data Pools	Data Items	Insecure Data Loading		Insecure Data Consuming	
					Data Pools	*Unsafe APPs*	*Data Items*	*Unsafe APPs*
Shared Preference	1075	1327	8874	205416	2576	261	813	400
Database	798	855	1684	18181	870	684	596	567
Others	193	201	281	281	61	17	0	0
Total	1714	2074	10839	223878	3507	925	1409	947

Identifying Data Sharing and Protection. For these 60 apps, `CLDroid` discovers 92 data sharing channels that could be abused for injecting malicious data. 59 of them are successfully injected by manual verification. There are several reasons for verification failures. First, 4 of them enforce security checks on the caller app's identity. For instance, the Samsung Internet Browser [48] checks the caller app's signature and only allows access from apps that are signed with the same certificate. Second, 4 components have hard-to-satisfy path constraints, e.g., a condition check that depends on the app run-time behaviors. Third, 16 are false positives introduced by the over-approximation of static analysis. Last, 9 fail to be manually analyzed due to code obfuscation.

Identifying Data Items. With manual verification, we confirm that these 60 apps expose 9,569 data items in their data pools, and `CLDroid` successfully identifies 9,275 (96.9%) data identifiers for them. The main reason for the failure is that many of the unresolved values need to be retrieved from function calls which are hardly analyzed via static analysis, e.g., system functions.

Identifying Cross-Layer Threats. We conduct a security analysis of these 60 apps, 41 of which are potentially vulnerable to insecure data loading and 38 to insecure data consuming. Specifically, we construct data poisoning requests from an unprivileged app and manually trigger the vulnerable data loading and consuming instructions in the victim app. Note that triggering their consequences can be difficult due to the manual efforts required which often involve various UI events. `CLDroid` facilitates this by precisely identifying vulnerable data loading and consuming operations in respective components. As a result, we confirm at least 32 (53.3%) apps can be successfully exploited and the overall results are shown in the Table 6. In detail, 28 of them are verified to suffer from data loading-based attacks and 27 of them contain insecure data consuming, which can affect critical functionalities inside the victim apps. There may exist cases (in the remaining 28 apps) that are difficult to trigger but can be exploitable with more effort. Therefore, our estimate of exploitable apps is only a lower bound. We responsibly disclosed the verified vulnerabilities to corresponding developers, and so far 56 CVE IDs have been assigned. Since `CLDroid` is the first to detect app cross-layer threats and there is no ground truth of all vulnerabilities, we lack a good way to predict false negatives.

4.2 Breakdown of Security Hazards

After verifying the exploitable apps, we break down their security consequences. Table 7 presents the overall results. Exploiting insecure data loading can lead to persistent app and functionality DoS, and exploiting insecure data consuming can achieve content spoofing and privilege escalation, which demonstrate the high severity of cross-layer exploitation. Below we present more details.

Persistent DoS Attacks. DoS is mainly caused by data loading-based attacks. If the size of a data pool can be controlled by an unauthorized app, a malicious app can inject large amounts of crafted data into it. This will cause the victim app to load excessive data into the memory, thus leading to DoS attacks. Since the injected data are persisted on the device and will not disappear after the app or device restart, the attack consequence (DoS) is persistent. Typically, executing such attacks requires over 500 app calls, as Android imposes a maximum limit of 512MB for app process memory, and the data transferred per inter-app call must not exceed 1MB. However, the impact on user experience during an attack can be mitigated to imperceptible levels. This is attributed to the persistent consequences of data injection from each call, enabling attacker apps to flexibly control the attack frequency at a low rate, thereby avoiding noticeable disruptions to users' regular usage.

- **App DoS.** When the injected data pool is essential for app initialization, it will be automatically loaded during app startup and trigger OOM. Thus, the victim app cannot be successfully launched anymore. A real case is our motivating example described in § 2.4.
- **Functionality DoS.** When the injected data is loaded only when the user visits a specific activity or uses a specific app feature, a functionality DoS occurs. For instance, the highlight feature, e.g., sound effect customization, of Poweramp (50,000,000+ downloads) can be disabled by injecting a large amount of data into its preset database.

Content Spoofing Attacks. Some vulnerable data determines the content presented to users. Attackers can tamper with the content to spoof users and further launch more sophisticated attacks.

Table 6. Examples of exploitable apps detected by CLDroid. Specifically, DoS stands for denial of service attacks in data loading. CS and PE represent content spoofing and privilege escalation attacks in data consuming respectively. Symbol ● means it is vulnerable to our attack. CVE with symbol ✔ means the vulnerabilities have been assigned with CVE IDs.

#ID	Package Name	Category	Downloads	Security Consequences			CVE	Description
				DoS	CS	PE		
01	*.launcher	Personalization	100M+	●	●	●	✔	Replace app font files with malicious files.
02	*.yandexnavi	Travel&Local	100M+	●		●	✔	Modify critical app settings, e.g., app notification.
03	*.simejikeyboard	Personalization	100M+	●		●	✔	Modify critical app settings, e.g., keyboard layout.
04	*.audioplayer	Music&Audio	50M+	●	●	●	✔	Manipulate sound effect and displayed UI content.
05	*.meetings	Business	50M+	●				Inject excessive data into a shared preference file.
06	*.edjingdjturntable	Music&Audio	50M+	●			✔	Inject excessive data into the playlist database.
07	*.security	Tools	10M+	●		●	✔	Manipulate virus scan whitelist, block in-app advertising and modify wifi security setting.
08	*.superlock	Tools	10M+	●		●	✔	Change app lock password and protected app list.
09	*.xsuperclean	Tools	10M+	●		●	✔	Block in-app advertising.
10	*.mp3player	Music&Audio	10M+	●			✔	Disable the search functionality in this app by injecting excessive data into search history database.
11	*.fasttyping	Personalization	10M+			●	✔	Arbitrary file overwrite and code execution.
12	*.who	Social	10M+	●		●	✔	Control the advertisements display settings.
13	*.solive	Social	10M+	●		●	✔	Manipulate the profiles of login users.
14	*.playstore	Personalization	10M+	●	●	●	✔	Manipulate app icons displayed on the phone screen and hijack inter-app communications.
15	*.lux	Health&Fitness	10M+	●		●	✔	Change app settings and system's display brightness.
16	*.bluelightfilter	Health&Fitness	10M+	●		●	✔	Change app settings and the system's display color.
17	*.liveFlightTracker	Travel&Local	10M+	●	●		✔	Manipulate the airport and airline information shown to users.
18	*.themeforandroid	Personalization	10M+			●	✔	Arbitrary file overwrite and code execution.
19	*.sleep	Lifestyle	10M+	●	●		✔	Modify UI settings and the audio file played for.
20	*.textme	Social	10M+		●		✔	Modify the audio file played for ringtone.
21	*.keyboard	Personalization	10M+	●		●	✔	Modify the urls for downloading language packs.
22	*.clean	Tools	1M+	●	●	●	✔	Manipulate app update settings and hijack the update to install malware.
23	*.blockCalls	Communication	1M+	●		●	✔	Manipulate the blacklist and whitelist of blocked phone calls.
24	*.amdroid	Productivity	1M+	●		●	✔	Modify the settings of phone's alarm clocks.
25	*.weather	Weather	1M+	●	●	●	✔	Manipulate the cached web content to launch content spoofing.
26	*.phone	Video Players	1M+	●				Inject excessive data into a shared preference file
27	*.truck	Transportation	1M+	●			✔	Inject excessive data into local file
28	*.crossx	Health&Fitness	500K+	●		●	✔	Manipulate user profiles.
29	*.byrk	Tools	500K+	●		●	✔	Modify the advertisement display settings.
30	*.unicornwallpaper	Art&Design	100K+	●	●		✔	Modify the URLs to load images.
31	*.wallpaperoffline	Art&Design	100K+	●	●		✔	Modify the URLs to load images.
32	*.android	News&Magazines	10K+		●		✔	Modify the URLs to fetch news.

Table 7. Breakdown of security consequences of 32 apps.

Attack Vector	Consequences	#APPs	Examples
Data Loading	Persistent APP DoS	21	SoLive
	Persistent Functionality DoS	7	Poweramp
Data Consuming	Content Spoofing	11	TextMe
	Privilege Escalation	21	BestWeather

- **UI Spoofing.** Due to the trust in used apps, the information presented on the UI is also trusted by users by default, which can be abused by attackers to deceive users. For instance, the tips on pop-up windows can be abused to trick users into downloading malicious apps (§ 5 Case#3).
- **Voice Command Injection.** Inspired by studies [45, 46, 61, 62] on voice assistants, manipulating the played audio files can open the gate for attacks through speakers. Specifically, a carefully-forged audio file can inject voice commands into the speech recognition systems adopted by IoT devices and smart vehicles. For example, by exploiting the insecure data consuming in TextMe (10,000,000+ downloads), the audio file played for ringtones can be replaced stealthily.

Privilege Escalation Attacks. Normally, the app's code is isolated and protected from other apps by the Android application sandbox. However, by controlling the exposed app-specific data, attackers can bypass the sandbox protection and indirectly affect the relevant code execution.

- **Functionality Manipulation.** The app behaviors may rely on some local settings. By tampering with these data, a third-party app can manipulate a victim app's key functionalities. For example, we find that a security app with 10,000,000+ downloads maintains a whitelist for virus scanning. A malicious app can add its own package name to the whitelist and evade virus detection.
- **Communication Hijacking.** An attacker may totally control who the victim app communicates with and also the communication content. The communication here includes not only local inter-app communication but also communication with remote servers. For instance, we find an attacker can control the server URL the victim app is going to access. Thus, the attacker can launch a phishing attack and steal user privacy data by replacing it with a malicious website.
- **Code Execution.** The app-specific data can open the way for attackers to execute code in the victim app's context. For example, by tampering with the path of a zip file to extract, an attacker can overwrite the dex files pre-saved in the app's internal storage and inject malicious code (detailed in § 5 Case#2).

5 REAL-WORLD CASE STUDIES

We now choose a subset of our results to demonstrate cross-layer exploitation in real-world scenarios. Note that all attacks are initiated by unprivileged apps and have been verified by us.

Case#1: Bypassing App Lock Protection. Lock Master (*.superlock, 10,000,000+ downloads) is a privileged security management app, which can help users enforce access lock on target apps. Its background service monitors and enforces password security validation when an app in its protection list comes to the foreground. Thus, it can protect sensitive apps, e.g., social or illness tracker apps, from being accessed by snoopers.

The app lock feature is strictly protected to be exclusively controlled by users and cannot be invoked by external apps. However, `CLDroid` discovers this functionality is vulnerable to cross-layer exploitation, which can bypass its strict protection and even block the original user's access to all apps. In particular, we find its critical dependent data (protected-app list and access password) is stored in a shared preference file with some trivial app data and is not properly protected. As a result, a malicious app can indirectly gain access to the protected apps by manipulating its protected-app list, and even block the original user's access to all apps by tampering with the user-set access password. This severely breaks the integrity and confidentiality of the app. `CLDroid` unveils this vulnerability by analyzing that the saved protected-app list and password determine the execution of privileged system APIs, i.e., *WindowManager.addView()* and *removeView()*, respectively. This vulnerability is rated as high severity and confirmed as CVE-2023-29733.

Case#2: Arbitrary Code Execution. Fast Typing Keyboard (*.fasttyping, 10,000,000+ downloads) is a customizable keyboard app. Mobile users can set it as the default keyboard of the phone. It allows users to select and download theme packs from various sources, including network and Google Play. However, we verify that through cross-layer exploitation, an adversary can stealthily overwrite arbitrary files in this app's isolated private storage by tampering with the path pointing to downloaded theme files, resulting in arbitrary code execution.

Specifically, `CLDroid` discovers a vulnerable data flow through a *shared preference* data pool to an arbitrary file write API. Our analysis shows this app stores the file paths (i.e., *go_res_zip_path*) of downloaded theme packs in a data pool. Unfortunately, the *go_res_zip_path* is not well protected and can be manipulated by unauthorized apps through a broadcast receiver. Even worse, this app

enforces no security checks when decompressing the theme packs and a path traversal vulnerability exists. Thus, an attacker can first let *go_res_zip_path* point to a crafted theme pack that contains a malicious dex file. Next, impacted by this data item, this app decompresses the crafted file. The malicious dex file is placed into this app's internal storage (overwriting the original benign dex file) and later executed. Note that, every time the app starts up, it automatically decompresses and installs the theme pack if the file pointed by *go_res_zip_path* exists in storage. Thus, the whole attack procedure is stealthy and does not require user interaction, which could be hardly noticed by victim users. This vulnerability has been rated as critical severity and confirmed as CVE-2022-47027.

Case#3: APP Update Hijacking. Super Clean (*.clean, 1,000,000+ downloads) is a powerful phone cleaner app for cleaning junk files and optimizing memory usage. In order to ensure that the app is updated in a timely manner, it will check the current app version each time the app is started and force users to update it once the app is outdated. However, through cross-layer exploitation, this process can be hijacked by attackers to trick users into downloading malicious apps.

Specifically, when the app is launched, it loads and checks the local app version record. If it needs to be updated, the app pops up a dialog with some update tips. The dialog guides users with an update button jumping to the new app page in app markets (e.g., Google Play). `CLDroid` discovers many critical data, e.g., app version, update tips, and link address of new app, are stored mixed with some harmless app configuration data in a *shared preference* file and are erroneously exposed in app-to-app data sharing. Thus, a third-party app could manipulate these data to form an attack chain and induce users to install a malicious app. First, by lowering the app version, the adversary can actively trigger the target app to pop up the update dialog. Then, the shown tips could be crafted to deceive the user, such as *'This app is no longer updated and maintained, please download our new app!'*. After the user clicks the update button, the jump destination is hijacked by modifying the new app's link address to update. Finally, malware could be installed on the victim's device. This vulnerability has been rated as high severity and confirmed as CVE-2023-27193.

6 MITIGATION, LIMITATION, AND DISCUSSION

Mitigation. To mitigate app cross-layer threats, we propose several mitigation strategies for app developers based on our analysis of the two attack vectors.

To prevent data loading-based attacks, apps can restrict requester access frequency by recording the source and timestamp of requests to increase the attack time cost. Furthermore, apps can set quantity restrictions for each external app when storing external data in the internal data pools. In case the limit is exceeded, the app can employ a FIFO strategy to clear out old data. In addition, when loading data from the data pool, it is advisable to use a streamlined approach where data is loaded and checked for size simultaneously or check the file size before loading it entirely into memory to ensure current available memory is sufficient.

For data consuming-based attacks, it is hard to design a one-size-fits-all protective method due to app-specific features. Standardized and fine-grained app data security management practices are urgently needed. First of all, to avoid unintended data exposure, app data should be stored separately considering their functional relevance and whether there is a need for sharing. Furthermore, for the necessary data to be shared, it is essential to carry out fine-grained sensitivity classification according to their app-specific purposes and organize them based on their sensitivity.

Furthermore, system vendors such as Google should explicitly notify developers about cross-layer security risks and optimize the design of best security development practices in their official documentation.

Limitation. The goal of this study is to explore the security impact of cross-layer threats in mobile apps. We design and implement `CLDroid` to identify instances that are of potential security

impact, bringing them to the surface for an analyst to further investigate and verify. Automatically reasoning about exploitability is beyond the scope of this study and is an extremely challenging task in practice, especially since the exploitation of such vulnerabilities can be tightly linked with app-specific business logic.

First, the analysis we performed in § 3 is conservative as we only focus on the commonly used intent-based data injection channels and app local storage data managed by Java APIs. Note that some app data are stored and operated in the cloud. We do not consider them since the internals of app backend services are black-box for us and it is hard to verify their attack consequences without affecting other users in practice. Second, `CLDroid` is limited in modeling non-linear data access, e.g., serialized data of objects, as their internal structures and access operations are nonstandard and customized. `CLDroid` analyzes them as a whole. Third, we implemented a simplistic version of string value analysis for Android apps drawing inspiration from prior work [63]. More advanced and robust string value analysis techniques [20, 31] could be applied for future extensions. Besides, the API-based critical data-consuming scenarios detection may overlook certain sensitive data usage in some apps due to the use of specific third-party libraries. We argue that the API list can be easily extended to accommodate specific customized app business logic. From a practical perspective, we believe our analysis was at an adequate level given the findings and goals of this study.

Discussion. While Android apps have been extensively studied, they continue to evolve and develop, with new architectures or features being introduced to enhance functionality and user experience. These new features may weaken or even break existing security principles, giving rise to new security demands. Therefore, their security also needs continuous improvement to keep the right balance between security and usability. Our study highlights the inadequacy of existing defense mechanisms when it comes to modern app-specific data management. The security mechanisms provided by the operating system are coarse-grained and inadequate to address the app-specific internal characteristics. The protective measures implemented by developers are discrete and lack uniform standards and best practices. Addressing this issue requires collaborative efforts from both developers and operating system vendors, and further research is warranted.

7 RELATED WORK

Insecure Communication Channels. Inter-app communication (IAC) is an essential feature in mobile apps, which supports the reuse of functionalities and data sharing across apps. However, it also constitutes a serious attack surface [26, 28, 30, 34, 35, 50, 51, 55, 58, 59]. Numerous studies [3, 11, 16, 21, 22, 25, 27, 32, 37, 47, 57] have investigated binder-based communication channels (i.e., intent) and discovered various vulnerabilities. For example, CHEX [37] detected component hijacking vulnerabilities by tracking taints between externally accessible interfaces and sensitive sources or sinks. Typically, existing work investigated code-layer attacks, which involved detecting whether sensitive or privileged functions can be directly invoked through public interfaces. Differently, we study cross-layer exploitation in mobile apps which is performed by polluting the dependent data of critical app functionalities. Since IAC is a common demand in apps, injecting data into data pools is not a strong indicator of sensitive operation and could be hardly used in vulnerability detection. In particular, ContentScope [27] studied the security of content providers and targeted verifying whether a content provider was exposed to other apps. However, we study which and how data items in the exposed data pools can be exploited to abuse or manipulate the apps' internal functionalities. With app cross-layer exploitation, even internal functionalities that cannot be touched by external apps may be exploited by attackers. In this paper, we conduct the first systematic study to understand their security impacts in the real world.

APP Data Security. Apps commonly maintain much sensitive and private user data, whose security has been studied by many researchers with both static [3, 6, 21, 24, 32, 33, 39–41, 57] and dynamic approaches [17, 38, 54, 56]. For example, IccTA [32] and DroidSafe [21] resolved the Intent and RPC calls to construct a precise inter-component model and detect privacy leaks via static taint analysis. TaintDroid [17] labeled data from privacy-sensitive sources and tracked the data flow in real-time. As a comparison, prior work aimed to detect privacy data leakage, while our work focuses on poisonous data injection. Furthermore, these tools mainly traced data flows that traveled along code execution and did not support fine-grained data flow tracking through data pools. In this paper, we propose `CLDroid`, which could understand the data pool access semantics and track data items through various data pools. Besides, privacy data can have obvious characteristics due to its strong correlation with the user and device, while our work focuses on app-specific data, which are tied to the diverse and customizable app functionalities, making them hard to pre-define.

8 CONCLUSION

In this paper, we revisit app component security in the context of modern app architecture and discover existing defenses have not caught up with the fast app development, leading to the feasibility of cross-layer exploitation. Then we design a novel vulnerability analysis tool, called `CLDroid`, to assess its security risks in real-world apps. Our experiments reveal that 1,215 apps (8.8%) are impacted, with more than 18 billion installs in total. Various serious security consequences have been verified, such as code execution, communication hijacking, phishing, and persistent DoS. Our findings highlight that new development practices should be designed carefully to keep the existing security mechanisms effective.

9 DATA AVAILABILITY

We have open-sourced `CLDroid`, which can be accessed on Github at our public repository https://github.com/LianKee/CLDroid.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Key Research and Development Program (2021YFB3101200), National Natural Science Foundation of China (62172104, 62172105, 61972099, 62102093, 62102091, 62202106). Yuan Zhang was supported in part by the Shanghai Rising-Star Program under Grant 21QA1400700 and the Shanghai Pilot Program for Basic Research - Fudan University 21TQ1400100 (21TQ012). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics Systems and Engineering Research Center of Cyber Security Auditing and Monitoring.

REFERENCES

[1] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. 2015. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1248–1259.

[2] AppBrain. 2023. *Android library statistics.* Retrieved April 5, 2023 from https://www.appbrain.com/stats/libraries

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[4] Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. 2014. Android security framework: Enabling generic and extensible access control on android. *arXiv preprint arXiv:1404.1395* (2014).

[5] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. 2013. AppGuard-Enforcing User Requirements on Android Apps.. In *TACAS*, Vol. 13. Springer, 543–548.

[6] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework.. In *Network and Distributed Systems Security (NDSS) Symposium*.

[7] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*. 239–252.

[8] John Corpuz and Jordan Palmer. 2023. *Best Android launchers 2022*. Retrieved April 5, 2023 from https://www.tomsguide.com/round-up/best-android-launchers

[9] Johannes Dahse and Thorsten Holz. 2014. Static detection of second-order vulnerabilities in web applications. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 989–1003.

[10] Johannes Dahse, Nikolai Krein, and Thorsten Holz. 2014. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 42–53.

[11] Biniam Fisseha Demissie, Davide Ghio, Mariano Ceccato, and Andrea Avancini. 2016. Identifying android inter app communication vulnerabilities using static and dynamic analysis. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*. 255–266.

[12] Android Developers. 2022. *Runtime Permissions*. Retrieved April 5, 2023 from https://source.android.com/docs/core/permissions/runtime_perms

[13] Android Developers. 2023. *Google MonkeyRunner*. Retrieved April 5, 2023 from https://developer.android.com/studio/test/monkeyrunner

[14] Android Developers. 2023. *Guide to App Architecture*. Retrieved April 5, 2023 from https://developer.android.com/topic/architecture

[15] Android Developers. 2023. *Security tips on content providers*. Retrieved April 5, 2023 from https://developer.android.com/training/articles/security-tips#ContentProviders

[16] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. {FIRMSCOPE}: Automatic Uncovering of {Privilege-Escalation} Vulnerabilities in {Pre-Installed} Apps in Android Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*. 2379–2396.

[17] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[18] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission re-delegation: Attacks and defenses.. In *USENIX security symposium*, Vol. 30. 88.

[19] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. 2020. An analysis of pre-installed android software. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1039–1055.

[20] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. 2013. JST: An automatic test generation tool for industrial Java applications with strings. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 992–1001.

[21] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *Network and Distributed Systems Security (NDSS) Symposium*, Vol. 15. 110.

[22] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic detection of capability leaks in stock android smartphones.. In *Network and Distributed Systems Security (NDSS) Symposium*, Vol. 14. 19.

[23] Roee Hay, Omer Tripp, and Marco Pistoia. 2015. Dynamic detection of inter-application communication vulnerabilities in Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 118–128.

[24] Jianjun Huang, Xiangyu Zhang, and Lin Tan. 2016. Detecting sensitive data disclosure via bi-directional text correlation analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 169–180.

[25] Yuede Ji, Mohamed Elsabagh, Ryan Johnson, and Angelos Stavrou. 2021. {DEFInit}: An Analysis of Exposed Android Init Routines. In *30th USENIX Security Symposium (USENIX Security 21)*. 3685–3702.

[26] Yunhan Jack Jia, Qi Alfred Chen, Yikai Lin, Chao Kong, and Z Morley Mao. 2017. Open doors for bob and mallory: Open port usage in android apps and security implications. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 190–203.

[27] Yajin Zhou Xuxian Jiang. 2013. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*.

[28] Xing Jin, Xunchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. 2014. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 66–77.

[29] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15.
[30] Phi Tuong Lau. 2019. Static detection of event-driven races in HTML5-based mobile apps. In *International Conference on Verification and Evaluation of Computer and Communication Systems*. Springer, 32–46.
[31] Ding Li, Yingjun Lyu, Mian Wan, and William GJ Halfond. 2015. String analysis for Java and Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 661–672.
[32] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.
[33] Shuai Li, Zhemin Yang, Nan Hua, Peng Liu, Xiaohan Zhang, Guangliang Yang, and Min Yang. 2022. Collect Responsibly But Deliver Arbitrarily? A Study on Cross-User Privacy Leakage in Mobile Apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1887–1900.
[34] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. 2017. Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 829–844.
[35] Chia-Chi Lin, Hongyang Li, Xiao-yong Zhou, and XiaoFeng Wang. 2014. Screenmilker: How to Milk Your Android Screen for Secrets.. In *Network and Distributed Systems Security (NDSS) Symposium*.
[36] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. 2017. Measuring the insecurity of mobile deep links of android. In *26th USENIX Security Symposium (USENIX Security 17)*. 953–969.
[37] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 229–240.
[38] Björn Mathis, Vitalii Avdiienko, Ezekiel O Soremekun, Marcel Böhme, and Andreas Zeller. 2017. Detecting information flow by mutating input data. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 263–273.
[39] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. 2015. {UIPicker}:{User-Input} Privacy Identification in Mobile Applications. In *24th USENIX Security Symposium (USENIX Security 15)*. 993–1008.
[40] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. 2018. Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps.. In *NDSS*.
[41] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 77–88.
[42] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 616–628.
[43] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A machine-learning approach for classifying and categorizing android sources and sinks.. In *NDSS*, Vol. 14. 1125.
[44] Matthew Rossi, Dario Facchinetti, Enrico Bacis, Marco Rosa, Stefano Paraboschi, et al. 2021. SEApp: Bringing Mandatory Access Control to Android Apps.. In *USENIX Security Symposium*. 3613–3630.
[45] Nirupam Roy, Haitham Hassanieh, and Romit Roy Choudhury. 2017. Backdoor: Making microphones hear inaudible sounds. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 2–14.
[46] Nirupam Roy, Sheng Shen, Haitham Hassanieh, and Romit Roy Choudhury. 2018. Inaudible Voice Commands: The {Long-Range} Attack and Defense. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 547–560.
[47] Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2015. Analysis of android inter-app security vulnerabilities using covert. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 725–728.
[48] Samsung. 2022. *Samsung Internet Browser*. Retrieved November 5, 2022 from https://play.google.com/store/apps/details?id=com.sec.android.app.sbrowser
[49] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. 2023. Silent spring: Prototype pollution leads to remote code execution in Node. js. In *USENIX Security Symposium 2023*.
[50] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. 2016. What Mobile Ads Know About Mobile Users.. In *Network and Distributed Systems Security (NDSS) Symposium*.
[51] Wei Song, Qingqing Huang, and Jeff Huang. 2018. Understanding javascript vulnerabilities in large real-world Android applications. *IEEE Transactions on Dependable and Secure Computing* 17, 5 (2018), 1063–1078.
[52] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. (2019).

[53] He Su, Feng Li, Lili Xu, Wenbo Hu, Yujie Sun, Qing Sun, Huina Chao, and Wei Huo. 2023. Splendor: Static Detection of Stored XSS in Modern Web Applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1043–1054.

[54] Mingshen Sun, Tao Wei, and John CS Lui. 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 331–342.

[55] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. 2013. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 635–646.

[56] Xiaolei Wang, Andrea Continella, Yuexiang Yang, Yongzhong He, and Sencun Zhu. 2019. Leakdoctor: Toward automatically diagnosing privacy leaks in mobile applications. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 3, 1 (2019), 1–25.

[57] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)* 21, 3 (2018), 1–32.

[58] Daoyuan Wu and Rocky KC Chang. 2015. Indirect file leaks in mobile applications. *Proc. IEEE Mobile Security Technologies (MoST)* (2015).

[59] Daoyuan Wu, Debin Gao, Rocky KC Chang, En He, Eric KT Cheng, and Robert H Deng. 2019. Understanding open ports in Android applications: Discovery, diagnosis, and security assessment. (2019).

[60] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. 2013. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 623–634.

[61] Qiben Yan, Kehai Liu, Qin Zhou, Hanqing Guo, and Ning Zhang. 2020. Surfingattack: Interactive hidden attack on voice assistants using ultrasonic guided waves. In *Network and Distributed Systems Security (NDSS) Symposium*.

[62] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. 2017. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 103–117.

[63] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1296–1310.

Received 2023-09-21; accepted 2024-01-23