Do App Developers Follow the Android Official Security Guidelines? – An Empirical Measurement on App Data Security

***Abstract*—The popularity of Android OS is largely credited to massive apps, and many app developers are involved in this ecosystem. On the other hand, various vulnerabilities are introduced into apps by developers carelessly, bringing security issues to users. To facilitate secure development and avoid common API misuses, Google provides a series of security guidelines and development practices for developers on the official developer community websites. However, the deployments of these guidelines in the wild have not been systematically evaluated. In this work, through large-scale app measurement (251,749 apps from 10 markets) and analysis, we investigated whether app developers follow the official Android security guidelines and the possible reasons behind it. In practice, we selected five guidelines related to app data security as representatives, covering: (1) secure file creation modes; (2) sensitive data storage; (3) validation check for file paths; (4) hardware ID usage; (5) custom permission protection. We also designed the corresponding detection strategies to check violations of the guidelines. The results show that most developers (> 90%) can comply with Guidelines 1 and 2. However, some guidelines have not been followed properly. For Guidelines 3, 4, and 5, less than 60% of developers followed the Google security suggestions.**

I. Introduction

As the most popular mobile operating system, Android has attracted an increasing number of developers to join the app development community. The number of mobile developers is expected to reach 27.7 million by 2023 [1]. Furthermore, due to the support of the Android SDK and the official development documentation, the number of apps in the Android ecosystem has significantly increased. As of 2023, there are 2.87 million apps available on Google Play [12]. However, the weak security awareness among developers has led to various app security vulnerabilities, as they tend to prioritize app functionality over security requirements [28]. Disturbingly, recent data from the Atlas VPN team indicates that 63% of Android apps contain known security vulnerabilities, averaging 39 vulnerabilities per app [13].

To further improve the app security and avoid common API misuses, Google proposes *app security best practices* in the Android Developer Documentation to guide wide developers [5]. These guidelines cover various aspects of app security, such as data storage, communication, permission, and user privacy. For example, the guidelines suggest developers should only use the log functions to output debugging information, not containing sensitive data. Though Google provides many app security guidelines, their deployment in the wild have not been systematically evaluated. The related essential research questions have not been systematically answered, such as "*Do app developers comply with these guidelines? If not, what are the corresponding reasons?*"

Our Work. In this work, through large-scale app measurement and analysis, we investigated guidelines' deployment status and possible reasons of guideline violations among developers. On a high level, we need to select appropriate guidelines, design & implement detection rules, analyze measurement results, and evaluate the guidelines' deployment status.

Google provides various app security guidelines, and we cannot cover all of them due to page limitations. For large-scale measurements, we select guidelines according to their range of action. For example, the evaluation of the network module needs to receive or send messages from the cloud server. However, not all apps have this function. Data is an integral part of every app, and the evaluation of data security guidelines can cover more apps. Therefore, as representatives, we mainly focus on the guidelines related to *app data security*. We carefully reviewed the Android Developer Documentation and extracted five guidelines for data security design. As listed below, they can be divided into three categories: [data storage: Guideline 1 and 2]; [data usage: Guideline 3 and 4]; [data protection: Guideline 5]. More details are given in Section II.

Guideline 1: *Use secure modes when creating files* [24].
Guideline 2: *Do not leak sensitive data on external storage or in logs* [16], [22], [21].
Guideline 3: *Check the validity of file paths when reading files* [14], [9].
Guideline 4: *Avoid using non-resettable hardware IDs* [23].
Guideline 5: *Exported components should be protected by appropriate custom permissions* [18], [7], [4].

To evaluate the deployment status of the above security guidelines among Android app developers, we constructed the corresponding lightweight detection rules. Further, we conducted a large-scale measurement on 251,749 apps collected from multiple app markets. The measurement results show that most developers (> 90%) can comply with Guidelines 1 and 2, while the situations of other guidelines are not very well. For Guidelines 3, 4, and 5, less than 60% of developers followed them well. Furthermore, we combined multiple dynamic and static analysis methods to obtain possible reasons of guideline violations, such as measurement results analysis, manual reverse analysis, development simulation analysis, and community resources analysis. By cross-validating the results obtained by different analysis methods, we can infer the possible reasons of guideline violation.

Though the specific reasons are different by guidelines, in general, both Google (official developer documentation and Android Studio – the official IDE) and app developers (code development habits) should be responsible for the violation of Guidelines 3, 4, 5. By analyzing the possible reasons of guideline violations, developers can gain insight into potential security issues and improvement directions during development. Additionally, we provide suggestions for improving guideline deployment to Google and app market maintainers.

Contributions. The main contributions of this paper are:

- *Guideline deployment Evaluation.* Based on 251,749 apps collected from multiple app markets, we evaluated the deployment status of data security guidelines in the wild.
- *Guideline violations analysis.* We systematically investigated the possible reasons of guideline violations in apps, which can help improve the deployments of guidelines.

II. Android Security and Guidelines

In this section, we provide the necessary background of Android OS security mechanisms and Google's data security guidelines for app developers.

A. Android Security Mechanisms

On Android, Google designs a series of security measures to protect data, such as permission and sandbox mechanism.

App Sandbox. In Android, apps run in independent sandboxes. Each Android app runs in its own process, with its independent memory space and private data [6]. The sandbox can be treated as a safe execution environment for apps and a restriction for malicious attackers. The implementation of the sandbox mechanism is based on the Linux access control mechanism, which utilizes user IDs to separate different app processes. By default, apps cannot interact with each other, and data access to other apps is restricted. The deployment of the sandbox mechanism further clarifies the app's security boundary and minimizes the impact of malicious behaviors.

Permission. Android refines the access control mechanism of Linux and uses a permission-based mechanism to protect access to sensitive resources [15]. System permissions are usually used to protect system resources. For example, if app developers want to save files to external storage, developers need to request the `WRITE_EXTERNAL_STORAGE` permission. Otherwise, the system will prohibit developers from accessing external storage. Android permissions are divided into three levels [3]: (1) `Normal`: As long as the app applies for the permission, it will be granted automatically. (2) `Dangerous`: The app applies for and requires user authorization to use the permission. (3) `Signature`: If the requesting app is signed with the same certificate as the app declaring the permission, it can be used.

B. Google's Guidelines for App Data Security

In this study, we mainly focus on the deployment of data security guidelines. The reason for selecting data security guidelines will be explained in Section III-B. Here we summarize Google's guidelines and suggestions for app data security:

Guideline 1: *Use secure modes when creating files* [24].

$\Rightarrow$ App developers can select the file creation mode when creating the `SharedPreferences`, database, and private files. `SharedPreferences` is a lightweight storage class, mainly used to store preference data (key-value pairs). Database files are mainly used to store structured record information, and private files are mainly used to store apps' private data. Each mode corresponds to a constant value. For example, `MODE_PRIVATE` corresponds to `0x0000`.

Insecure file modes will lead to data leakage. For example, under `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE`, the created files can be read and written by other apps without any permission. These insecure modes are designed to share files among apps, such as configuration files. Also, `MODE_MULTI_PROCESS` only can be used when creating `SharedPreferences` files. This file mode is mainly designed to launch the multi-process mode to improve operating efficiency and share `SharedPreferences` among multiple processes. However, this mode is not process-safe and may cause inconsistent reading and writing of `SharedPreferences` files by multiple processes. Currently, Google has provided more secure design solutions to achieve the same functions and recommends not using these insecure modes.

Guideline 2: *Do not leak sensitive data on external storage or in logs.* [16] [22], [21].

$\Rightarrow$ The storage space of Android OS is divided into internal storage and external storage. Internal storage is private to apps, and external storage is shared between apps. Therefore, app developers should store sensitive data on internal storage and non-sensitive data on external storage. In addition, the log function is designed to output some debugging information at runtime, and developers may leak sensitive data related to the app through logs. If malicious apps obtain the `READ_LOGS` permission, they can read sensitive data leaked by other benign apps through logs.

Guideline 3: *Check the validity of file paths when reading files* [14], [9].

$\Rightarrow$ The file path validity means that when developers decompress files or use `openFile` method under `ContentProvider` class for file sharing, they need to check whether the file path contains a relative path represented by "..". If developers do not check the file path validity, the app may be at risk of path traversal vulnerability. In that case, the app may incorrectly open a file indicated by the attacker or decompress the file to the location specified by the attacker.

Guideline 4: *Avoid using non-resettable hardware IDs* [23].

$\Rightarrow$ There are four hardware identifiers in Android, including `IMEI` (International Mobile Equipment Identity), `Serial Number`, `MAC Address`, and `SSAID` (Android ID). The hardware identifier is mainly used for device binding. These hardware identifiers may be leaked to attackers through network transmission, logs, etc. If the hardware identifier is leaked, it will harm the user's privacy, leading to the risk of device (user) tracking.

Guideline 5: *Exported components should be protected by appropriate custom permissions* [18], [7], [4].

⇒ In addition to system permissions, permissions also can be defined by app developers, called custom permissions. They are used to protect the data generated by apps and usually act on the four major components of Android. Using custom permissions can achieve data sharing between trusted apps and prevent malicious apps from accessing data.

This guideline focuses on two aspects: (1) Android has four major components: `Activity`, `Service`, `Receiver`, and `Provider`. In order to make the four components accessible to other third-party apps, developers can set components to be exported. Further, developers can use the `signature`-level custom permissions to protect these exported components. The `signature`-level ensures the data is shared among trusted apps. (2) Developers must follow the principle of least privilege, apply the defined custom permissions to exported components, and do not keep unused permissions in the manifest files. If the developer does not follow the above guidelines, it may leak the app's sensitive data.

III. Methodology and Dataset

To evaluate the proposed app data security guidelines, in this section, we illustrate our measurement approaches and app datasets.

A. Methodology

As illustrated in Figure 1, on a high level, our measurement contains three main steps, as follows:

- *Guidelines Selection.* First, we reviewed the entire Android security documentation to extract the suggestions on app data security, as described in Section III-B.
- *Dataset Construction.* Next, we constructed the app dataset used in the study, including ten popular third-party app markets, as described in Section III-C.
- *Measurement and Analysis.* Finally, we evaluated the deployment status of data security guidelines and investigated the possible reasons of guideline violation, as described in Section III-D.

B. Guidelines Selection

Google's guidelines on Android app security cover various aspects, such as interaction, data, and network. All security guidelines are crucial for developers to build secure apps, but we can only cover a subset of guidelines in this paper due to page limitations. When selecting guidelines from the Android Security Documentation, we followed the principles below: (1) *Cover as many apps as possible.* The guidelines we choose should cover as many apps as possible to benefit more developers and users. (2) *Cover the whole lifecycle.* The subset of guidelines we choose should be related to each other and cover the whole lifecycle of a certain module. For example, the lifecycle of the network module includes message sending, receiving, and processing.

Considering the above two requirements, we prioritize showing data security guidelines after reviewing all guidelines. Mainly because (1) data is an integral part of every app. The evaluation of data security guidelines can cover more apps and benefit more developers and users. The action ranges of some modules are limited. For evaluating network-related guidelines, the app may be required to send or receive some messages to the cloud server. Still, not all apps have the function of network communication. (2) There is a strong correlation among our chosen guidelines covering data use, storage, and protection. These guidelines cover the whole lifecycle of data-related operations. In addition, there is currently no comprehensive study on official guideline-based Android app data security, and we fill the gaps in this research area.

Note that some security guidelines focus on relevant development aspects. For better demonstration, we aggregated multiple practices/suggestions into one guideline. After aggregation, we obtained five data security guidelines, as shown in Section II-B.

C. Dataset Construction

Since Google Play did not provide the app bulk downloading APIs anymore, the app dataset was constructed based on multiple third-party app markets for universality. Regardless of whether developers belong to Google Play or the third-party app market, they should all follow the data security guidelines to improve the security of apps. We selected ten popular third-party app markets to crawl apps, including F-Droid, 9apps, 360, 2265, Anzhi, LapTopPCAPK, Lenovo, Leyou, Mdpda, and Uptodown. Since app developers may upload apps to multiple app markets, we calculated the MD5 value of each app to ensure the uniqueness of apps in the dataset. After deduplication, a total of 251,749 apps were obtained.

D. Measurement and Analysis

After extracting the data security guidelines, we implemented all the development behaviors described by the guidelines, taking into account the different implementations of these guidelines in apps, especially for guideline violation. According to the implementation form of the guidelines in apps, we heuristically constructed detection rules for each guideline and performed large-scale measurements on the apps in our dataset. To make the measurement results more accurate, we did not evaluate the compliance of some official libraries to these guidelines. These official libraries are usually supporting libraries that keep apps running, and their package names usually start with `com.google.*`, `com.android.*`, `androidx.*`, and `android.*`. After that, we conducted a multi-dimensional analysis of the measurement results to evaluate the deployment of data security guidelines proposed by Google.

After obtaining the measurement results, we focused on analyzing the possible reasons of guideline violation. Our violation reason analysis follows the following process:

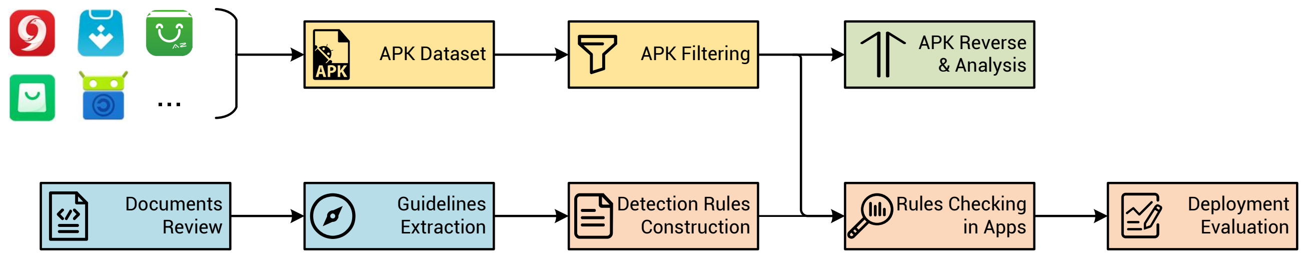

Fig. 1: Workflow of our measurement.

- *Measurement results analysis*: Based on our measurement results of each guideline, we can directly infer the possible reasons of guideline violation.
- *Manual revere analysis*: We unpacked the apps violating this guideline and analyzed the code. In practice, we focused on the apps that violated this guideline multiple times, the apps in different categories and versions.
- *Development simulation analysis*: We developed a demo app to re-produce the process of misuse and analyzed the possible reasons of guideline violation.
- *Community resources analysis*: Developers usually learn the use of API from some third-party Q&A websites, so we checked some online Q&A sites (such as Stack Overflow) to find hints of guideline violation.

IV. Measurement and Findings

In this section, we present our empirical research methods and results on the app data security guidelines described in Section II-B.

Guideline 1: Use secure modes when creating files.

To this guideline, we mainly focus on the file mode developers use when creating files. In detection, we extracted the integer value corresponding to the file mode by identifying the API that creates `SharedPreferences`, database, and private files. After that, the extracted values will be converted to file modes for further analysis.

Usage of Secure File Modes. We counted the usage times of different file modes for creating `SharedPreferences`, database, and private files in 251,749 apps, as shown in Table I. The corresponding percentages of creating files in secure modes are 94.01% (7,309,686), 97.93% (54,233), and 86.19% (341,287), respectively.

Usage of Insecure File Modes. There are mainly three risky file mode usage cases:

(1) *Incorrect usage of a single file mode.* The incorrect usage of a single mode indicates that the developers use the mode marked insecure in Table I. The proportions of such errors used in `SharedPreferences`, database, and private files are 4.73% (367,751), 0.95% (527), and 10.7% (42,353), respectively.

(2) *Incorrect usage of file mode combinations.* Android allows the assigned file mode can be a combination of multiple modes. It will be more harmful if the combination includes multiple insecure modes. For example, [`MODE_WORLD_READABLE + MODE_WORLD_WRITEABLE`] can grant other apps read and write permissions to the file. In our measurement, The proportions of insecure combinations are 0.57% (43,962), 0.03% (19), and 3.07% (12,142), respectively.

(3) *Usage of unsupported file modes.* Some developers used unsupported modes to create files. For example, `SharedPreferences` files only support four modes – `MODE_PRIVATE`, `MODE_WORLD_READABLE`, `MODE_WORLD_WRITEABLE`, and `MODE_MULTI_PROCESS`. However, some developers specify other modes such as `MODE_APPEND` or some unknown types of integer values. That is, these developers may be confused about the modes supported by each type of file. Also, some developers even do not know that the file mode is a fixed constant value, and they fill in an integer value casually. The proportions of this error used in the above three kinds of files are 0.69% (53,690), 1.09% (601), and 0.05% (195). After testing, we find that the files created with unsupported modes will be treated as `MODE_PRIVATE`. However, there are no security guarantees, i.e., with later versions of Android, file permissions created with unsupported modes can become dangerous, such as becoming world readable and writable.

```
private void addImageAttachment(String
    fileName, Bitmap bitmap) {
  FileOutputStream fileOutputStream = null;
  if (fileName != null && bitmap != null) {
    try { fileOutputStream = this.mContext.
        openFileOutput(fileName, 1);
      bitmap.compress(Bitmap.CompressFormat.
          PNG, 25, fileOutputStream);
      Uri fileUri = Uri.fromFile(new File(
          this.mContext.getFilesDir() + File
          .separator + fileName));
      this.mEmailAttachments.add(fileUri);}
      ...}}
```

Listing 1: Example of file sharing.

The possible reason why developers use insecure file mode is to achieve file sharing between apps. If the file is not set to `MODE_WORLD_READABLE`, the shared apps cannot obtain the shared content. For example, the app Monster[1] shares the image to mail app as an attachment of the mail, as shown in Listing 1. In this case, Monster specifies that the image is stored in the internal storage and sets the file mode to `MODE_WORLD_READABLE` through the `openFileOutput` method. Then,

[1]Package name: `nyanko.monster`

TABLE I: Configured file modes when creating files.

Mode	Value	Security	SharedPreferences	Database	Private File
MODE_PRIVATE	0x0000	Secure	7,309,686	54,206	321,125
MODE_WORLD_READABLE	0x0001	Insecure	53,612	319	40,302
MODE_WORLD_WRITEABLE	0x0002	Insecure	21,825	208	2,051
MODE_MULTI_PROCESS	0x0004	Insecure	292,314	N/A*	N/A
MODE_ENABLE_WRITE_AHEAD_LOGGING	0x0008	Secure	N/A	6	N/A
MODE_NO_LOCALIZED_COLLATORS	0x0010	Secure	N/A	21	N/A
MODE_APPEND	0x8000	Secure	N/A	N/A	20,162
MODE_INSECURE_COMBINATION†	Multiple◇	Insecure	43,962	19	12,142
MODE_ERROR_USE‡	Multiple	Insecure	53,690	601	195

∗: N/A represents the file does not support this mode.
†: MODE_INSECURE_COMBINATION represents a combination of unsafe file creation modes, such as MODE_WORLD_READABLE + MODE_WORLD_WRITEABLE.
‡: MODE_ERROR_USE represents a wrong mode is used to create the file.
◇: Multiple represents multiple values in this mode.

it will convert the image to an insecure file:/// Uri via the fromFile method. File Uri access control needs to open the underlying file system permissions, and the open permissions will be available to any apps until the next close.

Note that when creating database files, developers used the secure mode at the highest percentage. Unlike other types of files, database files are usually structured data, consisting of tables and records. Since using MODE_WORLD_WRITEABLE and MODE_WORLD_READABLE (insecure mode) can only share the entire database, developers are more inclined to use ContentProvider to achieve fine-grained access control.

Our Assessment: Most developers create files using secure modes, but some use insecure file modes to implement the file-sharing function.

Guideline 2: Do not leak sensitive data on external storage or in logs.

To this guideline, we mainly focus on whether developers leak some sensitive data on external storage or in logs.

Leakage on External Storage. We extracted the filenames on external and internal storage according to the file storage paths in the apps. To characterize how developers use the external storage, we performed a word frequency analysis on the filenames extracted from external storage. Also, third-party libraries usually create files to save some information. To exclude the influence of filenames generated by third-party libraries on the frequency of hot words, we save each filename into a collection for deduplication. In addition, we removed some meaningless words, such as "android" and "app".

The word frequency analysis results show that developers mainly use external storage to save pictures (such as "photo", "image", "camera", "picture"), videos (such as "video", "audio"), and other media data. We also compared whether there is an overlap between external and internal storage hot words. The result shows that 93 of the top 200 hot words on internal and external storage are the same. It means that developers cannot clearly distinguish whether some types of files should be stored on internal or external storage. As a result, external storage may contain sensitive data. For example, "log" is a hot word for both internal and external storage, and logs should be stored on internal storage. The log saves various information outputs by apps at runtime, which may contain sensitive data. However, many developers believe that log information is not critical and store it on external storage. In addition, the external storage also leaks the following sensitive data: Database, Contact, Plugin, QR Code, Screenshot and so on.

Leakage in App Logs. In this part, we adopted MobiLogLeak [37] to analyze the sensitive data leaked through app logs. MobiLogLeak is a taint analysis-based tool that classifies sensitive information into four categories: network, account, location, and database. For example, the app OSRSHelper[2] leaks the account and username through the following log:

```
DBController: getAccountByUsername: account=
com.infonuascape.osrshelper.models.Account@b95a6be
username=Alice
```

Also, the path-sensitive data flow analysis often leads to the path explosion problem, causing overhead. Considering our server's performance and running time, we only performed log leak analysis for apps less than 10 MB. There are 95,963 apps left in our dataset, which is still considerable and can reflect general statistical characteristics. In addition, we improved MobiLogLeak with the support of multi-process analysis for large-scale measurements.

Based on our statistics, 4,473 apps (4.66%) leaked sensitive data in their logs, for a total of 24,187 leaks. It is worth noting that our measurements are only a lower bound on actual leakage. When considering large-size apps, the leak may be more severe since they integrate more functions and add more logs for debugging. The usage times and ratios of different log levels for leaking sensitive data are listed in Table II. It shows that most developers prefer to use the Info and Debug levels to output logs. In addition, many leakages appear in the Warn and Error levels, which is also an alarming development practice. Warn is mainly to remind the developers that there may be potential risks, and Error is primarily used to print the error information of the app. Leaking sensitive data in logs is mainly for debugging purposes, so Verbose, Debug, and Info

[2]Package name: com.infonuascape.osrshelper

TABLE II: Log levels configured in apps.

Log Level	Verbose	Debug	Info	Warning	Error
Amount	1,209	10,213	7,159	2,682	2,924
Percentage	5.00%	42.22%	29.60%	11.09%	12.09%

levels should be used instead of `Warn` and `Error` levels. In our measurement, there are mainly two types of data leakage – network and database, of which proportions are 77.07% (18,641) and 22.93% (5,546), respectively. Most network-related leaks are the data about hardware identifiers, consistent with the heavy use of hardware identifiers we measured in Guideline 4. The database is mainly the leakage of some field information in database tables.

The possible reason for sensitive data leakage is that the developers want to confirm that the return value of the function or the value of the field is the expected result. For example, the app SunstarHealth[3] outputs the returned `IMEI` value into log to judge whether the call to `getDeviceId()` is successful, resulting in the leakage of `IMEI`. As stated in Guideline 4, it will cause privacy issues such as user tracking. The secondary reason is that developers may debug multiple times and not delete log information immediately after each debug. Also, many developers do not know how to automatically remove logs in apps [17]. There may be thousands of source files for the current Android app project, and only manually deleting the log will cause much time overhead. Further, the developers lack security awareness and do not remove the log in apps.

Our Assessment: App developers may not distinguish sensitive data well, and such data appears on external storage. In addition, though the leakage in app logs is not very common (< 5%), the leaked data is vital, usually relating to hardware identifiers and databases.

Guideline 3: Check the validity of file paths when reading files.

This guideline mainly focuses on whether developers have checked the validity of file paths when reading files, and the file path should not contain relative paths.

As mentioned in Section II-B, when developers decompress files or use the `openFile` method for file sharing, they need to check file path. In our dataset, around 28.12% (70,792) apps decompress files with 130,442 decompression operations. Among them, 78.01% of operations (101,755) are vulnerable, and most developers did not sterilize the file paths. Decompressing a zip compressed package has a relatively standard process: (1) Convert the compressed package to an input stream, and further convert it to a `ZipInputStream`. (2) Traverse each file in the compressed package and specify the decompression location of the file. (3) Convert each file to an output stream and extract it to the destination directory. This process may be a bit complicated for inexperienced developers, and the Android official documentation does not provide a concrete example to guide developers.

[3]Package name: `com.cinvision.sunstarhealth`

On another aspect, only 10.62% (26,725) apps override the `openFile` method under `ContentProvider` class for file sharing. Such a low ratio is a good phenomenon because if the `openFile` method is overridden in apps without filtering the file path, the apps will be exposed to the risk of path traversal vulnerabilities. The reason developers rarely override the `openFile` method is probably because it is more complicated to implement file sharing through the `openFile` method. As mentioned in Guideline 1, developers only need to specify the file creation mode to achieve (insecure) file sharing. A total of 30,485 `openFile` methods are overridden in 26,725 apps, of which only 17.85% (5,443) sterilize file paths.

In the above two path traversal vulnerabilities, only a small number of developers actively sterilize file paths. The possible reason may be the misleading of incorrect online code examples. We find that the Android official documentation does not provide concrete code examples to check file path validity. Facing the lack of official guidance, an empirical survey by Zhang et al. [36] shows that developers often learn the use of new APIs through online Q&A forums such as Stack Overflow and often suffer from API misuse issues. To investigate this issue, we reviewed the top answers on Stack Overflow about file unzipping and file sharing using `openFile` method [25] [20]. The result shows that none answers sterilize the file paths, and developers may directly reference these incorrect code snippets into their apps to quickly fulfill the business requirements of the apps.

Our Assessment: Only a small fraction of developers check the validity of the file paths when reading files, and Google should provide concrete API usage examples for reference to avoid path traversal vulnerabilities.

Guideline 4: Avoid using non-resettable hardware IDs.

This guideline mainly focuses on whether developers use non-resettable hardware identifiers, such as `IMEI` (International Mobile Equipment Identity), `Serial Number`, `MAC Address`, and `SSAID` (Android ID). We mainly analyzed the usage times of hardware identifiers in apps by parsing all the APIs for obtaining hardware identifiers. Then, we also got app's `targetSdkVersion` to analyze the average usage of hardware identifiers in different `targetSdkVersion`.

Overall Statistics. In our app dataset, 173,572 apps (68.95%) used at least one hardware identifier. We counted the total number of the four hardware identifiers, as listed in Table III. We can find that developers are more inclined to use `IMEI` than the other three identifiers. It is mainly because `IMEI` exists in every version of Android and has been added since API Level 1. In addition, it is more stable and easier to obtain than other identifiers. `SSAID` can be easily changed on a rooted device, and the `MAC Address` is only available after connecting to the Internet. Although `Serial Number` is relatively stable and not easy to be changed, it is used far less than the other three hardware identifiers. The possible reason is that the `Build.Serial` API is not available in all Android phones, and it was added at API Level 9. Also, the

TABLE III: Total usage of hardware identifiers.

Identifier	IMEI	S/N	MAC Addr	SSAID
Amount	1,650,153	191,064	768,581	747,133

value returned by Build.Serial may be inconsistent with the value listed in the Settings of the mobile phone [2] [10]. That is to say, the actual Serial Number cannot be acquired through the Build.Serial API. Therefore, many developers use Java reflection to access non-SDK APIs to read the Android.os.SystemProperties configuration file for obtaining the actual Serial Number. Non-SDK APIs have no compatibility guarantee and may be removed from the Android framework at any time, causing runtime crashes in apps [19]. Therefore, the usage of Serial Number is not common, and the total number of uses is far lower than the other three hardware identifiers.

```
public void getDetailMsg() {
  this.params = new HttpParams();
  UserEntity user = Find_User_Company_Utils.
      FindUser();
  PostUtil.postDefultStr(this.params, System
      .currentTimeMillis() + "", "", this.
      mActivity);
  this.params.put("markAddress",
      GetUniqueAndroidKeyUtil.getDeviceId(
      App.getInstance().
      getApplicationContext()));
  this.params.put("companyid", user.
      getCompanyid());
  this.params.put("userid", user.getUserid()
      );
  Core.getKJHttp().post(App.siteUrl + "
      appAttendance/attendanceIndex_new.
      action?n=" + Math.random(), this.
      params, this.getBack); }
```

Listing 2: Example of device binding using IMEI.

App developers mainly use hardware identifiers to bind the app to the device (user) for unique identification. For example, the app QuanFangTong[4] identifies a user by binding userid and IMEI (obtained through getDeviceId method) for achieving the function of employee clocking in, as shown in Listing 2. However, since the hardware identifier is usually associated with the device user, it may harm the user's privacy. Device tracking can be performed when a malicious attacker obtains the victim's hardware identifier.

Hardware IDs Migration. Note that, due to Google's restriction, if the apps with targetSdkVersion $\geq$ 29, obtaining some hardware identifiers in apps will involve a compatibility problem. For example, third-party apps cannot obtain the IMEI and Serial Number. If they are queried in code, a SecurityException will be thrown. For MAC Address, the Android OS will generate a random address for each query, which cannot uniquely identify the device [8]. To the apps with targetSdkVersion $\geq$ 29, the average usage times of the preceding hardware identifiers are decreasing gradually, as plotted in Figure 2. Especially for IMEI and MAC Address, because they have too many average references in apps before targetSdkVersion < 29, a cliff-like drop occurred when targetSdkVersion = 29.

[4]Package name: net.quanfangtong.hosting

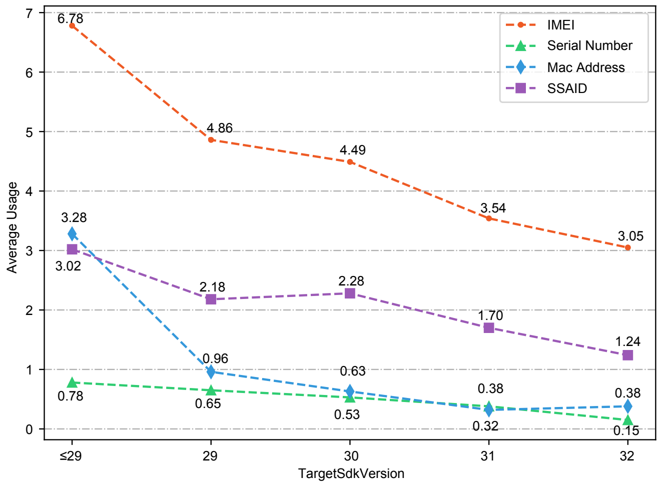

Fig. 2: Used hardware identifiers by targetSdkVersion.

Our Assessment: Using non-resettable hardware IDs may lead to the risk of device tracking. In practice, many app developers still use hardware identifiers for the device (user) binding and identification. This problem is gradually easing with the API restriction of Google after API Level 29.

Guideline 5: Exported components should be protected by appropriate custom permissions.

This guideline mainly focuses on the exportation of app components and the usage of custom permissions for protecting exported components. We mainly analyzed apps' AndroidManifest.xml files, and parsed all the four components defined and exported, as well as the custom permissions defined and unused.

Exported Components. Among the 251,749 apps, 230,201 apps (91.44%) have at least one exported Activity (the default export of MainActivity has been excluded). For Service, Receiver, and Provider, the percentages are 53.98%, 61.99%, and 20.58%, respectively. Activity exports are common in apps because developers need to use third-party SDKs to receive callback results from third-party apps. To receive the returned results correctly, third-party apps must be able to access the Activity in the developer's app to pass the callback result, so the Activity must be exported. For example, 52,187 apps in our dataset integrate the WeChat payment SDK, and these apps need to receive the returned results (success or failure) of payment after initiating new WeChat payment requests.

Also, as plotted in Figure 3, the total export percentages of all four major components in apps are 9.25%, 28.43%, 56.28%, and 15.33%. That is, the export rate of Activity is relatively low. After investigation, It is likely due to the default

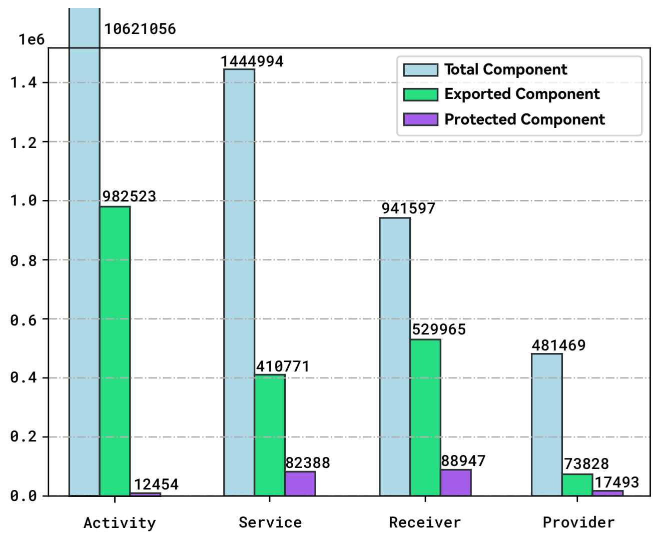

Fig. 3: Statistics of major components used in apps.

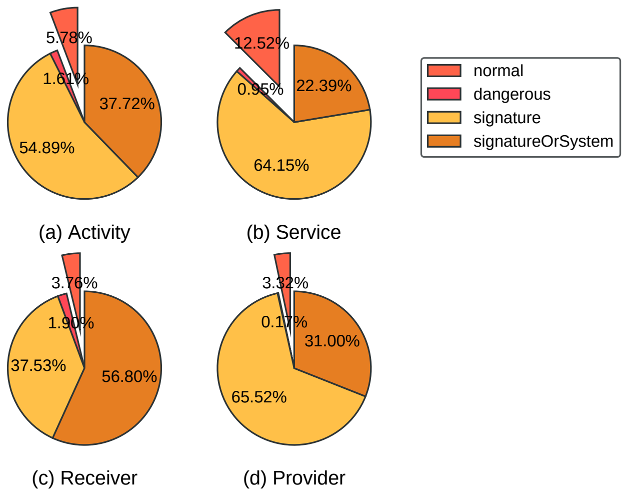

Fig. 4: Protection levels of custom permissions.

attribute configurations of Android Studio. Android Studio will set them exportable by default when developers create the `Service`, `Receiver`, and `Provider` components. However, to `Activity`, the default attribute is unexportable. The exported components may lead to the risks of information leakage, phishing, and denial of service [11]. For example, suppose the exported `Activity` does not handle exceptions to the data obtained by `Intent.getXXXExtra()`. In that case, an attacker can crash the app by sending empty, abnormal, or malformed data.

Protected Components. On the other hand, only 1.27% of exported `Activity` are protected by custom permissions. The percentages of `Service`, `Receiver`, and `Provider` being protected are higher than that of `Activity`, say 20.06%, 16.78%, and 23.69% respectively. The reason is that, after using the third-party SDK, the `Activity` that receives the callback result usually does not set any permissions to ensure this `Activity`'s accessibility.

We extracted the protection levels of custom permissions for protecting the four major components, as shown in Figure 4. As suggested by Guideline 5, the protection level should be `signature` or `signatureOrSystem`[5]. According to our statistics, most configurations are correct, say `Activity`: 92.61%, `Service`: 86.54%, `Receiver`: 94.33%, and `Provider`: 96.52%. A few developers use `normal-level` permissions to protect components, which is insecure. A `normal-level` permission can be obtained by any app. The reason is mainly due to the default parameters of the `protectionLevel` attribute of the `<permission>` element. If developers do not explicitly specify the protection levels of defined custom permissions, the default level will be assigned as `normal`. Developers may believe that the default protection level is secure to protect components. Also, some developers use `dangerous-level` custom permissions to protect components, which is also not recommended by Google. Creating `dangerous-level` permissions will introduce user decisions, and users may not have enough security knowledge to make appropriate decisions.

For `ContentProvider`, Google has implemented fine-grained permission control, dividing it into read-only or write-only permissions. 50.28% (8795/17493) usage cases apply read-only permission to `ContentProvider`, and 14.51% (2539/17493) usage cases apply write-only permission. However, 35.21% (6159/17493) usage cases still do not implement this find-grained protection, and apps are granted both read and write permissions, although most apps may only need write or read permissions.

Custom Permission Usage. For custom permissions, 12,492 apps (4.96%) exist the over privilege issue. That is, permission is defined without use. The possible reasons include:

```
<permission android:name="baidu.push.
    permission.WRITE_PUSHINFOPROVIDER.com.
    wzm.moviepic" android:protectionLevel="
    signature"/>
<provider android:name="com.baidu.android.
    pushservice.PushInfoProvider"
android:writePermission="baidu.push.
    permission.WRITE_PUSHINFOPROVIDER.com.
    wzm.moviepic " android:protectionLevel="
    signature" android:exported="true" />
```

Listing 3: Example of inconsistent permission names.

(1) The developer did not check whether the defined custom permissions are consistent with the custom permissions applied to the component, which means there is a spelling error. In addition, some spelling mistakes are so tiny that it is difficult to notice. For example, as shown in Listing 3, the app Shouhui Movie[6] defines a permission[7] to protect its `Provider` component. Note that the permission protection level is `signature`, and other apps with different certificates with Shouhui Movie

[5]In most cases, `signature` and `signatureOrSystem` are equivalent.

[6]Package name: `com.shouhui.moviesdy`

[7]`baidu.push.permission.WRITE_PUSHINFOPROVIDER.com.wzm.moviepic`

cannot access this `Provider`. However, the developer entered an extra space at the end of the permission name carelessly. As a result, from the system's view, the misspelled permission applied to the component above has not been defined. A malicious app can declare the above-misspelled permission name and set its protection level to `normal`, thereby gaining access to the exported component.
(2) The developer forgot to delete the unused custom permissions in time in new versions. For example, the app Tencent Map[8] defined a custom permission[9] protected component `RadioMapProvider`[10] in version 5.0.1. This `Provider` component was deleted in later versions, but the corresponding custom permission declaration was not removed.

Our Assessment: The app component export operations are widespread. Only a small part of the exported components are protected by custom permissions. Among these use cases, most developers can set the protection levels of custom permissions correctly.

V. Discussions

A. Threats to Validity

Some of the threats to validity of this study are discussed in this section.
Static code analysis. In our analysis, we decompiled apps mainly using AndroGuard, a Python-based static analysis tool. Since static analysis does not execute code, some identified behaviors (compliance and violation of guidelines) may not be triggered in apps at runtime. On the other hand, dynamic analysis is unsuitable for large-scale measurements due to its analysis efficiency and code coverage issues.
False negative results. For each guideline compliance and violation, developers may have diverse implementations. Although we have considered most implementation cases, there are inevitably omissions. Even so, we measured 251,749 apps, and the guidelines compliance and violations extracted from such a large number of apps reflect the overall characteristics.
Violation reasons analysis. We utilized various methods to analyze the possible reasons for guideline violations. To validate our analysis and obtain the actual reasons for guideline violations, we need to conduct large-scale app developer studies. As part of this work, we sent out 500 E-mails to app developers for reason confirmations. Unfortunately, we received only 13 responses, so we did not present app developer studies results in this work.

B. Lessons Learned

The reasons for the low adoption rates of Guidelines 3, 4, and 5 are different. The goal of developers is to rapidly implement the functionalities of apps, enabling them to be released quickly to capture market share. Developers will likely continue to make these mistakes unless they receive proper education on these security issues. To increase the adoption rate of these guidelines, we suggest starting with Google and app market maintainers. They can adopt the following suggestions:
(1) It would be beneficial for Google to provide developers with concrete code examples that demonstrate the implementation of complex development operations. By offering practical examples, developers can better understand how to apply the guidelines in their apps smoothly. In addition, adopting a security-by-default design approach for Android Studio can significantly contribute to the improved adoption rate of the guidelines. This means implementing default security measures within the development environment itself, such as enforcing secure coding practices and providing built-in security features.
(2) App marketplace maintainers should design corresponding automated detection tools based on the security guidelines provided by Google and integrate them into the app marketplace's review process to prevent the release of apps that violate the security guidelines.

VI. Related Work

Some previous works focus on the topics of API misuse and developer behaviors. Here we review these works.
API Misuse. Egele et al. [27] evaluated the cryptographic APIs used in Android and found that 10,327 of 11,748 apps had at least one cryptographic misuse based on CryptoLint. Diao et al. [26] analyzed the Android accessibility APIs framework. They found that the accessibility APIs are widely misused, and developers use the accessibility APIs to bypass the permission restrictions of the Android OS. Shao et al. [33] proposed SInpector to check for potential misuse of Unix domain sockets in apps and system daemons. A total of 14,644 apps and 60 system daemons were analyzed, and 45 apps and 9 system daemons were found to be vulnerable. Oltrogge et al. [32] analyzed the use of Network Security Configuration (NSC) in Android, and 88.87% of apps misused NSC to cause security degradation. Luo et al. [31] proposed the MAD-API framework to detect API misuse problems in apps with the evolution of Android APIs. They found that 93.13% of the evaluated apps had API misuse problems, and the total number of misuses reached 1,241,831. Li et al. [30] present ARBITRAR, which detects API misuses by interacting with users. They checked the usage of 18 target API methods in 21 C++ programs and discovered 40 bugs.
Developer Behaviors. Li et al. [30] surveyed the posts on StackOverflow and found the classes that frequently caused usage obstacles but not frequently used, and also revealed some causes of API usage barriers. Zhang et al. [36] found that developers may learn new API usage from online Q&A sites. They analyzed 217,818 StackOverflow posts using ExampleCheck and found that 31% posts may have potential API usage violations. Li et al. [29] found that most of the deprecated APIs are accessed by apps through popular libraries, and library developers are more likely than app developers to update deprecated APIs. Linares-Vásquez et al. [34] analyzed the factors affecting the success of Android apps, measuring

[8]Package name: `com.tencent.map`
[9]`tencentmap.provider.permission`
[10]`com.tencent.navsns.radio.provider.RadioMapProvider`

7,097 Android apps, and found that heavy use of fault- and change-prone APIs negatively affected the success of these apps. In another direction, Linares-Vásquez et al. [35] studied the developers' responses to API changes. When API behavior is modified, Android developers usually have more questions and triggered more discussions on StackOverflow.

Unlike the above research, in this paper, from the perspective of Google's app security development guidelines, we evaluate the deployment status of data security guidelines. To the best of our knowledge, we are the first to conduct an evaluation of the security guidelines provided by Google.

VII. Conclusion

In this paper, we evaluated whether app developers followed the Android official security guidelines and investigated the reasons behind the results. Taking the app data security as the case study, we evaluated five development guidelines for data security design. The high-level result shows that developers did not follow three of these five guidelines well. The possible reasons of guideline violations come from both parties, Google and app developers.

References

[1] (2023) Android Developer Salary - For Freshers and Experienced. [Online]. Available: https://www.simplilearn.com/tutorials/software-career-resources/android-developer-salary

[2] (2023) Android Get Serial Number. [Online]. Available: https://stackoverflow.com/questions/33079734/android-get-serial-number

[3] (2023) Android Permission Protection Level. [Online]. Available: https://developer.android.com/guide/topics/manifest/permission-element

[4] (2023) android:exported. [Online]. Available: https://developer.android.com/topic/security/risks/android-exported

[5] (2023) App Security Best Practices. [Online]. Available: https://developer.android.com/topic/security/best-practices

[6] (2023) Application Sandbox. [Online]. Available: https://source.android.com/security/app-sandbox

[7] (2023) Apply Signature-Based Permissions. [Online]. Available: https://developer.android.com/topic/security/best-practices#apply-signature-based-permissions

[8] (2023) Best Practices Android Identifier. [Online]. Available: https://developer.android.com/training/articles/user-data-ids#best-practices-android-identifiers

[9] (2023) Fixing a Zip Path Traversal Vulnerability. [Online]. Available: https://developer.android.com/topic/security/risks/zip-path-traversal

[10] (2023) How to Decode Android Os.Build.Serial? [Online]. Available: https://stackoverflow.com/questions/11794749/how-to-decode-android-os-build-serial

[11] (2023) Improper Export of Android Application Components. [Online]. Available: https://cwe.mitre.org/data/definitions/926.html

[12] (2023) Mobile App Download Statistics & Usage Statistics (2023). [Online]. Available: https://buildfire.com/app-statistics/

[13] (2023) Over 60% of Android Apps Have Security Vulnerabilities. [Online]. Available: https://atlasvpn.com/blog/over-60-of-android-apps-have-security-vulnerabilities

[14] (2023) Path Traversal Vulnerability. [Online]. Available: https://developer.android.com/topic/security/risks/path-traversal

[15] (2023) Permissions on Android. [Online]. Available: https://developer.android.com/guide/topics/permissions/overview

[16] (2023) Privacy & Security SC-DF2. [Online]. Available: https://developer.android.com/docs/quality-guidelines/core-app-quality#SC-10

[17] (2023) Remove Debug Logging. [Online]. Available: https://stackoverflow.com/questions/2446248/how-to-remove-all-debug-logging-calls-before-building-the-release-version-of-an

[18] (2023) Request Permissions. [Online]. Available: https://developer.android.com/training/articles/security-tips#RequestingPermissions

[19] (2023) Restrictions on non-SDK interfaces. [Online]. Available: https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces

[20] (2023) Share File. [Online]. Available: https://stackoverflow.com/questions/12170386/create-and-share-a-file-from-internal-storage

[21] (2023) Store Data in External Storage Based on Use Case. [Online]. Available: https://developer.android.com/topic/security/best-practices#external-storage

[22] (2023) Store Private Data Within Internal Storage. [Online]. Available: https://developer.android.com/topic/security/best-practices#internal-storage

[23] (2023) Use Resettable Identifiers. [Online]. Available: https://developer.android.com/docs/quality-guidelines/core-app-quality#sc

[24] (2023) Use SharedPreferences in Private Mode. [Online]. Available: https://developer.android.com/topic/security/best-practices#sharedpreferences

[25] (2023) Uzip File. [Online]. Available: https://stackoverflow.com/questions/3382996/how-to-unzip-files-programmatically-in-android

[26] W. Diao, Y. Zhang, L. Zhang, Z. Li, F. Xu, X. Pan, X. Liu, J. Weng, K. Zhang, and X. Wang, "Kindness is a Risky Business: On the Usage of the Accessibility APIs in Android," in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Beijing, China, September 23-25, 2019*, 2019.

[27] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An Empirical Study of Cryptographic Misuse in Android Applications," in *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS), Berlin, Germany, November 4-8, 2013*, 2013.

[28] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein, "Understanding the Evolution of Android App Vulnerabilities," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 212–230, 2021.

[29] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "CDA: Characterising Deprecated Android APIs," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2058–2098, 2020.

[30] Z. Li, A. Machiry, B. Chen, M. Naik, K. Wang, and L. Song, "ARBITRAR: User-Guided API Misuse Detection," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (IEEE S&P), San Francisco, CA, USA, 24-27 May 2021*, 2021.

[31] T. Luo, J. Wu, M. Yang, S. Zhao, Y. Wu, and Y. Wang, "MAD-API: Detection, Correction and Explanation of API Misuses in Distributed Android Applications," in *Proceedings of the 7th International Conference on Artificial Intelligence and Mobile Services (AIMS), Seattle, WA, USA, June 25-30, 2018*, 2018.

[32] M. Oltrogge, N. Huaman, S. Amft, Y. Acar, M. Backes, and S. Fahl, "Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications," in *Proceedings of the 30th USENIX Security Symposium (USENIX-Sec), August 11-13, 2021*, 2021.

[33] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao, "The Misuse of Android Unix Domain Sockets and Security Implications," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria, October 24-28, 2016*, 2016.

[34] M. L. Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "API Change and Fault Proneness: A Threat to the Success of Android Apps," in *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013.

[35] M. L. Vásquez, G. Bavota, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC), Hyderabad, India, June 2-3, 2014*, 2014.

[36] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are Code Examples on an Online Q&A Forum Reliable? A Study of API Misuse on Stack Overflow," in *Proceedings of the 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, May 27 - June 03, 2018*, 2018.

[37] R. Zhou, M. Hamdaqa, H. Cai, and A. Hamou-Lhadj, "MobiLogLeak: A Preliminary Study on Data Leakage Caused by Poor Logging Practices," in *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, February 18-21, 2020*, 2020.