An Empirical Analysis of Hazardous Uses of Android Shared Storage

Shaoyong Du, Pengxiong Zhu, Jingyu Hua ✉, Zhiyun Qian,
Zhao Zhang, Xiaoyu Chen, and Sheng Zhong ✉

Abstract—Android shared storage is shared with all the applications (apps for short) and the user. It is common to see that a large amount of apps store different kinds of files on it. It is well known that apps granted the read or write permissions can freely access any files in the shared storage. As a consequence, the shared storage has been demonstrated to expose sensitive information and jeopardize users' privacy.
In this paper, we systematically study a simple but overlooked threat related to the shared storage — the lack of input validation (e.g., integrity verifications) when consuming files on the shared storage. We argue that the untrusted input from the shared storage is a much ubiquitous problem. By undertaking an empirically study through a static analysis tool we develop, we find over 30% of the 13,746 analyzed popular apps on the market suffer from such problem. By investigating the types of files consumed, we find shockingly a large fraction of apps store and consume sensitive files, which allows us to construct end-to-end attacks. Considering the ubiquity of this class of vulnerabilities, we finally define better access control policies for external storage to eliminate them for most apps.

Index Terms—Shared Storage, Android, Data Security, Static Analysis, Integrity Verification, Attacks.

1 Introduction

With the storage volume of smartphones increased year by year, the apps tend to store more and more different files locally to enrich the user experience. Android, one of the most popular mobile operating systems to date, provides apps with two kinds of ways to save the files on the users' smartphones. One way is to save the files in the private directory specified by Android, where the strict access control and isolation policies are enforced to guarantee that each app can only access its own data. The other way is to use the shared storage that is shared with all apps and the user. Once an app is granted to access the shared storage, it can manipulate any files in the shared storage. Though the first way guarantees the security of the data, it is still common to see that the apps save a large amount of data on the shared storage. The shared storage also can be expanded by the extra TF card or U disk. According to Android development documents [1], they call the shared storage *external storage*, while the private directory as *internal storage*. In the following parts, we also use the item *external storage* to refer the shared storage and use the item *internal storage* to refer the private directories in the first way.

It is well known that the lack of access control in external storage can have security and privacy implications. Researchers have reported that much data kept in the external storage contain sensitive information such as the raw voice messages stored by some social networking apps [2], [3]. Once granted with the permissions to access the external storage, the attacker can not only read but also manipulate any files that he wants on external storage. In this way, the files on external storage should be considered untrusted. Prior work [4], [5], [6] has reported vulnerabilities related to manipulations on executable files, .apk, .so and .py files. However, several studies only point to instances of vulnerabilities [5], and it is unclear how widespread the problem is. In addition, it is unclear what other types of files are stored on external storage and what vulnerabilities they entail. In this paper, instead of just focusing on specific types of files, we conduct a comprehensive survey on the untrusted files stored on external storage and analyze their security impact.

Compared to many other sources of untrusted input such as socket [7] and Intent [8], we argue that the problem of "untrusted input from external storage" is an elephant in the room. Fortunately, to alleviate this security concern, Google does offer two concrete suggestions to developers [9]: (1) sensitive files should not be stored on external storage; (2) if developers have to do so, the file content needs to be validated prior to consumption (e.g., files can be signed cryptographically and verified). Unfortunately, as history shows us, when security is a responsibility of individual developers (instead of being addressed at the system level), mistakes are bound to happen [7], [10], [11]. In this paper, we

• *This work was supported in part by NSFC under Grant 61425024, Grant 61402223, and Grant 61321491, in part by the Jiangsu Province Double Innovation Talent Program, and in part by the Fundamental Research Funds for the Central Universities. The work of S. Du was supported by the Program B for Outstanding PhD Candidate of Nanjing University under Grant 201701B026. The work of J. Hua was supported by NSFC under Grant 61300235. The work of Z. Qian was supported by NSF under Grant CNS-1619450.*

• *S. Du, J. Hua, Z. Zhang, X. Chen and S. Zhong are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210023, China, and also with Department of Computer Science and Technology, Nanjing University, Nanjing, 210023, China.*
E-mail: shaoyong.du.cs@gmail.com, huajingyu@nju.edu.cn, ericzz0727@gmail.com, traumcxy@163.com, zhongsheng@nju.edu.cn
• *P. Zhu and Z. Qian are with the Department of Computer Science and Engineering, University of California, Riverside, CA, 92521.*
E-mail: pzhu011@ucr.edu, zhiyunq@cs.ucr.edu
• *Both of J. Hua and S. Zhong are the corresponding authors.*

set out to study empirically how well these two suggestions are implemented by developers. Specifically, we attempt to answer the following questions:

- What types of files are placed on external storage? Are they sensitive or not? Who places these files and who consumes them?
- Do apps perform any input validation? Are they effective (possible to bypass)?
- What are the consequences of consuming unvalidated files? Is it straightforward to exploit such vulnerabilities?

To answer the questions in the first two bullet points, we develop a static analysis tool *ExInspector*. Its main goal is to check an app's usage of external storage, e.g., what types of files are read, how they are consumed, and whether input validation is performed. Although seemingly trivial, there are a few technical challenges. For instance, we find that it is not easy to distinguish external storage read operations from the internal ones (as the open and read APIs are the same). This means that one will need to use the path parameter as indication, which is often dynamically constructed. Sometimes part of the file path could even be stored in an internal database, which further complicates things. We are able to overcome these challenges by reconstructing or inferring the origin of the file path (recursively when part of the file path also requires reconstruction or inference). We use our tool to scan 9,889 apps from Google Play and 3,874 apps from a third-party market APKPure.com [12]. The results show that a large fraction of developers indeed store sensitive files such as .apk, .html, .js, and voice message files on external storage, which violates Google's first suggestion. The tool is also able to understand and characterize a set of known input validation techniques employed by these apps. Our finding is that for all the file types, there are always more than 90% of the apps not performing any input validations prior to consuming the untrusted data, which means Google's second suggestion is also being violated by a large fraction of developers.

To answer the questions in the last bullet point, we sample apps that consume a variety of types of files and build targeted exploits against them. As we will discuss later in detail, besides being useful for storing data and sharing them (e.g., pictures), external storage is often used as a communication channel either between different app components or across apps. Interestingly, we also find that many such communication are transient, i.e., files are read immediately after they are written, therefore leaving a very small time window for tampering. For instance, third-party market apps may temporarily store a downloaded apk file on external storage and pass it immediately afterwards to the system for installation. We are able to overcome these challenges and construct reliable exploits by leveraging appropriate file replacement techniques. We conduct both large-scale analysis of apps to understand the overall landscape of such threats and detailed case studies that demonstrate end-to-end attacks such as impersonating a friend in a voice chatting session in the context of instant messenger apps, replacing downloaded apk files to install phishing software without the user's knowledge and replacing web resources passed to WebViews to show phishing pages. Attack demos can be found on our anonymous project website https://sites.google.com/site/externalstorageattacks/.

In summary, we make the following contributions:

- We study empirically an overlooked threat of data tampering against files on Android external storage. Based on a static analysis tool we develop, we are able to understand how well Google's security guidelines are followed by app developers in practice.
- After scanning 13,746 popular apps on the market, we find over 30% of them suffer from the overlooked threat. Various sensitive files are indeed stored on external storage by a large fraction of apps. We also report most read operations are not protected by any input validation. Even when input validation is present, it can often be bypassed. What's worse, by tracing the same set of apps in the last three years, we show that things have not changed and this problem is still being seriously overlooked.
- We demonstrate the consequences by constructing several serious exploits against a number of popular apps overcoming practical challenges such as attack timing. We also suggest several practical countermeasures against this class of attacks.

2 Background

We discuss the background of storage system in Android and its security concerns in the context of Android security model.

2.1 Android Security Model – Untrusted Input

In Android, there are a multitude of ways an app can face untrusted input. This ranges from network input, input from Inter-Process Communication (IPC) channels such as Intent and Unix domain socket, as well as input from files. To date, a number of research studies have shown that apps suffer from untrusted input through network sockets and open ports [13], Intent [8], [14], [15], Unix domain socket [7]. However, shared external storage is also a form of untrusted input that has not been scrutinized. Interestingly, Android-specific communication channels such as Intent and Broadcasts are much more heavily studied in terms of their security implications compared to the native Linux communication channels such as file systems, which is the focus of this paper. Our hypothesis is that app developers are generally not accustomed to the idea of verifying input from file systems (including external storage) even when Google does make such a recommendation [9].

2.2 Storage Management of Android

Just as we have talked in § 1, Android-compatible devices offer apps two kinds of ways to locally store the files. One is to keep the files in its private directory (i.e., "*/data/data/App Package Name/*") specified by Android. With the strict access control and isolation policies, the stored files can be only manipulated by their owners. However, when the app is removed, the associated private directory will be deleted, too. Different from the above one, the shared storage makes it possible for apps to permanently keep the files, even when the apps are removed. Meanwhile, an app can also share the files with other apps through it. For instance,

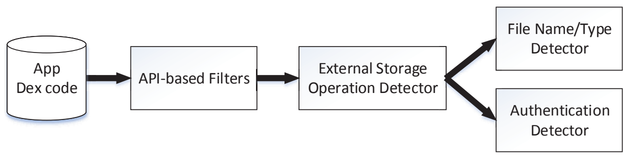


Fig. 1. Overview of ExInspector

pictures taken by a camera app can be stored in the shared storage and a photo editing app can subsequently read the files. Users may use the removable storage media (e.g. TF card) to increase the volume of the shared storage. In the remaining paper, we do not distinguish these two types of shared storage, as they share the same security policy and only differ in their file paths. Moreover, the default shared storage location depends on the user's settings.

Android has two permissions to manage the access to the shared storage, *android.permission.READ_EXTERNAL_STORAGE* and *android.permission.WRITE_EXTERNAL_STORAGE* for complete read and write access respectively. Since Android 4.4, even without the permissions mentioned above, apps are allowed to read and write its own files in the shared storage "*/sdcard/Android/data/App Package Name*" [1]. Note that these files are not private as they are exposed to apps that have read or write permissions granted, and they will be deleted when the app is removed.

2.3 Threat Model and Capability Analysis

Shared external storage can be accessed through apps that are installed on an Android device, as well as any authorized PC that attaches to the device. For the purpose of this paper, we assume that the threat comes from a malicious app co-located with a victim app. The malicious app needs to acquire proper permissions in order to tamper with files on the external storage.

It is important to also test what malware can do when it has read/write access to the external storage. In the Linux world, for instance, if an under-privileged program can write to a directory in which a privileged program will read, there are various name resolution attacks that can be carried out [11], e.g., replacing the file with a symbolic/hard link to a sensitive file that the privileged program will read on behalf of the under-privileged program (confused deputy attack). On Android, however, symbolic/hard links are explicitly disabled on external storage [16]. Therefore, the only attack vector here is to modify the file content and cause victim apps that read the file to behave in undesirable ways.

3 ExInspector Design and Implementation

The goal of ExInspector is to examine the use of external storage in apps, and identify those that are most likely vulnerable for further analysis. In this section, we describe our design and implementation of ExInspector. ExInspector is built on Soot [17] to analyze the apk files. Fig. 1 shows the system modules and overall analysis steps of ExInspector. First, ExInspector will screen apps through an API-based filter and check whether an app will ever access external storage. We then further conduct analysis to confirm that an app indeed reads/writes the files on external storage (by reconstructing the file access path). After that, we will conduct further analysis to answer the two questions we posed in the beginning. First, we will understand what file types are stored on the external storage and how the files are consumed — this will help us infer whether the files are sensitive. Second, we will check if the apps perform input validation before consuming the file — this will allow us to discover potentially vulnerable apps.

3.1 Identifying File Accesses on External Storage

It turns out that it is not easy to distinguish external storage file operations from the internal ones (as they share the same set of open and read/write APIs). As a result, it is necessary to evaluate or infer value of the path parameter that gets passed to the open or read/write APIs such as `java.io.FileInputStream openFileInput(java.lang.String)`. The evaluation of this parameter could be complex where multiple statements across different functions need to analyze.

ExInspector starts out by locating the read/write APIs invoked by the app, e.g., APIs in the *java.io* package. Once such an API is found, if the path parameter is not a constant, it will perform a backward inter-procedural dataflow analysis to find all the variables that contribute to the value of the path parameter. We will introduce the details of this process later. Here, we just emphasize that this is a "may analysis". In other words, we say that the path is from external storage as long as one of the backward execution chains satisfies the following conditions:

C1. *Reaches a definition that is a constant assignment indicating external storage.* Once a definition is a constant that begins with strings such as "/sdcard", we will know that it comes from external storage.

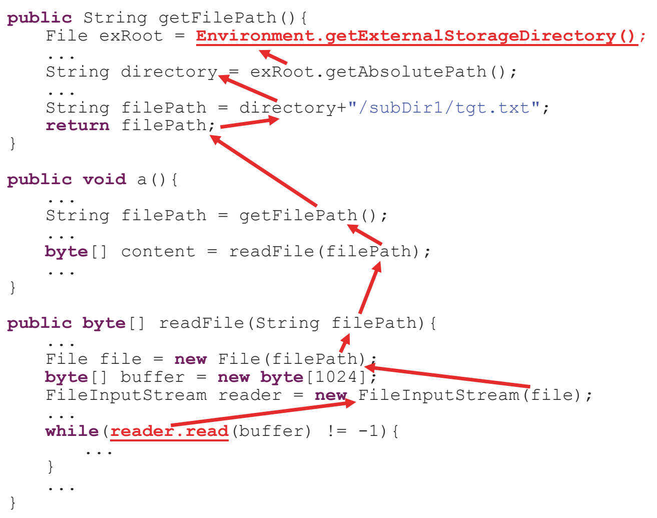
C2. *Reaches a definition whose return value is from a specific method provided by Android, listed in Table 1.* Examples include return values of `getExternalFilesDir()`. If this condition is satisfied, we can directly identify whether the path comes from the external storage.

This backward dataflow analysis is illustrated in Fig. 2. In this example, C2 is satisfied because ultimately it leads to an API that returns an external storage path. Note that the path at some point is assembled from two different values: `directory` and "`/subDir1/tgt.txt`". If we are interested in locating only the origin of the path (internal or external), it is sufficient to backtrack to `directory` alone. However, as we are also interested in knowing what type of files are being produced or consumed, we also need to resolve the second part of the file path (to identify the file name suffix).

One serious challenge is that sometimes the backtracking of the file path will lead to more complex sources such as a SQL query result from a table. Typically, this occurs when an app writes a file to internal/external storage and wants to keep track of them. In such cases, we'll need to model the database or key value stores (e.g., *android.content.Intent*, *android.os.Bundle* and *android.content.SharedPreferences*) and trace its real value further. For instance, as we show in Fig. 3, when we encounter the get function `Intent.getStringExtra()` during the

TABLE 1
List of APIs that access the internal and external storage

Method / Storage Class	Internal Storage	External Storage
android.content. Context	String[] databaseList() String[] fileList() File getCodeCacheDir() File getDataDir() File getDir(String name, int mode) File getDatabasePath(String name) File getCacheDir() File getFileStreamPath(String name) File getFilesDir() File getNoBackupFilesDir() FileInputStream openFileInput(String name) FileOutputStream openFileOutput(String name, int mode)	File getExternalCacheDir() File[] getExternalCacheDirs() File getExternalFilesDir(String type) File[] getExternalFilesDirs(String type) File[] getExternalMediaDirs() File getObbDir() File[] getObbDirs()
android.os. Environment	File getDataDirectory() File getRootDirectory()	File getDownloadCacheDirectory() File getExternalStorageDirectory() File getExternalStoragePublicDirectory(String type)
android.support. v4.content. ContextCompat	File getCodeCacheDir(Context context) File getDataDir(Context context) File getNoBackupFilesDir(Context context)	File[] getExternalCacheDirs(Context context) File[] getExternalFilesDirs(Context context, String type) File[] getObbDirs(Context context)

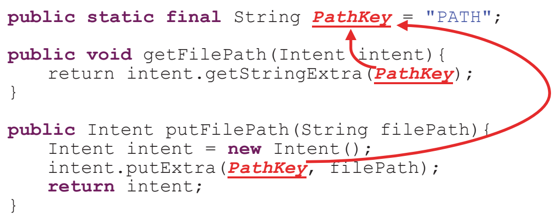


```
public String getFilePath(){
    File exRoot = Environment.getExternalStorageDirectory();
    ...
    String directory = exRoot.getAbsolutePath();
    ...
    String filePath = directory+"/subDir1/tgt.txt";
    return filePath;
}

public void a(){
    ...
    String filePath = getFilePath();
    ...
    byte[] content = readFile(filePath);
    ...
}

public byte[] readFile(String filePath){
    ...
    File file = new File(filePath);
    byte[] buffer = new byte[1024];
    FileInputStream reader = new FileInputStream(file);
    ...
    while(reader.read(buffer) != -1){
        ...
    }
    ...
}
```

Fig. 2. An example of backward tracking a target variable



```
public static final String PathKey = "PATH";

public void getFilePath(Intent intent){
    return intent.getStringExtra(PathKey);
}

public Intent putFilePath(String filePath){
    Intent intent = new Intent();
    intent.putExtra(PathKey, filePath);
    return intent;
}
```

Fig. 3. An example of tracking file paths through the key-value store

backward analysis, we will need to first resolve the "key" (`PathKey` in the example), which will recursively trigger the same backward dataflow analysis. More importantly, we need to find out what "value" corresponds to that key. To this end, we search through the function calls that contain put function invocations (`intent.PutExtra()` in the example) to understand what value can be stored corresponding to the same key. Similarly, for database operations, we attempt to resolve what file path has been written by correlating the database read/write operations (e.g., they have to be operating on the same table).

3.2 Identifying File Types and Consumers

Knowing there are file accesses in external storage is only the first step. An app can read different kinds of files for different purposes — could be as simple as loading an image or as sensitive as installing an apk file. To find out the "sensitive" uses of "sensitive" files, we set out to investigate the file types and their consumers.

We have already discussed how to reconstruct the file path (including the file name and its suffix) which can be a good indication of the file type. However, there can be two issues. Firstly, the suffix is not always reliable. Secondly, the same file type can still lead to different uses: some may be sensitive, while some may be not. Therefore, besides file suffix, we also take the consumers of the files into consideration. Specifically, there are well-known Java APIs to handle different kinds of files. For instance, zip files are typically processed by input stream obtained by `getInputStream()` of *java.util.zip.ZipFile*. Such APIs are indicative of the file type even when the file suffix is not zip or jar. By the way, we find in some cases that identifying file consumers may help us to understand how the data will be used and further determine whether such uses are sensitive or not. For instance, if the consumer of an apk file is the PackageManager API to install apps, we can safely conclude that the use of this file is sensitive. However, in many other cases (e.g., scores of APIs are used to read text/config files, but do not specify the "sensitivity" of

TABLE 2
Examples classes of consumers

Class	File Type
java.util.Properties	properties
android.graphics.BitmapFactory	image
android.media.SoundPool	audio
android.media.MediaPlayer	audio/video
android.widget.VideoView	video
android.webkit.WebView	html/js
android.database.sqlite. SQLiteDatabase	database
org.sqlite.database.sqlite. SQLiteDatabase	database

the config). Understanding those types of consumers will require substantial domain knowledge of an app, which indicates it is not straightforward to use them to identify the "sensitive" uses of data. In our following evaluation, we totally discovered 122 final file consumers. About 39.34% of them are unambiguously sensitive.

To this end, we perform a static taint analysis that file data as taint and any APIs that subsequently process the data as sinks (we omit the details of the methodology as it is fairly standard). The outcome is a list of APIs (sinks) and their corresponding Java classes that have processed the data. As will be discussed later, we report a common set of Java classes (that process file data). Table 2 gives some examples. Note that there may be very much custom-written data consumers (especially for custom data), which we may not characterize well (e.g., a configuration file specific to an app). Understanding those types of consumers will require substantial domain knowledge of an app.

3.3 Identifying Input Validation Checks

Once we know where an app performs readings on the external storage, we need to check whether there are any input validation surrounding them. As Google suggested, any data read from the external storage should be considered untrusted. Here we do not consider application-layer checks as they are typically not meant for security. For instance, an image reader app may check whether an image file conforms to its expected/supported format. For such checks, an attacker can likely create a malicious file that satisfies the format requirement.

The checks we consider are mostly concentrated on validating the integrity of the files that were produced by the same app. For instance, an app that initially created this file may store its corresponding SHA256 hash in its internal storage (computed on the file in memory), which then can be cross-checked with the actual file read later. After analyzing and characterizing a small handful of apps, we summarize the most common behaviors (the complete set of checks we consider is in Table 8).

- *Prior Verification:* The main checks are on the file's size and last modified time, through standard APIs such as `java.io.File.length()`, `java.io.InputStream.available()` for file sizes, and `java.io.File.lastModified()` for last modified time. If the information recorded in the internal storage does not match the ones from the actual file about to be read, then the integrity check will fail and the file will not be read.
- *Post Verification:* The main checks are regarding to the file's content. For instance, they could verify the `digest()` provided by *java.security.MessageDigest*, to check whether the file's content has been modified. This type of integrity check is typically strong and not easy to bypass. Meanwhile, they can also verify the checksum of a data stream, by *java.util.zip.CRC32* and *java.util.zip.Adler32*.

Note that the first class of checks based on size and last modified time can be potentially bypassed, as it is not particularly difficult for an attacker to create malicious files satisfying both requirements. In Android, the precision of the file's last modified time is truncated from millisecond to second. It provides the attacker with a chance to make the malicious file's value the same as the original one's, if the attacker manages to tamper with the file as it is being written. In contrast, the hash based check is much harder to bypass as hash collisions are usually hard to produce. There are caveats however in implementing these checks: (1) the stored integrity information has to be in the internal storage; otherwise an attacker can easily tamper with them. (2) The initial integrity information needs to be computed from a trusted source (ideally in memory before it persists to external storage).

To identify integrity checking logics around the external read operations, we exhaustively search through the statements between the open and read operations for prior verification, and between read and last consumption of the data for post verification. Sometimes, it is possible that these checks are not directly called by the same method as the one containing the read invocation.

4 Limitations

In fact, there are still some limitations in our analysis. Firstly, our analysis does not handle native code at the moment. It is entirely possible that external storage operations are performed only in native code and therefore will be missed. When we ignore the application-layer checks in § 3.3, we may also miss some logical input validations that an app uses to check some key metadata values in a file. However, its impact on our analysis result can be small mainly in that the logical input validation cannot guarantee the integrity of the whole data read and as a result attackers still can tamper with some metadata values that are used without being checked. When we rebuild the file path, the encoded file path is not considered, which also can result in missing some external storage operations. In addition, as our analysis largely depends on the knowledge of APIs for identifying file operations and input validations, it is possible the incompleteness will lead to missing certain app behaviors that invoke APIs outside of the common set we identify in our analysis. Our analysis tool does not analyze any metadata values in the file (e.g., in json/xml files) that may contain integrity-sensitive application configurations/account details. To better understand the content of these files, a significant manual effort is required. Therefore, we leave it out for our future work. Since our analysis only considers standard Java libraries used for certain types of

TABLE 3
Summary of analyzed apps

Total Num	Apps reading from external storage (%)	Apps writing to external storage (%)	Vulnerable apps (%)
13,746	4,799 (34.91%)	5,612 (40.83%)	4,700 (34.19%)

input validation, we may miss some other input validation methods implemented by third parties. Finally, since the dataflow analysis that we can conduct is a "may" analysis and can generate false positives, some program paths may not actually be feasible.

5 Evaluation

We obtained the apk files from two app markets: 9,899 apps are from Google Play and 3,847 apps are from a third-party app market APKPure.com [12]. All of the apps are top free apps in each category (we spread the apps roughly evenly across categories). We now apply ExInspector to analyze the downloaded apps. As we mentioned earlier, this tool may have to reconstruct the file path to distinguish external storage read operations from the internal ones or identify file types. The real experiments show 4,764 apps involve such path reconstruction.

Apps that read/write files on external storage. Before we apply ExInspector to the downloaded apps, we conduct a survey about the permissions declared in *AndroidManifest.xml* by each app, to get a sense on how many apps "claim" that they need read or write access to external storage. We find that there are more than 60% apps that have read or write access (although they may or may not actually be exercised due to permission overclaim [18]). As it is such a commonly requested permission, users may ignore the potential security risks when installing apps. As we show in Table 3, according to the scan of ExInspector, we are able to detect 34.91% apps reading from external storage and 40.83% apps writing to external storage. It is possible that we have missed some apps that do not use regular read/write APIs, which we plan to include in our future work.

File types. We first show the identified types of files read by apps from external storage. The statistics are shown in Table 4. As we can see, it is very diversely containing a few dozens of common file types — the file name suffix (.txt, .jpg, etc) and the corresponding number (and percentage) of apps that read them. Below we focus on a few sensitive file types:

Multimedia files. We can find that various multimedia data (images, audio files, videos, etc.) are the most common files read from the external storage. In particular, the percentage of apps reading image files reaches 43.95% (accumulating the statistics of all image types) and other multimedia files (e.g., audio and video) take more than 22%. This may be because that the apps just want to permanently store these multimedia files or want to share them with other apps. Nevertheless, multimedia data not only contain significant amount of sensitive information, but are also subject to tampering. For instance, as will be described later, many popular instant messaging apps (e.g., WeChat and WhatsApp), which support voice messages, directly save audio messages onto external storage. This allows a malicious app in the background to effectively hijack the communication.

Apk and related files. As prior work [6] states, apk file replacement can be completely transparent and not visible to users. In a successful attack, a user may be tricked into using a fake or phishing app, thinking that he or she is using a legitimate app. However, it is unclear if the vulnerability still occurs and is prevalent today. From our latest study though (shown in Table 4), we show that besides apk files, obb (opaque binary blobs) files, apk expansion files, are also commonly consumed yet not widely known (they were designed as a workaround for Google Play's size limit on apk files). Surprisingly, we can see that there are 400 (8.34%) apps storing the apk files on the external storage and 362 (7.54%) apps storing obb files, totaling 16% of analyzed apps! To understand what obb files contain, we sample 104 obb files from our dataset. The detailed information about files contained in obb files can be found in Table 5. In summary, obb files also contain multimedia files as well as .html, .dex, .so files which can alter the behavior of the app as well and are as dangerous as apk files. Of course, we still need to check if integrity check is performed when the apk and related files are consumed.

Executable files. In addition, from Table 4, we can see that 59 apps read js (Javscript) files and 45 apps read html files, which are usually cached or prefetched web resource files. An attacker may tamper with them to inject malicious Javascript code or create phishing pages (such attacks are demonstrated in § 6.4). Also, there are 27 apps reading jar files. The tampering of this kind of files can lead to serious security problems as they can be loaded dynamically into the victim's address space [6].

Json/xml files. Different from the executable files, there are some kinds of files (e.g., xml files and json files) that help app developers to store their own data in the customized formats defined by themselves. In Table 4, we can see that a fair amount of apps read these kinds of files, i.e., 317 apps read xml files and 213 apps read json files. It is easy for attackers to follow the customized formats to tamper with these files. Once the tampered files are successfully parsed by the original app, attackers can influence the original app's normal actions. For example, we find that a news app, ZAKER, stores each piece of news in a separated json file. Each time a user clicks to read an item of news, the app will read and parse the targeted json file. By tampering with the data attributes/values in this file, attackers can replace the text and images (to publish fake news or launch phishing attacks) or alter the web resource addresses (i.e., URLs) to trick the app into downloading different web resources from a malicious server.

Others. Besides above, it is not hard to imagine that other tampered file types can wreck havoc. For instance, .config and .properties files can potentially alter the behavior of a victim app. .bak and .cache files may cause a victim app to restore or load tampered critical data.

File consumers. Besides file types, we also use ExInspector to identify file consumers. We present the de-

TABLE 4
Types of files read from external storage

File Type	App Num (%)	File Type	App Num (%)	File Type	App Num (%)	File Type	App Num (%)
unknown file[1]	2436 (50.76%)	.bin	134 (2.79%)	.ini	34 (0.71%)	.ncl	16 (0.33%)
image file[1]	1201 (25.03%)	.debug	119 (2.48%)	.info	30 (0.63%)	.3gp	15 (0.31%)
.jpg	615 (12.82%)	.db	86 (1.79%)	.nomedia	27 (0.56%)	.appodeal	15 (0.31%)
.zip	573 (11.94%)	.mmsyscache	61 (1.27%)	.jar	27 (0.56%)	.result	14 (0.29%)
media file[1]	420 (8.75%)	.properties	61 (1.27%)	.dk	25 (0.52%)	.sqlite	14 (0.29%)
.txt	418 (8.71%)	.js	59 (1.23%)	.bugsense	22 (0.46%)	.m4a	13 (0.27%)
.apk	400 (8.34%)	.mp3	56 (1.17%)	.wav	21 (0.44%)	.ba	13 (0.27%)
.obb	362 (7.54%)	properties file[1]	56 (1.17%)	.referrer	21 (0.44%)	.bak	13 (0.27%)
.dat	355 (7.40%)	.mp4	55 (1.15%)	.udid	21 (0.44%)	.raw	13 (0.27%)
.xml	317 (6.61%)	.adobepassdb	52 (1.08%)	.uid	21 (0.44%)	.gif	12 (0.25%)
video file[1]	292 (6.08%)	.tmp	50 (1.04%)	.conf	20 (0.42%)	.csv	11 (0.23%)
.png	264 (5.50%)	.log	48 (1.00%)	.jpeg	19 (0.40%)	.bdi	11 (0.23%)
.cfg	249 (5.19%)	.cuid	47 (0.98%)	.data	16 (0.33%)	.pdf	11 (0.23%)
audio file[1]	214 (4.46%)	.config	45 (0.94%)	.aerserv	16 (0.33%)	.amr	10 (0.21%)
.json	213 (4.44%)	.html	45 (0.94%)	.temp	16 (0.33%)	.devel	9 (0.19%)
.adId	166 (3.46%)	database file[1]	41 (0.85%)	.ser	16 (0.33%)	.download	9 (0.19%)

[1] These files can have no suffixes but we can infer their type through their consumers.

TABLE 5
Types of files contained in the obb files

File Type	App Num (%)	File Type	App Num (%)	File Type	App Num (%)	File Type	App Num (%)
.png	37(35.58%)	.dat	5(4.81%)	.ANDROID	2(1.92%)	.fnt	2(1.92%)
.jpg	29(27.88%)	.dz	4(3.85%)	.IPAD	2(1.92%)	.fsh	2(1.92%)
.xml	26(25.00%)	.eifo	4(3.85%)	.IPHONE	2(1.92%)	.glsl	2(1.92%)
.split	24(23.08%)	.idx	4(3.85%)	.MF	2(1.92%)	.iml	2(1.92%)
.mp4	19(18.27%)	.ifo	4(3.85%)	.RSA	2(1.92%)	.m	2(1.92%)
.assets	18(17.31%)	.jpeg	4(3.85%)	.SF	2(1.92%)	.m4v	2(1.92%)
.txt	17(16.35%)	.otf	4(3.85%)	.a	2(1.92%)	.neo	2(1.92%)
.mp3	13(12.50%)	.resS	4(3.85%)	.aac	2(1.92%)	.pdf	2(1.92%)
.ttf	12(11.54%)	.sqlite	4(3.85%)	.arsc	2(1.92%)	.pvr	2(1.92%)
.DS_Store	10(9.62%)	.syn	4(3.85%)	.bat	2(1.92%)	.py	2(1.92%)
.html	10(9.62%)	.xoft	4(3.85%)	.bin	2(1.92%)	.s	2(1.92%)
.resource	9(8.65%)	.PNG	3(2.88%)	.css	2(1.92%)	.skin	2(1.92%)
.wav	8(7.69%)	.conf	3(2.88%)	.csv	2(1.92%)	.so	2(1.92%)
.json	7(6.73%)	.ogv	3(2.88%)	.db	2(1.92%)	.strings	2(1.92%)
.gif	6(5.77%)	.plist	3(2.88%)	.dex	2(1.92%)	.sub	2(1.92%)
.ogg	6(5.77%)	.zip	3(2.88%)	.dtd	2(1.92%)	.svg	2(1.92%)

tected consumers (i.e., classes that process the read data) as well as the corresponding number (and percentage) of apps in Table 6. The consumers are ranked by popularity. Consistent with the results of file type analysis, we find the most popular consumers are those classes that deal with multimedia data (top three consumers are all such cases). Interestingly, the fourth-placed consumer is *android.content.pm.PackageInstaller*, which is the system service responsible for reading and installing apk files. The fact that 395 apps call this service demonstrates from a different angle that the apk replacement attack is widely applicable. The results also show that 90 apps invoke *android.webkit.WebView*, which is to load and render webpages. In addition, some consumers such as *java.net.HttpURLConnection* will fetch or upload data according to the untrusted data stored on external storage. A malicious app could modify these files to make the victim app to retrieve the incorrect data and thus lead to dangerous behaviors (e.g., replacing the login authentication URL to a URL pointing to a fishing server).

Based on the analysis above, we can get the first conclusion that ***a large fraction of developers do not actually follow Google's first suggestion about not storing sensitive data on external storage***.

Input validations. We next apply ExInspector to answer the second question: do apps perform input validation? Table 7 presents the results for each file type. In particular, each cell counts the number of apps that contain at least one read operation of the specific file type that is protected by the corresponding input validation method. Accordingly, the "none" column counts the numbers of apps that contain one read operation without any input validation. The denominator to calculate each percentage value is the total number of apps that read the corresponding file type. Note that the sum of the percentages in each row may exceed 100%.

The results are really not very encouraging. For all the file types, there are always more than 90% of the apps not performing any input validations prior to consuming the

TABLE 6
Consumers of files read from external storage

Consumer Class	App Num (%)	Consumer Class	App Num (%)
android.graphics.BitmapFactory	1677 (34.94%)	android.util.Log	33 (0.69%)
android.media.MediaPlayer	498 (10.38%)	java.util.zip.CRC32	33 (0.69%)
android.widget.VideoView	494 (10.29%)	android.content.SharedPreferences$Editor	30 (0.63%)
android.content.pm.PackageInstaller	395 (8.23%)	java.net.HttpURLConnection[1]	27 (0.56%)
java.io.ByteArrayOutputStream[1]	361 (7.52%)	org.json.JSONArray	26 (0.54%)
java.io.StringWriter[1]	301 (6.27%)	com.google.gson.Gson	25 (0.52%)
android.media.SoundPool	227 (4.73%)	android.widget.TextView	21 (0.44%)
java.security.MessageDigest	214 (4.46%)	java.lang.String[1]	19 (0.40%)
java.util.Properties	206 (4.29%)	java.nio.ByteBuffer	15 (0.31%)
*.SQLiteDatabase	150 (3.13%)	android.content.ContentResolver[1]	14 (0.29%)
android.text.TextUtils	122 (2.54%)	android.content.ContextWrapper[1]	14 (0.29%)
org.json.JSONObject	107 (2.23%)	android.util.Xml	10 (0.21%)
android.webkit.WebView	90 (1.88%)	java.util.StringTokenizer	10 (0.21%)
java.io.IOException	83 (1.73%)	java.util.regex.Pattern	10 (0.21%)
android.os.Environment[1]	71 (1.48%)	javax.crypto.CipherOutputStream	8 (0.17%)
java.lang.Character	55 (1.15%)	javax.crypto.Cipher	8 (0.17%)
android.content.Context[1]	35 (0.73%)	com.google.a.k	8 (0.17%)
java.net.URLConnection[1]	34 (0.71%)	java.io.ByteArrayInputStream	8 (0.17%)

[1] These classes may propagate the untrusted file content further.

untrusted data. This percentage even exceeds 99% for media files and property files, respectively. This indicates the validations of these two kinds of files are mostly ignored, likely due to perception that they are not really sensitive (which unfortunately is a big mistake as our attack demonstrates in § 6.2). We also collect statistics for each validation method in Table 8. We can see that there are only a small fraction of apps that perform verifications. Prior verification is more widely used compared with the post verification although its absolute percentage is small as well. The message digest is the most adopted post verification method, and is mainly used by ZIP files (7.76%) and TXT files (6.28%). Interestingly, we are unable to find any apps that adopt the last three post verification methods. Unfortunately, even for extremely sensitive files such as apk and html files, the verification is consistently lacking (not to mention that many verifications can be bypassed as will be discussed in § 6). If we consider apps that adopt no input validations or adopt only prior validations as vulnerable apps, their percentage reaches 34.19% as we show in Table 3.

Note that our analysis only considers standard Java libraries used for certain types of input validation. Based on the above analysis, we can come to the second conclusion that ***many developers do not use these standard Java libraries to validate the file content before using them***. Although an app may check some metadata values it reads from a file by its own logical input validations, it does not take every metadata value in the file seriously. As a result, some metadata values are still used without being checked, which still does not follow Google's second suggestion.

Accuracy of input validation. All above results are obtained through ExInspector. To further evaluate its accuracy, we perform some limited manual validations of its outputs. Specifically, we randomly sample 320 apps from our app dataset to check whether the verification methods are correct. We focus on four metrics: true positive rate (TPR), true negative rate (TNR), false positive rate (FPR) and false negative rate (FNR). The results are show in Table 9. We can see that both TPR and TNR are above 88% while the FPR is lower than 3% and the FNR is about 10%, which indicates that our tool achieves a fairly reasonable accuracy.

Vulnerable apps. Combining the results from the file type analysis, consumption, and input validation, we want to understand a more complete picture of how many apps are vulnerable to consuming unvalidated files. We specifically look at a few file types:

Audio files. To generalize the results, we conduct a focused study on instant messaging apps which may be vulnerable to hijacking. In total, we analyze 40 communication apps that have been downloaded for *50,000,000+* times. We find that 19 apps support voice messages and 15 of them directly save the audio files on the external storage. The audio files are stored in various formats such as SILK [19], AMR [20], AAC [21] and OPUS [22]. Interestingly, compared to text messages which are stored exclusively in *internal storage*, most of voice messages are stored in *external storage*.

Apk files. Further, to automatically identify the app installation behavior (and the possibility of apk replacement attack), we ask ExInspector to also look for additional intent passing behavior (to installation service) of the data type *"application/vnd.android.package-archive"*. We investigate the downloaded apps that satisfy this requirement and the results are shown in Fig. 4. We can see that for almost all categories, there exist some apps that save the downloaded apk files on the external storage as well as subsequently install them. This suggests a serious attack surface if an attacker can replace them with phishing or malware apps successfully.

Upon a closer inspection, we find the vast majority of apps are flagged due to their app promotion functionality (part of advertisement). We analyze 64 apps across several categories that conduct app promotions and find that 42 of them will download the apk files directly from the Internet (as opposed to from Google Play which is safer but not

TABLE 7
Statistics of validation methods applied on different types of files

App Num (%) / File	Type	None	Size	Time	Message Digest	Check-sum	MAC[1]	Digital Signa-ture	Hash Code
Unknown Files		2,271 (93.23%)	51 (2.09%)	9 (0.37%)	88 (3.61%)	26 (1.07%)	0 (0%)	0 (0%)	0 (0%)
Image Files	.jpg, .bmp, .png, · · ·	1,696 (97.14%)	80 (4.58%)	15 (0.86%)	11 (0.63%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Media Files	.wav, .mp3, .mp4, · · ·	790 (99.50%)	4 (0.50%)	0 (0%)	4 (0.50%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Properties Files		56 (100%)	6 (10.71%)	1 (1.79%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Database Files	.DB, .db, .sqlite3,	122 (98.39%)	1 (0.81%)	0 (0%)	3 (2.42%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
TXT Files	.html, .js, .xml, · · ·	834 (91.85%)	12 (1.32%)	5 (0.55%)	57 (6.28%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
ZIP Files	.zip, .obb, .apk, · · ·	843 (92.13%)	11 (1.20%)	0 (0%)	71 (7.76%)	27 (2.95%)	0 (0%)	0 (0%)	0 (0%)
DOC Files	.doc, .xls, .ppt, · · ·	10 (90.91%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Other Files		1,576 (93.20%)	150 (8.87%)	0 (0%)	19 (1.12%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)

[1] It is short for Message Authentication Code.

TABLE 8
Input validation methods

Type	Check Content	App Num (%)
Prior Verification	File Size	283 (5.90%)
	File Time	28 (0.58%)
Post Verification	Message Digest	214 (4.46%)
	Data Checksum	27 (0.56%)
	Message Authentication Code	0 (0%)
	Digital Signature	0 (0%)
	Hash Code	0 (0%)

TABLE 9
Manual analysis of input validation results

App Num	TPR	TNR	FPR	FNR
320	83/93 (89.25%)	282/289 (97.57%)	7/289 (2.43%)	10/93 (10.75%)

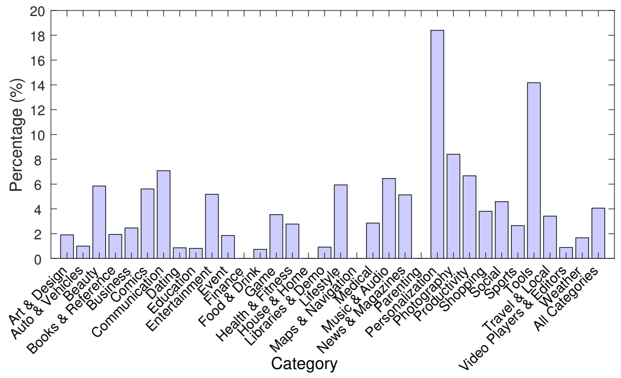


Fig. 4. Percentage of the apps that save apk files on external storage and call the system service to install them compared with the total number of apps in each category

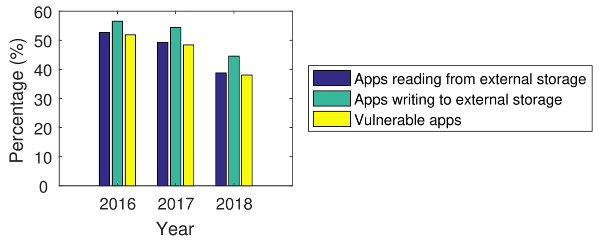


Fig. 5. Variation of vulnerable apps over recent years

available in some countries).

In addition, we conduct a focused study on third-party Android market apps. Due to the open nature of Android, in addition to Google Play, there is a wide range of third-party market apps that can help users find and install apps (e.g., Amazon Appstore and Samsung Galaxy Apps). In fact, in some countries such as China, Google Play is completely inaccessible and therefore users heavily rely on various local market apps built by Internet companies (e.g., Tencent and Baidu) and/or smartphone manufacturers (e.g., HUAWEI and Xiaomi). Through ExInspector, we find the following market apps that are vulnerable (as listed in Table 10) — installing apk files directly from shared storage. In fact, we are able to successfully exploit the following apps: GetJar, Mobogenie, 9Apps, Uptodown Android, Tencent Myapp, Baidu Mobile Guard and Baidu Mobile Assistant, as listed in Table 10, replacing the apk files downloaded by these apps right before installation.

Variations over the last three years. To learn how the situation has changed over recent years, we trace 1,818 apps

TABLE 10
Manual validation of the vulnerable third-party market apps discovered by ExInspector

	App Name	Save on external storage	Install the fake apk files
International	GetJar Apps	T	T
	Mobogenie	T	T
	9Apps	T	T
	Uptodown Android	T	T
	Percentage (%)	100%	100%
Chinese	Tencent MyApp	T	T
	Baidu Mobile Assistant	T	T
	Sougou Mobile Assistant	T	T
	Oppo App Store	T	F
	PP Assistant	T	F
	Mumayi Market	T	T
	Nduo	T	T
	d.cn	T	T
	ZTE App Market	T	F
	Wo Store	T	T
	Baidu Android Market	T	T
	Baidu Mobile Guard	T	T
	Percentage (%)	100%	75%

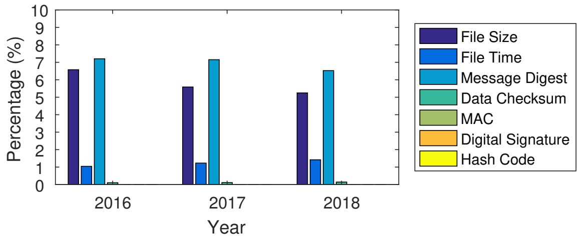


Fig. 6. Variation of validation methods over recent years

with invariant package names from 2016 to 2018. According to Fig. 5, we can see that the percentage of vulnerable apps declined somewhat over the past three years. Unfortunately, such decline is not caused by the increasing awareness of external storage security. Instead, it is due to the fact that the number of apps using external storage is decreasing, which has also been shown in Fig. 5. This claim can be further confirmed by Fig. 6, which shows little changes in the statistics of validation methods. In other words, there are still very few apps (below 7%) that adopt post validation methods to guarantee external files' integrity, which indicates that threats related to shared (external) storage is still being seriously overlooked although it has existed for a long time. Through the contact with a number of vulnerable apps' developers (e.g., Tencent, Qihoo 360 and Baidu), we find that they all think that installing a malware on the user's smartphone is difficult, which raises challenges to exploit the vulnerabilities related to external storage. However, the reality is the opposite, the unaware users are often tricked into installing malicious applications through different ways. Even users install the apps from the official market Google Play, Google Play Protect still fails to detect the new threats discovered within 90 days [23]. We think there can be some other potential reasons. Firstly, developers may misunderstand the "private" directory on external storage, provided by Android. An app's "private" directory on external storage is much different from the one on internal storage that can be accessed only by the app itself. Instead, on external storage, besides its real owner, the "private" directory still can be visited by each app granted with the permission to visit external storage. Meanwhile, in the early Android smartphones, volumes of external storage and internal storage are fixed, and the external storage is bigger than the internal one. Nowadays, with the storage volume of smartphone increasing rapidly, developers may still follow this old point so that they save a huge amount of files on external storage. However, in current Android smartphones, volumes of external storage and internal storage are not fixed, which are dynamically allocated based on current storage requirement of the smartphone. Besides the above analysis, we also compare the top 10 most popular file types and file consumers respectively, both of which are associated with files on external storage. The results are presented in Table 11 and Table 12, which show that their variations are both limited.

6 Attack Construction and Analysis

In this section, we attempt to construct realistic attacks based on the file tampering vulnerability. Firstly, we will discuss a building block of swift file replacement technique that comes in handy under stringent timing constraints. Then we will pick representative groups of apps as case studies with end-to-end attacks. These attacks are based on several kinds of files, i.e., multimedia files, apk files and html files. The attacks' applicability varies due to the different usages of files. Meanwhile, different from the latter two kinds, multimedia files may contain some biological features of humans, which requires attackers to pay more efforts to deal with. The details of these attacks will be presented soon. The demo for attacks described in this section can be found at https://sites.google.com/site/externalstorageattacks/. We have informed the apps' developers through different ways, such as their online security response centers (SRC), the built-in-app feedback mechanisms as well as the email addresses they leave on their websites. We are glad to see that some apps' developers (e.g., Tencent, Qihoo 360 and Baidu) have confirmed and remedied the vulnerabilities.

6.1 Swift File Replacement

As we begin analyzing the exploitability of the vulnerabilities, we realize that oftentimes files on external storage are consumed right after they are produced, leaving a small time window for file replacement. In this section, we describe a technique that can deterministically replace the file with a malicious one and have the subsequent read to operate on the malicious file.

The idea is straightforward. It hinges on the file system in Linux where a file can be renamed (moved) when it is being written. This is because once the file path resolution is finished, the corresponding inode remains unchanged and is what the file operations (read/write) rely on. This means that a malicious app can monitor the file write event and move the file away as soon as the write event is detected (without interrupting the write operation). Next, the attacker can simply create a new file taking up the original file path which is what the subsequent read operation will

TABLE 11
Variation of file types read from external storage over years

2016		2017		2018	
File Type	App Num (%)	File Type	App Num (%)	File Type	App Num (%)
unknown file[1]	602 (12.54%)	unknown file[1]	556 (11.59%)	unknown file[1]	444 (9.25%)
image file[1]	277 (5.77%)	image file[1]	264 (5.50%)	image file[1]	172 (3.58%)
.txt	129 (2.69%)	.zip	127 (2.65%)	.zip	102 (2.13%)
.zip	127 (2.65%)	.txt	114 (2.38%)	.txt	87 (1.81%)
.jpg	127 (2.65%)	.jpg	105 (2.19%)	.apk	82 (1.71%)
.apk	91 (1.90%)	.apk	101 (2.10%)	.jpg	79 (1.65%)
.xml	65 (1.35%)	.xml	61 (1.27%)	.obb	50 (1.04%)
.png	62 (1.29%)	video file[1]	59 (1.23%)	.png	45 (0.94%)
.obb	57 (1.19%)	.json	57 (1.19%)	.xml	42 (0.88%)
.json	55 (1.15%)	.png	55 (1.15%)	.dat	37 (0.77%)

[1] These files can have no suffixes but we can infer their type through their consumers.

TABLE 12
Variation of files consumers that read from external storage over years

2016		2017		2018	
Consumer Class	App Num (%)	Consumer Class	App Num (%)	Consumer Class	App Num (%)
android.graphics. BitmapFactory	383 (39.98%)	android.graphics. BitmapFactory	356 (39.78%)	android.graphics. BitmapFactory	245 (34.75%)
java.io.ByteArray-OutputStream[1]	93 (9.71%)	android.content.pm. PackageInstaller	100 (11.17%)	java.io.ByteArray-OutputStream[1]	94 (13.33%)
android.content.pm. PackageInstaller	88 (9.19%)	java.io.ByteArray-OutputStream[1]	97 (10.84%)	android.content.pm. PackageInstaller	80 (11.35%)
android.widget. VideoView	75 (7.83%)	android.widget. VideoView	79 (8.83%)	android.text. TextUtils	56 (7.94%)
java.security. MessageDigest	69 (7.20%)	java.security. MessageDigest	64 (7.15%)	java.security. MessageDigest	46 (6.52%)
java.io.StringWriter[1]	67 (6.99%)	java.io.StringWriter[1]	61 (6.82%)	java.util.Properties	44 (6.24%)
java.util.Properties	66 (6.89%)	java.util.Properties	57 (6.37%)	android.media. MediaPlayer	43 (6.10%)
android.media. MediaPlayer	57 (5.95%)	android.media. MediaPlayer	55 (6.15%)	java.io.StringWriter[1]	39 (5.53%)
android.text. TextUtils	48 (5.01%)	android.text. TextUtils	54 (6.03%)	android.widget. VideoView	36 (5.11%)
*.SQLiteDatabase	42 (4.38%)	*.SQLiteDatabase	38 (4.25%)	*.SQLiteDatabase	31 (4.40%)

[1] These classes may propagate the untrusted file content further.

rely on. In addition, an attacker still has access to the original file (that was earlier renamed) and be able to read the full content. According to experiments, even when reading only a small amount of data, the file move operation is still fast enough to succeed.

6.2 Realtime Voice Message Replacement Attack

As we mentioned earlier, most voice messages are stored on external storage and they can be manipulated by any apps granted the permission to write files on external storage. In this paper though, we are interested in understanding attacks that can be achieved through tampering the stored voice messages. To this end, based on the file replacement capability, we construct a powerful realtime "man-in-the-middle" attack to both listen and write the voice messages as they are being sent and received. For the remaining discussion, we take the WeChat app — a free text/voice messaging & calling app with over half a billion users — as a concrete case study.

Vulnerable Workflow. WeChat's workflow (similar to the one in WhatsApp and likely others) to send and receive voice messages is illustrated in Fig. 7 and Fig. 8. When sending a voice message, instead of sending the message directly over the network, there's an encoder thread to encode and persist the voice message to a file on external storage. A sender thread then immediately reads the file after it is completed (signaled by the encoder thread along with a specific file path), and send it over the network. The design is a fairly reasonable one from the perspective of software engineering, as it guarantees that the voice message can be retransmitted even if the network fails or crashes occur before the message is delivered. However, there is a time window between the encoder thread persisting the file and the sender thread picking up the file, in which a malicious app can replace the file to be sent.

In the case of receiving a message, a receiver thread persists it to a file on the external storage and signals the main thread to display a notification. Here again there is

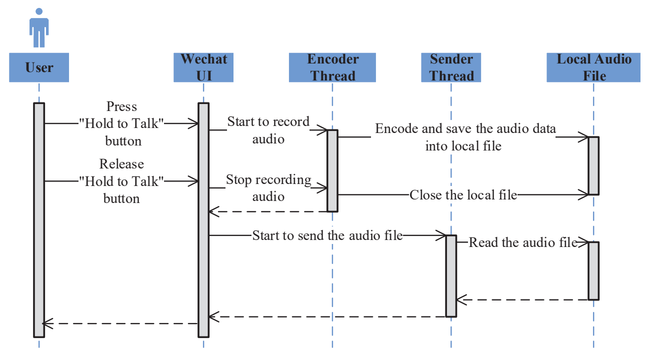


Fig. 7. WeChat's workflow to send voice messages

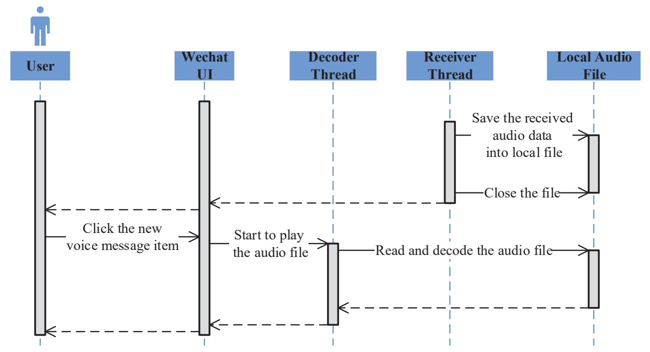

User
Wechat UI
Decoder Thread
Receiver Thread
Local Audio File
Save the received audio data into local file
Close the file
Click the new voice message item
Start to play the audio file
Read and decode the audio file

Fig. 8. WeChat's workflow to receive voice messages

a time window between the two operations in which a malicious app can replace the received file. In addition to being able to replace the outgoing and incoming voice messages, the file replacement technique also keeps a copy of the original message for read, and can also optionally drop the messages (by simply removing the file). This gives the malicious app an almost complete "man-in-the-middle" (read/write/drop) capability.

Attack Overview. Now we demonstrate an end-to-end attack illustrating how to leverage the swift file replacement to impersonate users on both ends during a voice chat session. We assume a malicious app running in the background on

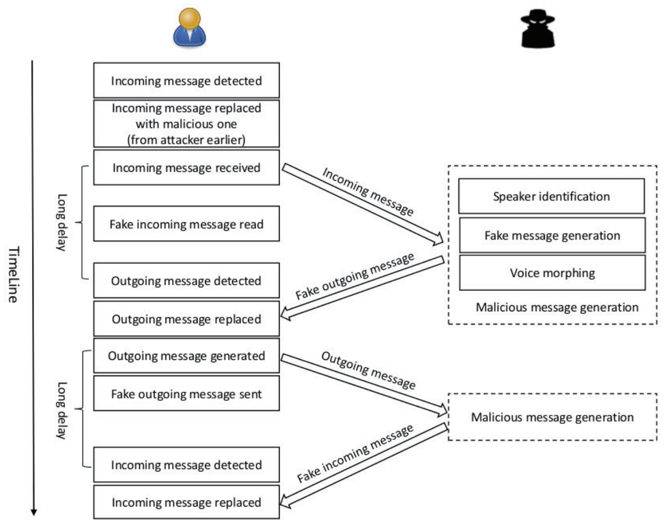

Timeline
Long delay
Long delay
Incoming message detected
Incoming message replaced with malicious one (from attacker earlier)
Incoming message received
Fake incoming message read
Outgoing message detected
Outgoing message replaced
Outgoing message generated
Fake outgoing message sent
Incoming message detected
Incoming message replaced
Incoming message
Fake outgoing message
Outgoing message
Fake incoming message
Speaker identification
Fake message generation
Voice morphing
Malicious message generation
Malicious message generation

Fig. 9. The voice message replacement attack sequence against WeChat

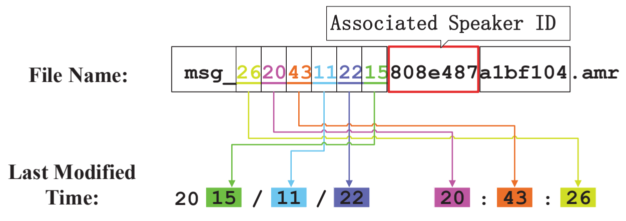


Fig. 10. Wechat naming rule of audio file

the victim's device (i.e., Alice's device). When Alice chats with Bob through voice messages, the malware aims to impersonate Alice to chat with Bob and impersonate Bob to chat with Alice at the same time. The attack process is shown in Fig. 9. The malware continuously monitors the target directory where the audio files are stored. Once discovering a new audio file is created, it utilizes some side channels (e.g., network statistics) to identify whether the audio file is an incoming or outgoing message. Next, it launches the swift file replacement technique against the original file. When the original file is completed written, the malware sends it to the server. The server extracts the biometric features from the audio file to identify the speaker, so that later fake voices can be created to mimic the speaker. Further, the attacker controlling the server can decide what fake voice messages to be put in place of the original one. If the audio file comes from Alice, the fake response is converted into the voice of Bob, and vice versa. The converted fake response will be immediately sent back to the user's device. The malware stores this fake response and use it to substitute the real response when it comes. Note we can replace the voice message only if Bob sends a real response back (which triggers a file to be written on external storage).

Attack Details. To implement a realistic end-to-end attack, there are three requirements which we address in order:

• *User identification.* Once we obtain a voice message, it is a key step to identify which user and the corresponding chat session a message belongs to, as the subsequent attack steps such as preparing the following responses is heavily dependent on it. For WeChat, fortunately we realize that the naming scheme for the audio files already leaks information about which user (or chat session) the message belongs to. After some reverse engineering, it is clear that file names are constructed based on the sender/speaker user ID (i.e., a hash of the wechat user ID) as well as timestamps. Fig. 10 illustrates the detailed naming scheme. Excluding the constant string "msg_", the first 10 characters indicate the file's last modified time (date, and hour:minute:second) and the last 7 characters are derived from the file's last modified time in milliseconds. The middle characters from 13 to 19 are generated by the hash of wechat user ID of the speaker. Note that if it is a message of the victim user (i.e., Alice), the user id will simply be the id of the victim. This means that we can also distinguish incoming vs. outgoing messages based on the speaker's user ID.

• *Voice morphing.* To trick the victim user into believing a fake voice is from a real friend, we need to apply voice morphing technique [24] to convert the attacker's voice into a specific speaker's voice. To do so, we need the training data for voices from the target speaker. Fortunately, WeChat (and other apps with voice messaging capabilities) stores all historic voice messages in external storage which can naturally serve as training data (as they are readable by any app). Through the user identification earlier, we can conveniently group these audio files by speakers. Without going into the details, we essentially build a fully functioning voice morphing system based on prior work [25], [26] and toolset [27].

• *Timeliness.* The above two steps can be time-consuming. In addition, the attacker committing a fraud needs time to prepare the corresponding responses (e.g., talking the victim into lending money). This asks for a very time-sensitive attack strategy. We have described the process of WeChat's handling of incoming and outgoing messages in Fig. 7 and Fig. 8. We know that the attack window to replace incoming or outgoing voice messages is small. Our attack strategy is to leverage the long delays due to human interactions to prepare the fake voice messages in time. We illustrate the process in a timeline in Fig. 9. As we can see, there are two long delays in a typical message receive and send cycle. The first one is from Alice receiving an incoming message from Bob, to when the message is actually read (by Alice clicking on the screen to play the voice). During this delay, the attacker sees the incoming message and prepares a proper answer message (fake outgoing message from Alice to Bob). The second long delay is from Alice generating an outgoing message to Bob, to when she receives an incoming message from Bob. During this delay, the attacker sees the outgoing message and prepares a proper answer message (fake incoming message from Bob to Alice). As we can see, the attacker always prepares the response messages one step ahead. As the send/receive cycle continues, the attacker will be able to replace any outgoing and incoming message.

If the long delay sometimes is not enough to cover the cost of generating a fake voice message and morphing it into the voice of the target speaker, we consider a few backup strategies to buy time. One simply strategy is to use some universally common or meaningless terms to replace the original messages (e.g., "hello" or "hold on a second"). In this way, the attacker can send the real response later to replace the following voice message. As a further optimization, the attacker can utilize *android.os.FileObserver* to monitor the events (e.g., OPEN, ACCESS, CLOSE_WRITE, CLOSE_NOWRITE) on the specific voice file. If the real response has been generated successfully later and the meaningless message has not read by the user, the attacker still has the chance to replace the meaningless one with the meaningful one. Our attack video demo illustrates the case where we pretend to be Alice and try to trick Bob somehow without letting Alice notice.

6.3 Apk Replacement Attack

To launch the apk replacement attacks successfully, it is important to know how the involved processes work. As Fig. 11 shows, besides the attack process, there are two processes involved, one is responsible for downloading and saving the apk files and the other is the system service responsible for installing the app. There is therefore a time window from when the apk file being written to when the

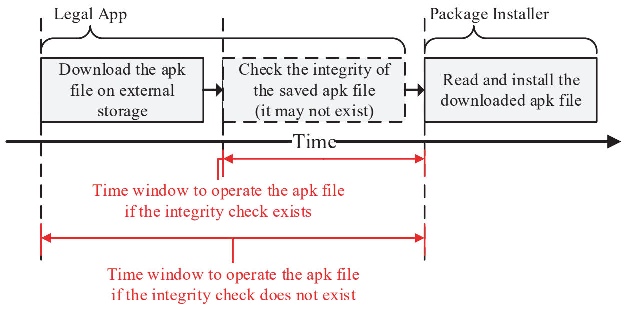

Legal App
Package Installer
Download the apk file on external storage
Check the integrity of the saved apk file (it may not exist)
Read and install the downloaded apk file
Time
Time window to operate the apk file if the integrity check exists
Time window to operate the apk file if the integrity check does not exist

Fig. 11. Time window to replace the apk file

file being read by system service for installation. Even if there exists validation during this time window, it is inherently difficult to guarantee security because of the nature of Time-Of-Check, Time-Of-Use (TOCTOU) problem. Let us say the first process validates its integrity before passing the file to system service; right before system service picks up the file, the attack process can replace the file, bypassing any prior validation and the system is completely in the dark [6].

Here we take the Tencent Myapp (a third-party market app) as an example. Tencent Myapp is the top 1 app store in China, the coverage of which is 25.40% by June 2017 according to Newzoo's report [28]. Myapp downloads the apk file on the external storage */sdcard/Tencent/tassistant/apk/*. Once we discover a new downloaded apk file is being checked by Myapp, we can use the fake apk file to replace the original one. We name the fake apk file with a different package name but it can be much similar to the original one's. In this way, Android will be happy to install the fake apk file just as a new app is installed.

6.4 Web Resource Replacement

Nowadays, many developers piggyback on the web infrastructure to save resources for app development. It is therefore natural to support web object caching (e.g., HTML, video, images, Javascript). To provide a fluent user experience, many of these apps even prefetch objects according to some policies. If these web resources are stored on external storage, they are subject to tampering. Unfortunately, as we show in § 5, there do exist many apps that use sensitive files such as .html or .js files on external storage (and most of them do not have any integrity checks). Most of the time, apps simply directly pass such files to the corresponding APIs such as `WebView` for consumption. As a result, it is straightforward for the adversary to replace these files with fraudulent ones. In fact, if these resources are prefetched and stored, the time window for launching attacks can be huge (after prefetching before consumption). The attack will be able to read what data is contained in the original file and craft a phishing version of it easily (e.g., by searching for login boxes and replace the login request URL with a malicious one).

No Verification. We pick some apps that are reported to have no integrity checks. For instance, MakeInIndia is an information sharing app, whose most activities involve displaying html files. To our surprise, it caches almost all important html files on the external storage */sdcard/.MakeInIndia/www/*, including its home page. After manual analysis, we are able to confirm that it indeed has no verification steps whatsoever after reading the offline html files. We tested a trivial attack by replacing the original homepage with a fraudulent one, and the app would happily display the updated page. This clearly allows us to conduct a phishing attack that can steal user's login credentials (or even other sensitive information such as credit card information depending on whether it is part of the expected functionalities). Interestingly, given that every html page is cached, we can theoretically completely rewrite the functionality by controlling the starting page and creating a number of sub-pages.

Complex Verification. Although some apps will perform some complex verification before loading html files, it is still bypassable. For instance, we studied an app called Tencent Video, which is a popular video distribution platform (1.9 billion downloads in China [29]), that also stores html files on external storage *Android/data/com.tencent.qqlive/files/.webapp/dirs/*. Unlike MakeInIndia, it does check the integrity of the file before reading. However, the validation and the file read are two separate operations, making it a Time-Of-Check, Time-Of-Use (TOCTOU) vulnerability, which allows us to construct an exploit and replace any files downloaded by the app. Since Tencent Video is a part of the big network of Tencent, it has many features such as account management and even financial information management (to pay for premium content). Worse, as opposed to browsers, smartphone apps generally do not show security indicators for TLS and SSL through `Webview` (as confirmed in the Tencent Video app). An attacker can therefore trick the users into trusting a locally constructed payment page (linked from the tampered page).

7 Defense

As we have seen in this paper, file tampering on external storage can lead to a series of attacks against vulnerable apps, posing serious threats to all Android users. Though, prior work (e.g., [30]) has proposed to apply Linux file permissions on external storage, the backward compatibility issue is really a big problem, i.e., the external storage cannot be formatted to a Linux permission-enabled file system as the need to be compatible with FAT-formatted SD cards is real, which keeps any Linux-permission-related enforcement out of the question. Therefore, defense at the app level becomes the sole solution.

As Google suggests, developers should always perform input validation when they read files from external storage. Unfortunately, not all validation methods are equal and there are not much guidance on how this should be done in practice. Below we list a number of caveats of input validation and best practices. As we show, there are a number of validation strategies are easily bypassable (e.g., file size, modification time). Worse, some are just plain wrong. For instance, if the verification information is purely extracted from the very file stored on external storage (subject to tampering), it can be easily replaced. Therefore, the best practice should use verification information out-of-band (e.g., stored on internal storage) instead of being embedded in the file on external storage itself. For example, if the files are downloaded from the network, a cryptographic signature can be attached by the server to aid the verification of the integrity of the downloaded file (the signature can be kept in memory or internal storage). In addition, it is critical that the verification should not be done prior to reading the file. Otherwise, there is a potential Time-Of-Check, Time-Of-Use (TOCTOU) vulnerability where an attacker replace the file after the verification process is passed.

8 Related Work

In this paper, we focus our attentions on the files on external storage. In fact, many researches [2], [3], [31], [32] have presented the risks caused by the files on external storage. The malicious advertisement [33] and the residue of the deleted apps [34] all can endanger the users' privacy. Meanwhile, tampering the files on external storage also can cause serious problems, such as code injection attacks on HTML5-based mobile apps [35], the Android installer hijacking [5], the vulnerabilities caused by the world writable code [4] and the vulnerabilities caused by code dynamically loading [6]. Different from these work, instead of just focusing on some special kinds of files on external storage, we conduct a more comprehensive survey about the untrusted files on external storage, find and validate some widespread attacks on external storage.

Beside the attacks, some security strategies [30], [32] are also proposed to protect the shared files on external storage. However, shared files on external storage are not only allowed to read, but also to write. We have proofed the risks caused by tampering the files on external storage in this paper.

Actually, besides the files on external storage, apps can utilize some other methods to share data, such as Unix domain sockets (though they are not secure [7]). The privacy leakages also can be caused by the data sharing with the web code [36] and the external libraries [37], [38], [39]. Meanwhile, some apps are vulnerable to the component hijacking attacks [40].

In this paper, we utilize ExInspector to find out the vulnerable apps, which is based on the static code analysis. In fact, static code analysis has already been used to analyze Android apps in prior work [41], [42]. Some work such as ScanDroid [43], Leakminer [44], AndroidLeaks [45] and DroidSafe [46] utilize static code analysis to discover the sensitive information leakages in the apps. Meanwhile, through the static code analysis, researchers also find out some interesting information, such as the component hijacking vulnerabilities [47], [48], and inter-component privacy leaks [49].

9 Conclusion

In this paper, we systematically study a simple but overlooked threat related to Android external storage — the lack of input validation (e.g., integrity verifications) when consuming files on external storage. By undertaking an empirical study through a static analysis tool we develop, we find over 30% of the 13,746 analyzed popular apps on the market suffer from such problems. By investigating the types of files consumed, we find shockingly a large fraction of apps store and consume sensitive files such as .apk, .html, .js, and voice message files. These findings allow us to construct a variety of attacks. The ubiquity of this class of vulnerabilities calls for revamps of the basic feature of external storage. We thereby define better access control policies for external storage to eliminate this class of vulnerabilities for most apps. In the future work, we will continue to find more vulnerabilities about Android external storage.

References

[1] "Android storage options," https://developer.android.com/guide/topics/data/data-storage.html, Accessed in May 2017.

[2] S. Gisdakis, T. Giannetsos, and P. Papadimitratos, "Android privacy c (r) ache: Reading your external storageand sensors for fun and profit," 2015.

[3] X. Liu, Z. Zhou, W. Diao, Z. Li, and K. Zhang, "An empirical study on android for saving non-shared data on public storage," in *IFIP International Information Security Conference*. Springer, 2015, pp. 542–556.

[4] "World Writable Code Is Bad, MMMMKAY," https://www.nowsecure.com/blog/2015/08/10/world-writable-code-is-bad-mmmmkay-4/, Accessed in August 2017.

[5] "Android Installer Hijacking Vulnerability Could Expose Android Users to Malware," https://researchcenter.paloaltonetworks.com/2015/03/android-installer-hijacking-vulnerability-could-expose-android-users-to-malware/, Accessed in August 2017.

[6] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications." in *NDSS*, vol. 14, 2014, pp. 23–26.

[7] 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 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 80–91.

[8] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11, 2011.

[9] "Best Practices for Security & Privacy – Security Tips," https://developer.android.com/training/articles/security-tips.html, Accessed in May 2017.

[10] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in)security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 50–61. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382205

[11] H. Vijayakumar, J. Schiffman, and T. Jaeger, "Sting: Finding name resolution vulnerabilities in programs," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 585–599.

[12] "Apkpure.com," https://apkpure.com/, Accessed in May 2017.

[13] Y. J. Jia, Q. A. Chen, Y. Lin, C. Kong, and Z. M. Mao, "Open doors for bob and mallory: Open port usage in android apps and security implications," in *IEEE European Symposium on Security and Privacy*, 2017.

[14] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11, 2011.

[15] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, 2015.

[16] "Android Source Code Comments," https://android.googlesource.com/platform/system/core/+/android-7.1.2_r6/sdcard/sdcard.c#61, Accessed in May 2017.

[17] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[18] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11, 2011.

[19] "Silk," https://en.wikipedia.org/wiki/SILK, Accessed in May 2017.

[20] "Adaptive multi-rate audio codec," https://en.wikipedia.org/wiki/Adaptive_Multi-Rate_audio_codec, Accessed in May 2017.

[21] "Advanced audio coding," https://en.wikipedia.org/wiki/Advanced_Audio_Coding, Accessed in May 2017.

[22] "Opus interactive audio codec," http://opus-codec.org/, Accessed in May 2017.

[23] "McAfee Mobile Threat Report Q1, 2018," https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2018.pdf, Accessed in November 2018.

[24] Z. Wu and H. Li, "Voice conversion versus speaker verification: an overview," *APSIPA Transactions on Signal and Information Processing*, vol. 3, p. e17, 2014.

[25] D. Erro, I. Sainz, E. Navas, and I. Hernaez, "Harmonics plus noise model based vocoder for statistical parametric speech synthesis," *IEEE Journal of Selected Topics in Signal Processing*, vol. 8, no. 2, pp. 184–194, 2014.

[26] D. Erro, E. Navas, and I. Hernaez, "Parametric voice conversion based on bilinear frequency warping plus amplitude scaling," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 21, no. 3, pp. 556–566, 2013.

[27] "Ahocoder," https://aholab.ehu.es/users/derro/software.html, Accessed in May 2017.

[28] "Top 10 Android App Stores In China," https://newzoo.com/insights/rankings/top-10-android-app-stores-china/, Accessed in August 2017.

[29] "Tencent Video," http://sj.qq.com/myapp/detail.htm?apkName=com.tencent.qqlive, Accessed in May 2017.

[30] Q. Do, B. Martini, and K.-K. R. Choo, "Enforcing file system permissions on android external storage: Android file system permissions (afp) prototype and owncloud," in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2014, pp. 949–954.

[31] X. Liu, W. Diao, Z. Zhou, Z. Li, and K. Zhang, "Gateless treasure: How to get sensitive information from unprotected external storage on android phones," *CoRR, vol. abs/1407.5410*, 2014.

[32] Y. Xu and E. Witchel, "Maxoid: Transparently confining mobile applications with custom views of state," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 26.

[33] S. Son, D. Kim, and V. Shmatikov, "What mobile ads know about mobile users," in *Proc. 23rd Annual Network and Distributed System Security Symposium (NDSS16)*, 2016.

[34] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du, "Life after app uninstallation: Are the data still alive? data residue attacks on android," in *Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, California, USA*, 2016.

[35] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "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*. ACM, 2014, pp. 66–77.

[36] G. S. Tuncay, S. Demetriou, and C. A. Gunter, "Draco: A system for uniform and fine-grained access control for web code on android," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 104–115.

[37] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. A. Gunter, "Free for all! assessing user data exposure to advertising libraries on android," *NDSS 2016*, 2016.

[38] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou, "Following devils footprints: Cross-platform analysis of potentially harmful libraries on android and ios," 2016.

[39] D. Titze and J. Schütte, "Preventing library spoofing on android," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 1. IEEE, 2015, pp. 1136–1141.

[40] M. Zhang and H. Yin, "Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications." in *NDSS*, 2014.

[41] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[42] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applica-

tions," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 89–99.

[43] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android," 2009.

[44] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *Software Engineering (WCSE), 2012 Third World Congress on*. IEEE, 2012, pp. 101–104.

[45] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *International Conference on Trust and Trustworthy Computing*. Springer, 2012, pp. 291–307.

[46] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*. Citeseer, 2015.

[47] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 229–240.

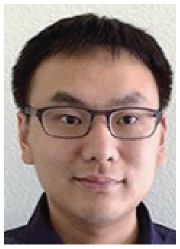
[48] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones." in *NDSS*, vol. 14, 2012, p. 19.

[49] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 280–291.

Shaoyong Du received the B.E. degree in software engineering from Zhengzhou University, China, in 2012. He is currently working toward the Ph.D. degree in the Department of Computer Science and Technology at Nanjing University, China. His current research focuses on security and privacy in mobile computing.

Pengxiong Zhu received the B.S. degree in computer science from Nanjing University, China, in 2017. He is currently working toward the Ph.D. degree in the Department of Computer Science at University of California, Riverside. His current research focuses on network security.

Jingyu Hua received the B.E. and M.E. degrees both in software engineering from Dalian University of Technology, China, in 2007 and 2009, respectively. He received the Ph.D. degree in Informatics from Kyushu University, Japan, in 2012. His current research interests include security and privacy in mobile computing, and system security.



Zhiyun Qian received the Ph.D. degree in Computer Science and Engineering from University of Michigan in 2012. His research interest is on system and network security, including Android security, Internet security (e.g., TCP/IP), side-channel attacks and defenses, and infrastructure security (e.g., cellular networks).

Zhao Zhang received the B.E. degree in software engineering from Northeastern University, China, in 2012. He is currently working toward the Master degree in the Department of Computer Science and Technology at Nanjing University, China. His current research focuses on security and privacy in mobile computing.

Xiaoyu Chen received the B.E. degree in department of mathematics from NanJing University, China, in 2016. He is currently working toward the master degree in the Department of Computer Science and Technology at Nanjing University, China. His current research focuses on security and privacy in mobile computing.

Sheng Zhong received the B.S. and M.S. degrees from Nanjing University, in 1996 and 1999, respectively, and the Ph.D. degree from Yale University, in 2004, all in computer science. He is interested in security, privacy, and economic incentives.