2019 IEEE Symposium on Security and Privacy

Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps

Chaoshun Zuo
The Ohio State University

Zhiqiang Lin
The Ohio State University

Yinqian Zhang
The Ohio State University

***Abstract*—Increasingly, more and more mobile applications (apps for short) are using the cloud as the back-end, in particular the cloud APIs, for data storage, data analytics, message notification, and monitoring. Unfortunately, we have recently witnessed massive data leaks from the cloud, ranging from personally identifiable information to corporate secrets. In this paper, we seek to understand why such significant leaks occur and design tools to automatically identify them. To our surprise, our study reveals that lack of authentication, misuse of various keys (*e.g.*, normal user keys and superuser keys) in authentication, or misconfiguration of user permissions in authorization are the root causes. Then, we design a set of automated program analysis techniques including obfuscation-resilient cloud API identification and string value analysis, and implement them in a tool called LeakScope to identify the potential data leakage vulnerabilities from mobile apps based on how the cloud APIs are used. Our evaluation with over 1.6 million mobile apps from the Google Play Store has uncovered 15,098 app servers managed by mainstream cloud providers such as Amazon, Google, and Microsoft that are subject to data leakage attacks. We have made responsible disclosure to each of the cloud service providers, and they have all confirmed the vulnerabilities we have identified and are actively working with the mobile app developers to patch their vulnerable services.**

I. Introduction

The cloud has significantly changed the landscape of modern computing. It has been proven to be the go-to platform for data storage, data processing, data analytics, and data backup and recovery, due to its immense benefits such as being highly available, massively scalable, hugely cost-saving, and quickly deployable. The cloud is everywhere, and "the cloud presence is becoming a norm in almost 70% enterprises across the globe, and they have at least one application running on the cloud" [34].

With the rapid growth of cloud computing, there has also been a tremendous growth of mobile apps. From an end-user perspective, mobile apps can be considered the front-end of an Internet service with the cloud as the back-end. By using the cloud, service providers (*e.g.*, news, weather, and shopping) do not have to worry about the scalability and availability of their back-end servers, and instead they can just focus on their core business logic and develop their mobile apps. Almost all of our daily used Internet services have their own dedicated mobile apps. As of today, there are more than five million mobile apps in only the Google Play Store and Apple App Store [20].

Unfortunately, with the use of the cloud as a mobile app back-end, we have also witnessed massive data leaks from the cloud recently, ranging from personal medical records to corporate secrets. For example, it was reported that insecure back-end databases of mobile apps were exposing an estimated 280 million sensitive user records including personally identifiable information (PII) such as user-names, passwords, emails, phone numbers, and locations [36]. Such leakage also applies to high profile companies such as Verizon, which may have leaked 100MB of its corporate secrets from a publicly accessible Amazon S3 bucket [33].

A conventional wisdom is to get better security by using the cloud, but this has actually led to massive data leaks. Consequently, it has resulted in many questions to be answered. For instance, what are the root causes of data leaks in the cloud? Are they caused by cloud providers, app developers, or both? Can we systematically identify data leakage vulnerabilities in cloud services? How can they (*i.e.*, both cloud providers and app developers) prevent them from happening again?

In this paper, we seek to perform a systematic study and answer these questions. Since we do not have lower-level access to how each cloud provider manages the customers' services, we can only analyze the front-end of the cloud services. One such front-end is mobile apps. Increasingly, we notice that more and more mobile apps are using cloud APIs for various services such as authentication, authorization, and storage, without directly setting up and managing those back-end infrastructures. Therefore, by inspecting how the cloud APIs are used, we can understand how each mobile app manages its customer data and thus identify the data leakage vulnerabilities.

In particular, we first look into the typical APIs offered by cloud providers for mobile app development, and examine how app developers would develop their mobile apps and manage their security with these APIs. From this study, we uncover that a mobile app must perform an extra service authentication when communicating with the cloud back-end, such that the cloud provider knows which app issues the request and which resource this app aims to access. Improper management of user authentication and misconfiguration of user permissions in authorization are the root causes that lead to the various data leaks from the cloud. In our study, we find many cloud services suffer from such vulnerabilities.

Having discovered the root causes of the data leaks in the cloud, we also notice that it is possible to develop a principled approach to automatically identify these data leakage vulnerabilities by inspecting how mobile apps use the authentication keys and how servers handle users' requests. One challenge lies in how to make sure there is no leakage of customer data when performing our analysis, since we are a third party and we certainly must not access any of the customer data. Also, there are millions of mobile apps today, so we must design a scalable, automated, and efficient approach. We have addressed these challenges and built a tool called LeakScope with a set of program analysis techniques including obfuscation-resilient cloud API identification, string value analysis, and zero-data-

DOI 10.1109/SP.2019.00009

Cloud Service	Provider	Database Management	User Management	Storage	Notification Delivery
AWS	Amazon	`DynamoDB`	`Cognito`	`Amazon S3`	`Amazon SNS`
Azure	Microsoft	`Easy Tables`	`Azure Active Directory`	`Azure Storage`	`Notification Hubs`
Firebase	Google	`NoSql Database`	`Firebase Authentication`	`Google Storage`	`Notifications`

Table I: Key Components Offered by Popular Cloud Providers for Mobile App Development.

leakage vulnerability identification to automatically locate cloud data leakage vulnerabilities in mobile apps.

We have evaluated LeakScope with 1,609,983 mobile apps. Surprisingly, our tool has uncovered 15,098 unique mobile apps (10 of them have between 100 million and 500 million users), whose back-end servers—hosted in clouds such as Amazon AWS, Google Firebase, and Microsoft Azure—are subject to data leakage attacks. We have made responsible disclosure to all of the cloud providers, and they all have confirmed the vulnerabilities we identified. We also rely on them to further notify the app developers by sending them the list of the vulnerable apps and some other detailed information. Cloud providers have seriously considered our discovery and notified the vulnerable app developers. The cloud providers are also actively working on patching their services, some of which can be noticed publicly (e.g., updating the official documentation to correct a misguided example in the SDK [8]). In addition, a number of popular mobile apps have also been patched after cloud providers notified them.

In short, we make the following contributions.

- **Systematic study**. We make a first step towards systematic understanding of data leaks in the cloud, and our study has identified the root causes of recent massive data leaks.
- **Automated techniques**. We develop a set of program analysis techniques including new obfuscation-resilient cloud API identification, string value analysis, and zero-data-leakage vulnerability identification to automatically locate cloud data leakage vulnerabilities from mobile apps.
- **Empirical evaluations**. We have evaluated our techniques with 1,609,983 mobile apps, and have identified 15,098 unique mobile apps, whose services running in the cloud are subject to data leakage attacks. We have made responsible disclosures, and some of the services have been patched.

II. Background

In this section, we present the background related to why there are cloud APIs for mobile app development (§II-A) and how to use them (§II-B).

A. Why Using Cloud APIs for Mobile App Development

Before cloud APIs became popular, when developing a mobile app (running atop Android or iOS), developers had to build the entire infrastructure from scratch, including both the front-end app and the back-end servers (although they could rent some infrastructure, *e.g.*, virtual machines or database servers, from a cloud provider). Moreover, even after releasing the app, the entire system would require non-trivial efforts to ensure its stability and security. Furthermore, when an app become popular, they would have to maintain high availability and scale the system to accommodate a potentially massive increase in users.

Having recognized the needs of mobile app developers, mainstream cloud providers have provided mobile back-end as a service (mBaaS), including the corresponding platforms and APIs for rapid mobile app development. For instance, with a few configuration steps, developers can quickly build a fully functional mobile app back-end. The advantage of using mBaaS clouds is that a developer does not have to worry about how to set up and manage back-end servers and handle massive scale requests. Instead, they can just focus on developing their front-end mobile apps and rely on cloud providers to provide the infrastructure and manage server security, reliability, and scalability.

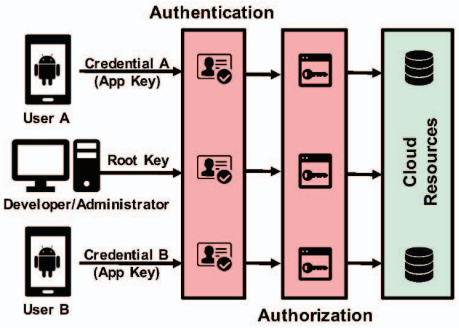

Figure 1: A Typical Architecture of Mobile App Development with Cloud APIs.

As of today, there are many mBaaS cloud providers including Amazon, Backendless, Google, Kinvey, Microsoft, Oracle, *etc.* [11]. Due to our limited man power, in this paper, we only focus on the mBaaS clouds provided by Amazon, Google, and Microsoft, though our approach should be applicable to other clouds as well. As shown in Table I, each cloud provider offers four typical components, including Database Management, User Management, Storage, and Notification Delivery with corresponding APIs such as `setValue` and `removeValue` [13], `createUserWithEmailAndPassword` [1], `sendPasswordResetEmail` [10], `putFile` [17], `listFilesAndDirectories` [2], `CreatePlatformEndpoint` [4], and `publish` [18].

B. How to Use the Cloud APIs

One important difference compared to developing mobile apps without cloud APIs is that the cloud needs to know who issues the API calls (for various purposes such as isolation, accountability, and security). Therefore, cloud APIs often take an app key as one of the parameters. There is also another key issued by cloud providers, which is the root key. These two keys are completely different:

- **App Key**. An app key has very limited permissions. With this key, the cloud knows which app the API call comes from, and only the publicly available resources that belong to that app can be accessed. Since multiple

Service	Key Type	Example
Azure Storage	Account Key	DefaultEndpointsProtocol=https; AccountName=*;AccountKey=*
	SAS	https://*.blob.core.windows.net/* ?sv=*&st=*&se=*&sr=b& sp=rw&sip=*&spr=https&sig=*
Notification Hub	Listening Key	Endpoint=sb://*.servicebus.windows.net/; SharedAccessKeyName= DefaultListenSharedAccessSignature; SharedAccessKey=*
	Full Access Key	Endpoint=sb://*.servicebus.windows.net/; SharedAccessKeyName= DefaultFullSharedAccessSignature; SharedAccessKey=*

Table II: Examples of the Keys Used in Mobile App Development with Microsoft Azure Cloud. We use symbol * to anonymize those sensitive data in the keys.

apps may use the same cloud database and storage in the back-end, the cloud providers virtualize and isolate these resources by using app keys.

- **Root Key**. The root key has very powerful permissions. With this key, all back-end resources belonging to it can be accessed. Typically developers should keep this key secret, as only system administrators should use it.

A typical usage of the app key and root key issued by cloud providers and the architecture of a cloud based mobile app is illustrated in Figure 1. At a high level, developers need to first register with the cloud back-end to acquire the app key and root key, set up the back-end database (e.g., tables) and storage, and implement proper authentication and authorization in the back-end. Then, the mobile app can invoke the corresponding cloud APIs by passing the corresponding app key and other parameters to access the resources it needs from the cloud.

III. Our Discovery

Having explained the details of how to use mBaaS clouds for mobile app development, we next uncover the common mistakes made by both app developers and cloud providers and consequently the vulnerabilities introduced. Since cloud providers have made mobile app development much easier by using their APIs, the playground for app developers becomes much smaller. A developer's security responsibilities reduce to properly performing authentication and authorization for mobile app users. Any mistakes with these operations will lead to account compromises such as data leaks and data tampering. In particular, we have identified two major types of data leakage vulnerabilities: misuse of various keys in authentication (§III-A) and misconfiguration of user permissions in authorization (§III-B).

A. Misuse of Various Keys in Authentication

When using cloud APIs, it is necessary to pass the corresponding keys so the cloud providers can connect the apps with the corresponding virtualized back-end servers. The fundamental reason for key misuse is that developers forget the differences between an app key and a root key, and that the cloud providers do not enforce the proper use of the keys. More specifically, when using cloud APIs, if a cloud interface accepts both the app key and the root key, this will cause confusion to developers, thereby resulting in vulnerabilities. We found that both Amazon's and Microsoft's clouds suffer from this problem, and we have identified the key misuses in

```
1 package com.appname
2 public class ImagesHelper {
3   private final String storageAccountKey;
4   private final String storageAccountName;
5
6   private ImagesHelper(Context arg3) {
7     int v0 = 2131099713;
8     int v1 = 2131099712;
9     this.storageAccountName =
10       Utils.getStringFromResources(this.imgContext, v0);
11    this.storageAccountKey =
12       Utils.getStringFromResources(this.imgContext, v1);
13  }
14
15  public void downloadImages(com.appname.Listeners.Callback arg5,
16       com.appname.Listeners.OnDownloadImagesUpdateListener arg6) {
17    StringBuilder v0 = new StringBuilder();
18    v0.append("DefaultEndpointsProtocol=http;AccountName=");
19    v0.append(this.storageAccountName);
20    v0.append(";AccountKey=");
21    v0.append(this.storageAccountKey);
22    String v1 = v0.toString();
23    if(Utils.isNetworkAvailable(this.imgContext)) {
24      new AsyncTask(v1) {
25        String val$conStr;
26        public AsyncTask(String arg1){
27          this.val$conStr = arg1;
28        }
29        protected void doInBackground(Void[] arg17) {
30          String v0 = this.val$conStr;
31          CloudStorageAccount v7 = CloudStorageAccount.parse(v0);
32 ...
33 package com.appname.Utils
34 public class Utils {
35   public static String getStringFromResources(Context arg1,
36        int arg2) {
37     Resources v0 = arg1.getResources();
38     String v1 = v0.getString(arg2);
39     return v1;
40   }
41 ...
```

Figure 2: A Sample Piece of Decompiled Code for Azure Storage Access from a Real Android App.

(i) Microsoft Azure Storage, (ii) Azure Notification Hubs, and (iii) Amazon AWS S3.

(I) Key Misuses in Azure Storage. Microsoft Azure cloud provides two kinds of keys for developers to access its storage.

- **Account Key**. An Azure storage account provides a unique namespace to store and access various data in Azure Storage such as Tables, Queues, Files, Blobs, and virtual machine disks billed to a particular user. This account key works like a root key, which has full access to the cloud storage. An example of such a key is presented in the first row of Table II.
- **Shared Access Signature (SAS)**. A SAS provides delegated access to resources that belong to a specific Azure storage account. Unlike the account key, developers can configure the permissions (*e.g.*, read, write) of SAS so that the key can only access certain resources with limited permissions. An example of SAS is presented in the 2nd row of Table II.

Even though the account key and SAS have entirely different formats and use cases, we find developers actually have misused them. Obviously, developers should have exclusively used SAS in the app and kept the account key confidential. For instance, when a user is trying to access a private file from the mobile app, developers should assign a SAS, which has the minimum permission to access the particular files for this user. However according to our experiments, many developers directly use their account key in the mobile apps. Unfortunately, the account key can be easily extracted by attackers through reverse engineering. As shown in Figure 2, we extracted code from a real Android app, whose name is

Figure 3: How to Use Azure Notification Hub.

anonymized on purpose. The code is used for downloading some images from the cloud storage. We can see that the key (at line 31) used in this app is an account key (by looking at its format at line 18), which means anyone with this key can access the entire storage allocated to this particular account.

(II) Key Misuse in Azure Notification Hubs. With cloud APIs, developers can easily send messages to specific users or broadcast to all users. Microsoft Azure provides a notification hub with two APIs for message notification. As illustrated in Figure 3, to use Azure Notification Hub, developers first need to register a channel, and then the mobile app just invokes the API to listen to the developer registered channel using a listening key; after that, the app server can push notifications to the channel registered in the Hub (using a full access key), which will further relay this message to all channel listeners. There are two keys involved:

- **Full Access Key**. The full access key works like a "root" key, and has full access to send or listen for notifications on the channel. An example of such a key is shown in the 3rd row of Table II.
- **Listening Key**. The listening key has limited privileges and can only listen to the notifications registered in a particular channel. The last row in Table II shows an example of a listening key.

Clearly, a listening key should only be used by the mobile app, whereas the full access key should only be used by app servers. However, we find that some developers are using full access keys directly in the mobile apps. While the key misuse in Azure Notification Hubs may not directly lead to data leakage attacks, we have to stress that once attackers extract the full access key they still have access to powerful capabilities. For example, they can easily push phishing messages to steal data from other app users.

(III) Key Misuse in AWS. Amazon AWS also provides a number of keys for mobile apps to access the AWS resources (*e.g.*, the S3 storage) billed to the developers. One of the keys is the root access key (or root account key), which has full access to all of the resources under a particular AWS account. Since this root key has all privileges, it should be kept secret. Unfortunately, we also noticed this root key being used in mobile apps. Once an attacker extracts this root key, she has full access to the entire storage.

B. Misconfiguration of User Permissions in Authorization

In addition to the misuse of keys in authentication, misconfiguration of user permissions in authorization can also result in data leaks. Typically, authentication only tells the system who the user is, and it is authorization that decides which specific resources an authenticated user can access. Lack of authorization or incorrect configuration can make the authorization layer useless, thereby leading to data leaks.

```
{
  "rules": {
    "users": {
      "$uid": {
        ".read": "$uid === auth.uid",
        ".write": "$uid === auth.uid"
      }
    }
  }
}
```
Figure 4: A Correct Firebase Authorization Rule

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```
(a)

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}
```
(b)

Figure 5: Two Misconfigured Firebase Authorization Rules

While misconfiguration of authorization is not a new problem and has been studied for many years, this problem becomes more critical when using cloud APIs. In particular, when developing mobile apps without cloud APIs, different developers can implement authorization systems in completely different ways. Even though vulnerabilities may be present, it is quite challenging for attackers to systematically exploit many diverse implementations. However, when developers are using cloud APIs, adversaries can easily launch attacks; since many developers are using the interfaces of only a few cloud services, attackers can just focus on these interfaces to systematically identify vulnerabilities.

Meanwhile, unlike in authentication, in which developers have a limited playground when using cloud APIs (*e.g.*, the mistakes only come from key misuses), there are a variety of ways for developers to configure authorization, making the authorization configuration much harder and more error-prone. For instance, Google even provides a language for developers to specify the user permissions in authorization. It is not surprising that developers will make mistakes: we found (i) apps with Firebase that have misconfigured user permissions in the back-end. In addition, we found (ii) apps with AWS that suffer from this misconfiguration problem as well.

(I) Misconfiguration of User Permissions in Firebase. When using Firebase, developers have to define user-specific access control polices, *i.e.*, the "rules". An example of such a rule is shown in Figure 4. Note that Firebase is a real time database that organizes data in a hierarchy structure with JSON. According to this example rule, there is a node named "users" in the database: when a user attempts to access (read/write) one of its child nodes that has an associated `$uid`, the system will only grant access if the user's uid equals the `$uid` of the child node for this specific read and write operation. Unfortunately, we notice that not all developers follow the correct way to write the rules. For instance, as shown in Figure 5, a developer can write rules to just ignore the checks, or just check whether a user is authenticated. These are obviously insecure rules, as they imply that any (authenticated) user has full access to the entire database.

(II) Misconfiguration of User Permissions in AWS. With AWS, Amazon provides Identity and Access Management (IAM) for user management and permission configurations. To securely access the resources billed to a particular mobile app

developer, an IAM user needs to be created. The developer can configure the permissions for IAM users, and each IAM user can generate two secret access keys to access the specified resources. While using IAM users appears to be secure, developers may in fact over provision the permissions for an IAM user, such as granting full access to a particular storage, which will lead to data leaks.

IV. Problem Statement and Overview

After the discovery of the root causes of the data leaks in the cloud, we would like to develop techniques to systematically identify the leaks. Since we do not have any access to the mBaaS cloud implementations (only to their SDKs and APIs), we can only start from the front-end of the services (*i.e.*, the mobile apps) to inspect how mobile apps use the various keys and also infer the permission configurations based on server responses. Therefore, we must design a mobile app exclusive approach. In this section, we provide an overview of the problem we aim to solve and the solutions and insights we have. The detailed design and implementation of our tool is presented in the next section.

A. Problem Statement

As illustrated in Figure 1, we notice that developers must embed the app key in the app, and then use it to invoke cloud mBaaS cloud APIs. The data leakage vulnerabilities we aim to identify are mainly caused by key misuse and permission misconfiguration. Therefore, *the first problem we must solve is how to systematically identify various keys used by mobile apps*.

At first it might appear trivial to identify the API keys used by mobile apps by inspecting each cloud API (e.g,. the `CloudStorageAccount.parse` API used at line 31 in Figure 2). However, the keys might not be directly visible and could have gone through multiple string acquisitions (*e.g.*, at line 9-12) and concatenations (*e.g.*, at line 18-21). In addition, there are millions of mobile apps, so we have to design a scalable approach. Therefore, *the second problem we must solve is how to identify the relevant key strings that are used by mobile apps*.

Also, increasingly mobile apps are using obfuscation to thwart app reverse engineering and repackaging [45], [43], [42]. It might be possible that app developers have obfuscated the APIs we aim to inspect. As such, we cannot simply scan the app code for API signatures (*e.g.*, `CloudStorageAccount.parse`); instead, we must design an obfuscation-resilient approach. It would also be interesting to know whether obfuscated apps are vulnerable to data leaks, given the fact that app developers using obfuscation are potentially more aware of security issues. Therefore, *the third problem we need to solve is how to design an obfuscation-resilient approach to identify cloud APIs and key strings of our interest*.

Finally, after we have extracted the keys, we have to identify the type of each key (*e.g.*, a root key or an app key). Once we have determined the types of the keys, we must also verify whether the apps have misused them without accidentally accessing any private data stored in the cloud. Similarly, we also must identify user permission misconfigurations during authorization without leaking any data. Therefore, *the final problem we (as third-parties) must solve is to design a verification approach with zero data-leakage to confirm the existence of data leaks in the cloud.*

B. Our Solutions

Recognizing the Keys by Cloud API Identification. To identify keys that are used by an app, we can actually infer them from the parameters of the well-known APIs that are used by the app. Note that each cloud provider has offered a set of APIs in their SDKs for mobile app development. As illustrated in Table III, the APIs that take parameters with various keys are actually quite limited, and we acquire this list based on our best understanding of the corresponding SDKs. Therefore, if we are able to recognize these APIs, then we can identify the keys used by the app in the corresponding parameters.

However, the APIs listed in Table III can be obfuscated. Interestingly, we find that there are often two strategies used in API obfuscation:

- **Renaming names** involved in the API, such as the sub-package name, class name, function name, and variable name, from the standard name to some meaningless characters. This is often achieved by some automated obfuscation tools (*e.g.*, Dexprotector [7], Dexguard [5]).
- **Removing the functions/APIs** that are never used. Since the non-used functions/APIs can often reveal the packages and classes used by the app, removing these functions from the apps by automated tools (*e.g.*, Proguard [12], [15]) can further help hide the APIs of interest.

Therefore, we propose to build an obfuscation-resilient function signature for each function in the APK (including its library) in order to identify the cloud APIs in Table III. Our signature ignores the names of packages, classes, functions, and variables. Instead, a function's signature is the hash of the strings that are composed of the types of the parameters, local variables, and return values, as well as the signatures of callees.

Using String Analysis to Identify the Value of Keys. Having identified the APIs of our interest, we cannot directly extract the values of the corresponding parameters from the app code. For instance, we cannot directly extract the value of `v0` from `CloudStorageAccount.parse` in Figure 2, as this value is computed from multiple string operations. While we can use dynamic analysis to execute the app and extract the value at runtime, such an approach does not scale well, especially considering that we have millions of mobile apps. Therefore, eventually we decided to take a static analysis approach and propose a targeted string value analysis to identify the used keys. At a high level, our string value analysis can be considered a particular case of value set analysis [24]. It involves backward slicing and string related operation analysis.

Zero-data-leakage Vulnerability Verification. After we have retrieved the values of the keys used by the cloud APIs, next we have to infer the types of the keys. For some cloud services (*e.g.*, Azure Storage), app keys and root keys are in different formats (as shown in Table II), and we can easily tell whether a key of our interest is a root key. However, for some other cloud services (*e.g.*, AWS), app keys and root keys have the same format. To deal with this problem, a straightforward approach is to send a request to the server by using the key to access some root user exclusive data. If we are able to retrieve these data, then it implies there is a data leakage vulnerability. However, such an approach would violate the ethics of accessing private-sensitive data.

Cloud Service	APIs	Definition	Indexes of The String Parameters of Our Interest
AWS	1*	TransferUtility: TransferObserver downloadUpload(String, String, File)	0
	2*	AmazonS3Client: void S3objectAccess(String, String, ...)	0
	3	CognitoCredentialsProvider: void <init>(String,String,String,String,...)	1
	4	BasicAWSCredentials: void <init>(String,String)	0,1
Azure	5	MobileServiceClient: void <init>(String,Context)	0
	6	MobileServiceClient: void <init>(String,String,Context)	0,1
	7	NotificationHub: void <init>(String,String,Context)	1
	8	CloudStorageAccount: CloudStorageAccount parse(String)	0
Firebase	9	FirebaseOptions: void <init>(String,String,String,String,String,String,String)	0,1,2,5
	10	FirebaseOptions: void <init>(String,String,String,String,String,String)	0,1,2,5

Table III: Targeted mBaaS Cloud APIs of Our Interest. In total, there are 32 APIs. Due to limited space, we use API 1* and 2* to actually represent two sets of APIs. The complete list of the APIs of these two sets is presented in Appendix in Table IX.

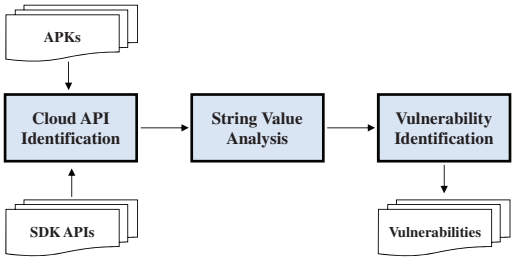

Figure 6: An Overview of Our LeakScope.

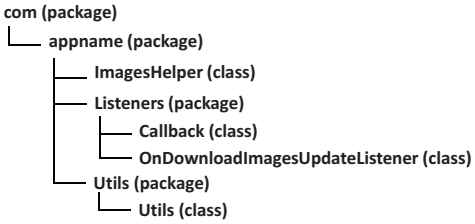

Figure 7: The Hierarchical Structure of the Package Tree of The Sample Code in Figure 2.

Fortunately, we have another observation: we notice that the server typically will return different response messages when accessing nonexistent data in the cloud with a root key versus a regular user key. As such, to verify whether a key is a root key, we will send a request to the cloud with the key to retrieve some nonexistent data. If the key is an app key, the return message is usually in the form of "`permission denied`", and if the key is a root key, the return message is usually something like "`data not found`". In either case, no real data is leaked, but we have inferred the types of the keys. We can also use the same approach to verify the authorization misconfigurations.

V. Design and Implementation

We have built a tool called LeakScope to automatically detect data leakage vulnerabilities when given a mobile app. An overview of LeakScope is presented in Figure 6. It consists of three key components: Cloud API Identification (§V-A), String Value Analysis (§V-B), and Vulnerability Identification (§V-C). In this section, we describe how we design and implement (§V-D) these components.

A. Cloud API Identification

It is important to identify the APIs used by an app, especially those listed in Table III. As discussed in §IV-B, we need to design an obfuscation-resilient approach to identify the cloud APIs. Fortunately, we notice that there are at least two invariants that are preserved regardless of the obfuscations for a given function (or method).

- The first invariant is the hierarchical structure (or the shape, the layer) of the types and package trees. Note that each class, method, parameter, and variable all have types (e.g., the type of `arg5` is `com.appname.Listeners.Callback` at line 15 in Figure 2). While the names of the types can be obfuscated (except the name of system types), the hierarchical structure of the types will not change. For instance, as shown in Figure 7, type `Callback` is stored at the fourth layer starting from the package `com`.
- The second invariant is the caller-callee relation in a method. For instance, method `downloadImage` will call functions such as `StringBuilder.append` and `Util.isNetworkAvailable`. We can build each callee's signature recursively, and then merge them to build the signature for the caller, namely `downloadImage` in this example.

Therefore, if we can encode these two invariants and add them together to build a signature for a function (including APIs in linked non-system libraries), then we can just search for these signatures in the bytecode of mobile apps to find the cloud APIs we need. Inspired by LibScout [23], in which a Merkle hash tree is used to build a library signature (for 3rd party library detection in Android apps), we use the hash of the encoded invariants (*i.e.*, the hierarchical structure of the types and package trees, and the caller-callee relation) as the function/API signature for the API detection. Note that we cannot directly use LibScout as it focuses on the detection of third-party libraries. In addition, LibScout only generates coarse-grained signatures for each class, whereas we have to generate fine-grained signatures for each function.

Generating Signatures for Each Function. More specifically, as shown in Algorithm 1, for a given function, we will generate a signature (GenFunSig) by using the MD5 hash of the encoded type string of (i) the function's belonging class (*i.e.*, home class) (line 5-6), (ii) its arguments (line 7-8), (iii) its local variables (line 9-10), (iv) its return value (line 11-12), and (v) recursively all of its callees (line 13-18). If a callee is an Android system function (not obfuscated), then we directly

Algorithm 1 Function Signature Generation

```
1: Input: f_0: target function; C_s: system classes; F_s: system functions
2: procedure GenFunSig(f_0, C_s, F_s)
3:     L ← ∅
4:     fBuf← ∅
5:     t_0 ← GetHomeClassType(f_0)
6:     fBuf ← fBuf ∪ TypeEncoding(t_0, t_0, C_s)
7:     for t_p ∈ GetParameterType(f_0) do
8:         fBuf ← fBuf ∪ TypeEncoding(t_0, t_p, C_s)
9:     for t_v ∈ GetLocalVarType(f_0) do
10:        fBuf ← fBuf ∪ TypeEncoding(t_0, t_v, C_s)
11:    t_r ← GetReturnType(f_0)
12:    fBuf ← fBuf ∪ TypeEncoding(t_0, t_r, C_s)
13:    for f_i ∈ GetCallee(f_0) do
14:        if f_i ∈ F_s then
15:            L ← L∪ name(f_i, argType(f_i), retType(f_i))
16:        else
17:            L ← L∪ GenFunSig(f_i, C_s, F_s)
18:    fBuf ← fBuf ∪ Sort(L)
19:    return MD5 (fBuf)
20: Input: c_h: home class; c_t: target class; C_s: system classes;
21: procedure TypeEncoding(c_h, c_t, C_s)
22:    L ← ∅
23:    tBuf ← ∅
24:    if c_t ∈ C_s then
25:        tBuf ← tBuf ∪ name(c_t)
26:    else
27:        rp ← NormalizedRelativePath(c_h, c_t)
28:        tBuf ← tBuf ∪ rp
29:        c_p ← GetSuperClass(c_t)
30:        tBuf ← tBuf ∪ TypeEncoding(c_h, c_p, C_s)
31:        for c_i ∈ GetInterfaces(c_t) do
32:            L ← L∪ TypeEncoding(c_h, c_i, C_s)
33:        tBuf ← tBuf ∪ Sort(L)
34:    return tBuf
```

add the name of this function including the type name of its arguments and return value to our type string (line 14-15).

To perform the string encoding (TypeEncoding) of the hierarchical structure of the types and package trees (line 20-34), basically we take the home class (c_h) to which the type belongs, the target class (c_t), and system classes (C_s) as input to encode the target type c_t. Note that in Java bytecode, all of the types are defined as classes. We directly return the string name of the type if c_t is a system class (line 24-25) as its name cannot be obfuscated. Otherwise, we take the encoded string that combines the normalized relative path between c_t and c_h (line 27-28), recursively the encoded string of the superclass of c_t (line 29-30), and the interfaces of c_t (line 31-33), as the final type string.

Note that to uniquely encode the type of a given class, we would like to use as much reliable information as possible. Again, we cannot use any of the class names (except the system classes) and we must normalize them. Since a class typically has a super class, we would like to include the type encoding of its superclass, recursively. Meanwhile, a class could have defined a number of interface functions [19]. Note that an interface is a collection of empty functions (no implementation body). Any class implementing an interface will inherit these functions. That is why eventually our type encoding algorithm considers the name and path normalization, superclass, and class interfaces. We do not include other information in a class, such as its fields and methods, in the signatures since they can be removed by tools such as Proguard [15].

When performing the normalization, we replace any non-system defined names with the symbol 'X', similar to LibScout [23]. In addition, our algorithm takes a normalized relative path into our type string. For instance, when processing the type encoding of the first argument of function `downloadImage`, whose home class is `ImagesHelper`, we take a normalized relative path to encode the type `com.appname.Listeners.Callback`. More specifically, according to Figure 7, starting from the home class `ImagesHelper`, we reach `com.appname.Listeners.Callback` by `../Listeners/Callback`. After normalizing all non-system names into 'X', we get a normalized path with string "`../X/X`". Meanwhile, the superclass of `com.appname.Listeners.Callback` is actually `java.lang.Object`, which is a system class (cannot be obfuscated). Therefore, eventually, for the type encoding of `arg5`, we get the string "`../X/X#java.lang.Object`" where # denotes string concatenation. Similarly, we get a type string for its second argument, local variables, and callees, respectively. Together, we have created a unique hash for function `downloadImage`.

Identifying the Cloud API with the Signatures. With our GenFunSig algorithm, we generate signatures for all functions, including the cloud APIs (initially acquired from the SDK libraries) for a given app. We then search for the signatures of the cloud APIs in the mobile apps; if a signature matches, we identify the corresponding cloud API of our interest.

B. String Value Analysis

Having identified the cloud APIs of our interest, we are then able to identify where each API is called and further pinpoint the parameters that contain the authentication keys in the mobile app. However, we are not be able to directly observe their values since we use static analysis. Therefore, we have to develop a targeted string value set analysis (VSA) [24] to reveal the possible values of the keys. Note that VSA is a technique that analyzes the possible values for registers and memory addresses at the x86 binary code level. We cannot directly use it to solve our problem, and instead we have to customize it to reveal the string values in the context of mobile app bytecode. At a high level, our string value analysis needs to perform the following *inter-procedural backward slicing* and *string value computation* statically.

Inter-procedural Backward Slicing. Program slicing [37] is a widely used program analysis technique and has been used to solve many important security problems such as software vulnerability diagnosis (e.g., [38]) and automatic patch generation (e.g., [35], [44]). In our string value analysis, we have to first identify the variables and instructions that are related to the computation of the final strings of our interest by applying backward slicing of the Java bytecode.

More specifically, the first step of our analysis is to build an intro-procedural control flow graph (CFG) for each method/function, where nodes represent the contiguously executed bytecode instructions and edges represent the control flow transfers within the function. Then starting from a variable of our interest, *e.g.*, `v0` of API `CloudStorageAccount.parse`, we iterate the instructions in the CFG backwards: if there are any variables that contribute to the computation of `v0`, we add them to our data dependence graph (DDG) and meanwhile push the involved instructions and variables into a string computation stack, which is an internal last-in-first-out data structure maintained by LeakScope that is used to track the order of the execution of

the string operations that contribute to the final string values; if there is any function call, we perform a context-sensitive inter-procedural analysis and recursively analyze the callees. We keep iterating the CFG until we reach a fixed point, namely when our DDG cannot be expanded further. In our running example in Figure 2, all of the statements that are highlighted in red are involved in the computation for the string value `v0` used at line 31.

String Value Computation. With the tracked DDG and string computation stack, next we need to compute the values of the final strings of our interest. Starting from the top of the string computation stack, we pop the involved variables and instructions based on the CFG and our DDG, and we forward execute the involved string operations based on the instruction semantics until the stack is empty or the string value is fully determined.

Forward execution is not real execution and is instead based on the API summaries of the string operations. For instance, if the involved instruction is a string `append` API, we perform the string append operation; if it is a `getString`, we know it is used to read a string from an `xml` file, and we then perform the read operation of the specified string from the `xml` file and return its result. Note that strings are system defined classes, and the corresponding APIs are not obfuscated.

Back to our running example in Figure 2, the last pushed variables on the stack are `StorageAccountKey` and `StorageAccountName`. Then, starting from there with our tracked DDG, we perform a forward string analysis to compute the value of `this.StorageAccountName` and `this.StorageAccountKey` at lines 9 and 12 by executing the API summary of `getResources` and `getString` (lines 37-38). Next, we compute the value of `v0` (line 17-21), `v1`, `this.val$conStr`, and finally the value of `v0` at line 31.

C. Vulnerability Identification

After obtaining the keys identified by LeakScope, next we would like to detect data leakage vulnerabilities in the cloud services. The detection is cloud specific, and we have designed the following algorithms to detect key misuses and permission misconfigurations for services that use Azure, AWS, and Firebase.

(I) Detecting Key Misuses in Azure (Storage and Notification Hub). As discussed in §III-A, if there are root keys or full access keys in mobile apps, then an attacker can easily use these keys to leak data. Therefore, we do not need to probe the back-end to confirm the vulnerabilities. As long as we identify root keys or full access keys in a mobile app, we know the cloud service is vulnerable.

(II) Detecting Key Misuses in AWS. An AWS root key has full access to the corresponding AWS account, which means all resources under that account can be accessed by an adversary if we identify a root key in an app. We found that a root key has the permission to get some AWS instance information with the following rest API: `https://ec2.amazonaws.com/?Action=DescribeInstances&InstanceId.1=X`, where `X` is the ID of the target instance. In contrast, an app key does not have this permission. Therefore, to detect key misuses in AWS, we set `X` to be a nonexistent ID. When we send a request alone with the key identified, we will receive the error message "`InvalidInstanceID`" if the key is a root key or "`UnauthorizedOperation`" if the key is an app key.

(III) Detecting Permission Misconfiguration in Firebase. There are two typical permission misconfiguration rules as shown in Figure 5: (a) no authentication check (the database is entirely open to anyone), and (b) no permission check (only checks whether the user is authenticated). We use the following algorithms to detect them:

- **Detecting "Open" Database**. When a developer mistakenly specifies a read policy as `".read": "true"`, then anyone can read the database. Firebase provides a REST API to access data from the database in a given "path". By setting the "path", a user can read specific data. If we set "path" to "root", then we can read the entire database. However, we do not have to read the entire database, as Firebase supports queries with the `indexon` field. If we set this field to a nonexistent value when attempting to read the "root" path, no data will leak, but we can still confirm the data leaks as the returned error message is different when the user has root read permissions.
- **Detecting No Permission Check**. When a data record read policy is `".read": "auth != null"`, we must use an authenticated user in order to perform the `indexon`-based leakage testing. It would be a huge engineering challenge to register a legitimate user in each corresponding cloud service. Fortunately, we noticed that Firebase also provides a number of cloud APIs for user registration, *e.g.*, by using email/password, phone number, Google/Facebook SSO, *etc.*, in addition to developer customized registration methods. We therefore can directly invoke these Firebase APIs to register legitimate users in the corresponding service if the tested app has used them. After that, we authenticate the registered user and then perform the `indexon`-based test.

(IV) Detecting Permission Misconfiguration in AWS. AWS keys are used to access the specified resources under particular access control policies. In AWS, there are several types of resources. Among these resources, we only focus on the detection of permission misconfiguration of S3 Storage. Note that several recent high profile data leaks (e.g., [33]) were from S3. To perform our test, we have to collect not only AWS keys, but also the S3 Storage names. Therefore, we have to apply our string value analysis for both of them. With the identified AWS keys and storage names, we directly invoke an AWS API `HEAD Bucket` [9] to verify whether a key has permission to access the storage or not. If so, a data leak is detected.

D. Implementation

We have implemented LeakScope[1] atop `dexlib2` [6] and `soot` [16]. In particular, to implement our Cloud API Identification, we leverage `dexlib2`, a lightweight APK static analysis framework that allows easily parsing of dex files to build function signatures. We build our String Value Analysis atop `Soot`, a powerful framework for analyzing Java bytecode with particularly useful features such as data flow analysis. The Vulnerability Identification is quite straightforward; we just wrote a python script to send the requests and parse the

[1]The source code of LeakScope is made available at https://github.com/OSUSecLab/LeakScope.

	Total #Apps	%	Non-Obfuscated #Apps	%	Obfuscated #Apps	%
w/ Cloud API	107,081	-	85,357	79.71	21,724	20.29
w/ AWS only	4,799	4.48	4,548	5.33	251	1.16
w/ Azure only	899	0.84	720	0.84	179	0.82
w/ Firebase only	99,186	92.63	78,475	91.94	20,711	95.34
w/ AWS & Azure	3	0.00	2	0.00	1	0.00
w/ AWS & Firebase	1,973	1.84	1,427	1.67	546	2.51
w/ Azure & Firebase	210	0.20	174	0.20	36	0.17
w/ Three Services	11	0.01	11	0.01	0	0.00

Table IV: Result of Our Cloud API Detection.

response. In total, LeakScope consists of around 6,000 lines of our own Python and Java code.

VI. Evaluation

In this section, we present our evaluation result. We first describe our experiment setup including how we collect the mobile apps in §VI-A, describe our detailed result in §VI-B, and finally provide an analysis of the identified vulnerabilities and the false positives of our approach in §VI-C.

A. Experiment Setup

Collecting the Mobile Apps. We focus on the Android apps published in Google Play. To obtain an app from Google Play, we have to provide the app package name. As such, we first developed a python script atop the `scrapy` [14] framework to crawl for each app package name. After two weeks of crawling, we retrieved about 1.9 million app names in May 2017. Then we crawled the entire package contents of 1,609,983 free Android apps within two months. Note that we could not download all 1.9 million apps due to restrictions such as paid apps or certain apps only being available in certain countries. In total, these 1.6 million mobile apps consumed 15.42 TB of space on our hard drive.

Environment Setup. Our experiments were conducted on seven workstations, each equipped with Intel Xeon E5-2640 CPU with 24 cores and 96 GB memory and running Ubuntu 16.04. We do not need any real mobile devices since LeakScope is mostly a static analysis tool, and only the Vulnerability Identification component needs to communicate with the cloud servers using dynamic analysis. All of our experimental data including the target apps and intermediate results are stored in a Network Attached Storage with 34.90TB hard drive space.

B. Experiment Result

In total LeakScope spent 6,894.89 single CPU computation hours analyzing these 1,609,983 apps, which consumes 2.56 TB of storage for storing the intermediate results. Among the tested apps, eventually LeakScope detected 15,098 unique mobile apps (with 17,299 vulnerabilities in total), whose cloud servers are subject to data leakage attacks. In the following, we present detailed results based on how each component of LeakScope performs.

1) **Cloud API Identification:** We first evaluated our Cloud API Identification with the tested 1,609,983 apps. Since our approach is obfuscation-resilient, it clearly works for non-obfuscated apps as well. In total, our Cloud API Identification generated 39,617,809,277 function signatures and identified 107,081 mobile apps that used some of the 32 cloud APIs of our interest. Among these apps, 21,724 of them (20.29%) are obfuscated. Therefore, as reported in Table IV, we separated our experimental results into two sets: apps without obfuscation (85,357 apps), and apps with obfuscation, in order to understand whether obfuscated apps provide better protection against data leakage attacks.

We also reported the detailed distributions of the cloud services used by the mobile apps. In particular, among these 107,081 apps, 4,799 (4.48%) exclusively use Amazon AWS, 899 (0.84%) exclusively use Microsoft Azure, and 99,186 (92.63%) exclusively use Google Firebase. There are also 3 apps that use both AWS and Azure, 1,973 with AWS and Firebase, and 210 with Azure and Firebase. Interestingly, there are also 11 apps that use all three cloud services. The detailed breakdown for non-obfuscated and obfuscated apps are reported from the 4th to 7th column, respectively.

It is quite surprising that the vast majority (over 90%) of the mobile apps actually used Google Firebase for their back-end services, at least according to the results reported in Table IV. We consulted with the Google Firebase team when we made our responsible disclosure. They informed us that part of the reason for this is that there may be a significant portion of mobile apps that have used Amazon's or Microsoft's clouds, but not their mBaaS clouds (*e.g.*, they may use their IaaS clouds instead). Our mBaaS cloud API Identification cannot identify these clouds.

2) **String Value Analysis:** Among the 107,081 apps, our String Value Analysis statically computed 631,551 strings of the parameters of our interest, and our detailed performance results are reported in Table V. In particular, we report the string parameters of our interest in the 2nd column, followed by the corresponding APIs to which each string parameter belongs in the 3rd column (the original definition of these APIs is presented in Table III and Table IX in the Appendix). Then for both non-obfuscated and obfuscated apps, we report how many of the corresponding APIs were called in the 4th and 8th columns, how many apps have called these APIs in the 5th and 9th columns, how many of the parameter strings had their values eventually resolved in the 6th and 10th, and the corresponding percentages in the 7th and 11th columns.

We can observe from Table V that we are interested in a parameter called `bucketName` from 2 different sets of APIs in AWS. This is because we need `bucketName` to locate the corresponding S3 storage for authorization vulnerability verification. We are also interested in `identityPoolId`, which is used to detect the permission misconfiguration vulnerability in AWS, and `accessKey` and `secretKey` are also clearly of direct interest. For Azure, we are interested in parameters `appURL`, `connectionString`, and `appKey`. For Firebase, we are interested in `google_app_id`, `google_api_key`, `firebase_database_url`, and `google_storage_bucket`. The app may call each of these APIs one or more times at different places.

Based on how the strings are used by the app code, String Value Analysis has resolved the vast majority of the string values, as shown in the 7th and 11th column in Table V for both non-obfuscated and obfuscated apps. It would also be interesting to understand why not all the parameter strings could be resolved by our String Value Analysis. Our further investigation revealed that there are two reasons for this. The first one is that many of the unresolved values of these parameters were actually retrieved from the Internet. This case is particularly common for Firebase, as Google actually

	String Parameter Name	APIs	Non-Obfuscated #API-Call	Non-Obfuscated #APP	Non-Obfuscated #Resolved Str.	Non-Obfuscated %	Obfuscated #API-Call	Obfuscated #APP	Obfuscated #Resolved Str.	Obfuscated %
AWS	bucketName	1*	2,460	1,229	2,190	89.02	398	1,229	321	80.65
	bucketName	2*	2,069	1,703	2,045	98.84	444	439	442	99.55
	identityPoolId	3	3,458	3,458	3,315	95.86	291	291	266	91.41
	accessKey	4	3,280	1,769	2,650	80.79	277	203	199	71.84
	secretKey	4	3,280	1,769	2,646	80.67	277	203	197	71.12
Azure	appURL	5	185	39	185	100.00	11	4	11	100.00
	appURL	6	824	316	817	99.15	32	21	32	100.00
	appKey	6	824	316	809	98.18	32	21	31	96.88
	connectionString	7	700	513	643	91.86	207	189	200	96.62
	connectionString	8	345	97	303	87.83	29	21	22	75.86
Firebase	google_app_id	9	2,378	1,228	2,222	93.44	935	908	934	99.89
	google_api_key	9	2,378	1,228	2,230	93.78	935	908	927	99.14
	firebase_database_url	9	2,378	1,228	2,039	85.74	935	908	882	94.33
	google_storage_bucket	9	2,378	1,228	2,050	86.21	935	908	882	94.33
	google_app_id	10	154,664	78,859	143,735	92.93	20,723	20,385	20,657	99.68
	google_api_key	10	154,664	78,859	137,589	88.96	20,723	20,385	20,199	97.47
	firebase_database_url	10	154,664	78,859	118,786	76.80	20,723	20,385	18,077	87.23
	google_storage_bucket	10	154,664	78,859	119,606	77.33	20,723	20,385	18,041	87.06

Table V: Result of Our String Value Analysis for the Parameters of Our Interest.

	The Root Cause	Non-Obfuscated #Apps	Non-Obfuscated %	Obfuscated #Apps	Obfuscated %
Azure	Account Key Misuse	85	9.37	18	8.33
	Full Access Key Misuse	101	11.14	12	5.56
AWS	Root key Misuse	477	7.97	92	11.53
	"Open" S3 Storage	916	15.30	195	24.44
Firebase	"Open" Database	5,166	6.45	1,214	5.70
	No Permission Check	6,855	8.56	2,168	10.18

Table VI: App Statistics with the Detected Vulnerabilities

recommends that developers retrieve keys from the remote servers [3]. Without dynamic analysis of the apps, we could not infer their values. The second reason is that some apps are using cryptographic functions to protect the string, which we cannot resolve with static analysis.

3) **Vulnerability Identification:** With the identified keys and strings of our interest, our 3rd component, Vulnerability Identification, then detects the vulnerabilities based on our zero-data-leakage policies described in §V-C and has identified 17,299 vulnerabilities in total. Note that one app may have multiple data leakage vulnerabilities, and we count the vulnerabilities based on the vulnerable services.

- **Key Misuse Vulnerabilities.** As discussed in §III-A, these vulnerabilities mainly exist in the Azure and AWS clouds. Based on the app key value, and the format of the keys, we directly detect vulnerabilities in Azure if we notice that the app key is either an account key or full access key. The statistics of the vulnerable apps in Azure is presented in Table VI (the first two rows). We can see that among the 907 non-obfuscated Azure apps, 186 of them (20.51%) have misused the keys; for the 216 obfuscated apps, 30 of them (13.89%) contain a data leakage vulnerability. For the AWS root key misuse, we detect 477 vulnerable apps out of 5,988 (7.97%) non-obfuscated AWS apps, and 92 out of 798 (11.53%) obfuscated apps, as presented in the 3rd row of Table VI.
- **Permission Misconfiguration Vulnerabilities.** This type of vulnerability mainly exists in the AWS and Firebase cloud servers. As reported in the 4th row of Table VI, we detect 916 vulnerable apps out of 5,988 (15.30%) non-obfuscated apps, and 195 out of 798 (24.44%) obfuscated apps. For the "Open" database in Firebase, we detect 5,166 vulnerable apps out of 80,087 (6.45%) non-obfuscated apps, and 1,214 out of 21,293 (5.70%) obfuscated apps. For the No Permission Check vulnerabilities in Firebase, we detect 6,855 out of 80,087 (8.56%) non-obfuscated apps, and 2,168 out of 21,293 (10.18%) obfuscated apps.

We can notice from Table VI that the most vulnerable category (in terms of percentage) is from permission misconfiguration of the "Open" S3 Storage of AWS: 15.30% for non-obfuscated apps and 24.44% for obfuscated apps. It can be observed for Azure that obfuscated apps tend to be less vulnerable (13.89% vs. 20.51%). However, in AWS and Firebase, obfuscated apps are even more vulnerable (except the "Open" Database for Firebase). This is likely because the misconfiguration errors are product-specific and have less connection with the user's security expertise.

C. Vulnerability Analysis

Severity Analysis. Next, we would like to study the severity of the vulnerabilities among the mobile apps we discovered. We use the number of the downloads of the vulnerable apps to characterize the severity: the higher number of downloads, the more severe the vulnerability. To this end, we count the number of downloads of the vulnerable apps in each download category (*e.g.*, between one billion to five billion). This result is reported in the last four columns of Table VII. For the very popular apps (we define an app is very popular if its total number of downloads exceeds one million) that have used cloud APIs, 569 of them are subject to the data leakage attack. Among these apps, 10 of them have downloads between 100 million and 500 million, 14 with 50 million to 100 million, and 80 with 10 million to 50 million. Clearly, the data leakage vulnerabilities we studied are quite concerning. If an attacker has exploited them, then billions of sensitive data records could have been leaked.

Obfuscation vs. Non-Obfuscation. Since we are able to differentiate non-obfuscated and obfuscated apps, we also would like to understand the effect of obfuscation with respect to app security. It is interesting to observe that obfuscation is typically applied to top downloaded apps. As shown in Table VII: the higher number of downloads an app has, the more

#Downloads	# Non-Vulnerable Apps				# Vulnerable Apps			
	Azure	AWS	Firebase	Obfuscated%	Azure	AWS	Firebase	Obfuscated%
$1,000,000,000 - 5,000,000,000$	0	0	1	100.00	0	0	0	0.00
$500,000,000 - 1,000,000,000$	0	0	3	66.67	0	0	0	0.00
$100,000,000 - 500,000,000$	0	1	35	58.33	0	1	9	50.00
$50,000,000 - 100,000,000$	0	4	67	45.07	0	2	12	71.43
$10,000,000 - 50,000,000$	2	35	480	47.78	1	4	75	50.00
$5,000,000 - 10,000,000$	3	32	467	37.85	1	6	66	38.36
$1,000,000 - 5,000,000$	16	136	2,405	32.15	2	21	369	30.10
$500,000 - 1,000,000$	10	105	1,823	29.36	1	29	260	28.28
$100,000 - 500,000$	65	356	6,987	26.01	14	66	1,026	26.13
$50,000 - 100,000$	42	249	4,608	25.52	11	50	695	25.13
$10,000 - 50,000$	167	679	12,868	24.85	21	174	1,862	21.88
$5,000 - 10,000$	82	369	6,090	24.05	11	100	770	23.61
$1,000 - 5,000$	272	976	15,920	21.42	40	248	1,977	20.66
$0 - 1,000$	464	3,844	49,626	15.92	111	754	6,402	20.30

Table VII: The Number of Apps that Have Used the Cloud APIs in Each of The Accumulated Download Category.

	App Name	App Description and Functionality	Obfuscated?	Data in Database/Storage	Privacy Sensitive?
AWS	A1	Sending messages with multiple fancy features	✓	User Photos	✓
	A2	Editing user photos with magical enhancements	✓	User Photos	✓
	A3	Editing user photos with featured specialties	✓	User Photos; Posted Pictures	✓
	A4	Allowing users to organize and upload photos	✗	User Uploaded Pictures	✓
	A5	Helping users in planning and booking trips	✓	User Photos	✓
	A6	A game app to build and design attractive hotels	✗	User Backups	✓
	A7	A game app to express revenges on game NPCs	✗	Premium Plug-ins	✗
	A8	Pushing news and allowing users to report news	✗	User Uploaded Pictures & Videos	✓
	Drupe	Helping user to manage and reach their contacts	✓	User Voice Messages	✓
	A9	Pushing news and allowing users to report news	✗	User Uploaded Pictures & Videos	✓
Azure	A10	Helping users to start a diet and control weight	✓	User Photos; Posted Pictures	✓
	A11	Calculating and tracking calories for human health	✗	User Photos	✓
	A12	Showing fertility status from correspondent kits	✗	User Uploaded Pictures	✓
	A13	Helping users to easily play a popular game	✗	Configurations about the Game	✗
	A14	A real time translation tool, for calls, chats, etc.	✗	User Photos; Chat History	✓
	A15	Showing images of nations' commemorative coins	✓	Coins Images	✗
	A16	A convenient tool to take notes with rich content	✓	User Uploaded Pictures	✓
	A17	A convenient tool for users to schedule a taxi	✗	Driver Photos	✓
	A18	Allowing users to buy/renew general insurances	✗	Inspection Videos	✓
	A19	Providing accurate local weather forecast	✓	Device Info (IMEI, etc.)	✓
Firebase	A20	Editing and enhancing users photos and selfies	✗	User Info (①④); User Private Messages	✓
	A21	Allowing users to guess information about music	✓	Music Details	✗
	A22	Allowing users to sell and buy multiple products	✗	User Info (②④); Transactions	✓
	Photo Collage	Creating photo collage with personal photos	✓	User Info (②③)	✓
	A23	Helping users to translate and learn languages	✓	User Info (①); Quiz Data	✓
	A24	Editing user photos with effects for cartoon avatar	✗	User Info (①); User Pictures	✓
	A25	Help users to learn how to draw human bodies	✓	User Info (①②③); User Pictures	✓
	A26	An offline bible learning app with texts and audios	✗	User Info (①③④)	✓
	A27	Music platform for hiphop mixtapes and musics	✗	User Info (①②③); Play List	✓
	A28	Helping users to learn drawing different things	✓	User Info (①②③); User Pictures	✓

Table VIII: The Detailed Study of the Top-10 Vulnerable Apps from Each Cloud Category. Note that symbol ① denotes the user name, ② the user ID, ③ the user email, and ④ the user token.

likely it is obfuscated. We believe this is because developers of these apps are more likely to have a better security mindset.

However, even though the apps might have been obfuscated, we can still detect their data leakage vulnerabilities. (In fact, many of the top apps detected as vulnerable are obfuscated, as shown in Table VII). This is because our key techniques are obfuscation-resilient and regardless whether an app is obfuscated or not, we are still able to resolve the vast majority of the strings of our interest, as shown in the 7th and 11th column of Table V. Therefore, obfuscation does not help developers defeat data leakage attacks; they must implement proper authentication and authorization in order to prevent these attacks.

False Positive Analysis. LeakScope first uses static analysis to identify strings of interest (e.g., various keys used by the app), and then uses dynamic analysis to confirm the data leaks by inspecting the responses to our leakage-probing requests. There are no false positives in determining whether the data stored in the cloud can be leaked. That is, for all the vulnerable apps we detected, their servers are subject to data leakage attacks. However, there might be cases in which developers may deliberately leave their data open. To really decide whether LeakScope has any false positives in this regard, we must look at the data itself. If the corresponding leaked data is not privacy sensitive, then it is a false positive.

To this end, we manually registered a user account from the app with the corresponding cloud server, and we reverse engineered both the app code and the network traffic to understand whether the data stored in the cloud is privacy sensitive or not. We could not confirm this with all of the apps data due to our limited man power and also the grand challenge of reverse engineering the obfuscated apps, so instead we only focused on the top 10 most popular vulnerable apps in which we have the best understanding from each of the tested clouds.

The detailed report for each of these apps and the data that can be leaked is reported in Table VIII. Note that we would like to keep the app name anonymized since not all of them have been patched yet, and therefore we only report the name of the app (the 2nd column) shown in the Google Play if its vulnerabilities has been patched (as of May 2018, there are two apps whose servers have been patched.), followed by the app description and functionality (the 3rd column), whether this app has been obfuscated (the 4th column), the specific data that can be leaked from the cloud server for this particular app (the 5th column), and finally whether these data are privacy sensitive (the last column).

For the top 10 vulnerable apps that have used AWS, we notice that many of them store user photos: either user avatars or the photos taken by the users. There are also some other files such as videos and configurations. Interestingly, there are two news apps that use AWS for storage (but the news content is not stored in AWS). In particular, they allow users to report news such as a witnessed accident and in the meantime allow users to attach pictures or videos about the reported news. Clearly, an attacker could easily grab these files. We also have to stress that an attacker can tamper with the integrity of the files stored in AWS as well. We have similar observations for the vulnerable apps that use Azure. Most of the data are privacy sensitive, such as user photos, chat history, and videos. For instance, the 9th app, a car insurance related app, allows the users to take and upload a video for car inspection. We believe these files clearly should be protected.

Unlike AWS and Azure, which are mainly used by apps for storage, Firebase is a database that contains a variety of data records. As reported in the 5th column of Table VIII, we summarize those data records according to their category such as user name, user ID, and user email. While most of the data are privacy sensitive, we notice there is one app that only stores music related data in the database. In particular, the 2nd app, a music related app, allows users to guess the information of songs. All the data in its database are related to the music, such as music ID, music download URL, and the singer. While it does not contain any privacy sensitive data, anyone could obtain the entire database with a single HTTPS request. We believe this is not what the developers have intended (*e.g.*, a competitor could easily build a similar system with these data).

In summary, what LeakScope can automatically discover is the cases in which the data stored in the cloud back-end can be leaked. To really determine whether LeakScope has any false positives, both end-users and service providers would need to classify whether the data is of importance to them and is privacy sensitive or not. Currently, we do not have an automatic technique, though our manual classification with 30 vulnerable apps has shown that 86.7% of these apps' data is indeed privacy sensitive.

VII. Discussion

Our study has uncovered tens of thousands of mobile apps that contain cloud data leakage vulnerabilities. Altogether, these apps have accumulated downloads of between 4 billion and 14 billion. As such, it is a very serious security problem. In this section, we discuss further why such vulnerabilities exist and the countermeasures (§VII-A), the limitations and future work (§VII-B), and finally how we handled ethics during our study (§VII-C).

A. Root Causes and Countermeasures

There are many reasons for the data leaks in the cloud. The first one is "security through obscurity". Developers may believe that no one could find their (root) keys. But unfortunately, with simple reverse engineering of the (obfuscated) mobile apps, an adversary can easily extract various keys and directly use them to communicate with the server. The second reason is the lack of security training when using the SDK. For instance, it is an absolute security disaster to use a root key to communicate with the server. It is also a huge mistake to not validate the user's identity when accessing particular resources.

In response, providing security training to developers is an immediate step to alleviate this problem. Cloud providers should clearly document various mistakes that developers could make and their consequences in their manuals, and most importantly provide the correct way (not the wrong way) to use the keys. In fact, very surprisingly, we discovered that an official example from Azure documentation had actually misused the root keys (instead of using the SAS keys) to communicate with the cloud back-end from the mobile apps. This also explains why there are so many key misuses in Azure.

More importantly, cloud providers should also offer better security tools and SDKs to help developers. For instance, the SDK should perform type checks, and the cloud back-end should also reject the incorrect use of the keys. The SDK should also make the security policy specification easier, especially for Google Firebase, where more templates or GUI interfaces could have been provided to make it easier for developers to follow. Finally, cloud providers could also develop security tools to detect data leaks (e.g., by checking for insecure access control policies in the cloud back-end periodically).

B. Limitations and Future Work

While LeakScope has detected many data leakage vulnerabilities in the cloud back-end from mobile apps, clearly it is not perfect and has many limitations. First, it has false negatives. This is because the detection of the vulnerability is based on the APIs listed in Table III. If there are any other APIs that also involve app or developer credentials in their parameters, LeakScope will have missed the identification of these strings. For one of our future efforts, we would like to focus on on more systematically examining all of the APIs from cloud provider SDKs.

Second, our String Value Analysis does not recognize dynamically generated values, e.g., those received from remote servers, since we use static analysis without actually executing the apps. For instance, there are 404 apps from which LeakScope has failed to extract strings of our interest. Note that this also contributes to the false negatives. To handle these apps, we plan to use dynamic analysis to run the apps, hook the APIs of our interest, and extract the corresponding parameters. This is another future work of ours.

Third, due to ethics considerations (§VII-C), we only validated the data leakage vulnerabilities with our best efforts. For instance, we only identified 9,023 no permission check vulnerabilities in Firebase. In fact, this is because we could only automatically register users with only 13,506 out of the 101,380 apps that used Firebase in our dataset. We should be able to identify more vulnerabilities if there are any other approaches to bypass the authentication for the remaining 87,874 apps, or if there is collaboration from the cloud

providers. Increasing the coverage of our analysis is the 3rd avenue for our future work.

Finally, LeakScope only focuses on the apps that use cloud APIs to develop the mobile apps. Clearly, there is a significant portion of the apps that have directly used other types of cloud services (*e.g.*, the IaaS cloud) in their back-end. How to identify these apps and their vulnerable cloud services in a principled manner, as LeakScope has achieved for mBaaS, is another avenue for our future work.

C. Ethics

Since our work aims to identify data leakage vulnerabilities in the cloud, we had to ensure that our research would not directly leak any of the customer data. As described in §V-C, we took the ethics into consideration and designed a zero-data-leakage vulnerability identification approach by considering how the server would respond to a client request based on different user roles. Though this approach has limited the number of vulnerabilities we could identify, it is secure with respect to the customer data.

Moreover, we have made responsible disclosure to each of the cloud providers, and through them can reach the mobile app developers. All of the cloud providers are actively working on addressing the issues we reported. For instance, we have learned from Google that they have immediately warned the vulnerable Firebase users and are monitoring the vulnerability patching process, especially for the super popular apps (w/ between 100 and 500 million users).

Furthermore, over the past a few months, we have also been engaging with the cloud providers on how to detect, mitigate, and prevent these data leakage vulnerabilities. More importantly, as part of the consequences of our research, Google has planned to provide more developer-friendly SDKs when configuring the user permissions for authorization. Azure has corrected its documentation on how to use the right keys to communicate with the cloud in its recent git commits [8].

VIII. Related Work

Vulnerability Identification with Mobile Systems. Developing mobile apps is similar to developing traditional software in which developers could have made mistakes, thereby leading to various security vulnerabilities. Over the past many years, significant efforts have focused on identifying various vulnerabilities from mobile apps. Early efforts focused on identifying privacy leakage, since a user's GPS coordinates, address book, etc., can be accidentally leaked. TaintDroid [28], PiOS [27], and AndroidLeaks [29] are examples of these efforts. They leveraged either dynamic analysis to track whether sensitive information (*e.g.*, the address book) can be leaked, or static analysis to identify leakage.

In addition to privacy leaks from mobile apps, there are also other security vulnerabilities. For instance, component hijacking vulnerabilities [32] allow an attacker to hijack the flow and perform unauthorized read and write operations, code injection vulnerabilities [30] enable an attacker to inject malicious Javascript code into mobile apps, and hanging attribute references vulnerabilities [21] allow a malicious app to acquire critical system capabilities. Correspondingly, a number of tools such as CHEX [32] and Harehunter [21] have been developed to identify them.

Most recently, there were also a number of efforts to identify server side vulnerabilities of mobile apps. For instance, there are password brute-forcing attacks [47] if a server fails to track the number of user login attempts, shopping for free if a merchant server does not validate the payment information [40], SQL injection and server API misuse vulnerabilities if the servers do not check the requests from the apps [46], [49], and the use of insecure user tokens (*e.g.*, no randomness) in server authorization [48]. There are also corresponding tools such as AutoForge [47] and AuthScope [48] to identify them.

Misconfiguration Vulnerability Detection. Complex software systems such as the mBaaS cloud are difficult to configure and manage, and consequently various configuration errors can be introduced. Incorrect access control configuration, such as the permission misconfiguration that LeakScope aims to discover will clearly lead to security vulnerabilities. Unlike the key misuse vulnerabilities, which are caused by mistakes from the app developers, permission misconfigurations are caused by system administrators.

To detect permission misconfiguration in access control systems (*e.g.*, firewalls), FIREMAN [41] uses symbolic model checking of the firewall configurations to infer policy violations and inconsistencies. In typical application systems (*e.g.*, healthcare), Bauer et al. [25] have applied association rule mining to the access control logs to infer the intended policies and the misconfigurations. In an enterprise network, Baaz [26] infers the permission misconfiguration by monitoring updates to the access control metadata and looking at inconsistency among peers.

There are also numerous efforts to detect misconfigurations in software systems with configuration testing. ConfErr [31] is a blackbox configuration testing tool, which exposes configuration errors by injecting spelling errors, structural errors, and semantic errors. ConfAid [22] is a white box configuration diagnosis tool, which explores the control and data flows related to the erroneous behavior to specific tokens in configuration files. SPEX [39] is also a white-box configuration testing tool, which generates configuration errors based on how the configuration parameter is read and used.

Being a blackbox testing tool, LeakScope only explores the differences in the server response messages to infer whether a cloud server has configured the user permissions correctly. We believe cloud providers can certainly go beyond blackbox testing, and instead they can develop whitebox approaches to proactively detect permission misconfigurations.

IX. Conclusion

We have studied the problem of why there have been so many recent private data leaks from the cloud, and we discovered that the misuse of various keys in mobile app authentication and misconfiguration of user permissions in authorization are the two root causes that can lead to the massive data leaks in the cloud. We have designed and implemented LeakScope to automatically identify the cloud services that can contain data leakage vulnerabilities from mobile apps. Our evaluation with over 1.6 million mobile apps from the Google Play Store has uncovered tens of thousands of vulnerable cloud services including those from Google, Amazon, and Microsoft. We have made responsible disclosure to each of the vulnerable service providers, and they have all confirmed the vulnerabilities we identified and are actively working with the mobile app developers to patch their vulnerable services.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their invaluable feedbacks. We also would like to thank Erick Bauman and Atanas Rountev for their helpful comments on an early draft of the paper. This work was supported in part by AFOSR under grant FA9550-14-1-0119, and NSF awards 1718084, 1834213, and 1834215. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the AFOSR and NSF.

REFERENCES

[1] "Authenticate with firebase using password-based accounts on android," https://firebase.google.com/docs/auth/android/password-auth.

[2] "azure-storage-android," http://azure.github.io/azure-storage-android/.

[3] "Best practices for securely using api keys," https://support.google.com/cloud/answer/6310037.

[4] "Create a platform endpoint and manage device tokens," http://docs.aws.amazon.com/sns/latest/dg/mobile-platform-endpoint.html.

[5] "Dexguard android obfuscator," https://www.guardsquare.com/dexguard.

[6] "dexlib2," https://github.com/JesusFreke/smali/tree/master/dexlib2.

[7] "Dexprotector android obfuscator," https://dexprotector.com.

[8] "Disclaimer on the use of account key," https://github.com/Azure/azure-storage-android/commit/d90c3a49312e77c2cc911c8f55a37be9947454e4.

[9] "Head bucket," http://docs.aws.amazon.com/AmazonS3/latest/API/RESTBucketHEAD.html.

[10] "Manage users in firebase," https://firebase.google.com/docs/auth/android/manage-users.

[11] "Mobile backend as a service," https://en.wikipedia.org/wiki/Mobile_backend_as_a_service.

[12] "Proguard java obfuscator," https://http://proguard.sourceforge.net.

[13] "Read and write data on android," https://firebase.google.com/docs/database/android/read-and-write#updating_or_deleting_data.

[14] "Scrapy | a fast and powerful scraping and web crawling framework," https://scrapy.org/.

[15] "Shrink your code and resources," https://developer.android.com/studio/build/shrink-code.html.

[16] "Soot - a framework for analyzing and transforming java and android applications," http://sable.github.io/soot/.

[17] "Upload files on android," https://firebase.google.com/docs/storage/android/upload-files?authuser=0.

[18] "Using the aws sdk for java with amazon sns," http://docs.aws.amazon.com/sns/latest/dg/using-awssdkjava.html.

[19] "What is an interface?" https://docs.oracle.com/javase/tutorial/java/concepts/interface.html.

[20] "The statistics portal: Mobile app usage," https://www.statista.com/topics/1002/mobile-app-usage/, December 2017.

[21] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace, "Hare hunting in the wild android: A study on the threat of hanging attribute references," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1248–1259.

[22] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, Vancouver, BC, Canada, 2010, pp. 237–250.

[23] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 356–367.

[24] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Compiler Construction*. Springer, 2004, pp. 2732–2733.

[25] L. Bauer, S. Garriss, and M. K. Reiter, "Detecting and resolving policy misconfigurations in access-control systems," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 1, pp. 2:1–2:28, Jun. 2011.

[26] T. Das, R. Bhagwan, and P. Naldurg, "Baaz: A system for detecting access control misconfigurations," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10, Washington, DC, 2010.

[27] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *NDSS*, 2011.

[28] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI*, 2010.

[29] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *Trust*, 2012.

[30] 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*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 66–77.

[31] L. Keller, P. Upadhyaya, and G. Candea, "Conferr: A tool for assessing resilience to human configuration errors," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 157–166.

[32] 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.

[33] P. Muncaster, "Verizon Hit by Another Amazon S3 Leak," https://www.infosecurity-magazine.com/news/verizon-hit-by-another-amazon-s3/, September 2017.

[34] M. OLSON, "Cloud computing trends to watch in 2017," https://apiumhub.com/tech-blog-barcelona/cloud-computing/, April 2017.

[35] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, "Enhancing server availability and security through failure-oblivious computing." in *OSDI*, vol. 4, 2004, pp. 21–21.

[36] T. Spring, "Insecure backend databases blamed for leaking 43tb of app data," https://threatpost.com/insecure-backend-databases-blamed-for-leaking-43tb-of-app-data/126021/, June 2017.

[37] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[38] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic diagnosis and response to memory corruption vulnerabilities," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 223–234.

[39] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania, 2013, pp. 244–259.

[40] W. Yang, Y. Zhang, J. Li, H. Liu, Q. Wang, Y. Zhang, and D. Gu, "Show me the money! finding flawed implementations of third-party in-app payment in android apps," in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2017.

[41] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra, "Fireman: A toolkit for firewall modeling and analysis," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, ser. SP'06, 2006, pp. 199–213.

[42] S. Yue, W. Feng, J. Ma, Y. Jiang, X. Tao, C. Xu, and J. Lu, "Repdroid: An automated tool for android application repackaging detection," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 132–142.

[43] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 2014, pp. 25–36.

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

[45] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *ACM*

Cloud Service	API	Definition	Indexes of The String Parameters of Our Interest
AWS	**1***	**TransferUtility: TransferObserver downloadUpload(String, String, File)**	0
	1.1	TransferUtility: TransferObserver download(String, String, File)	0
	1.2	TransferUtility: TransferObserver download(String, String, File, TransferListener)	0
	1.3	TransferUtility: TransferObserver upload(String, String, File)	0
	1.4	TransferUtility: TransferObserver upload(String, String, File, ObjectMetadata)	0
	1.5	TransferUtility: TransferObserver upload(String, String, File, CannedAccessControlList)	0
	1.6	TransferUtility: TransferObserver upload(String, String, ObjectMetadata, CannedAccessControlList)	0
	1.7	TransferUtility: TransferObserver upload(String, String, ObjectMetadata, CannedAccessControlList, TransferListener)	0
	2*	**AmazonS3Client: void S3objectAccess(String, String, ...)**	0
	2.1	AmazonS3Client: void deleteObject(String, String)	0
	2.2	AmazonS3Client: void deleteVersion(String, String, String)	0
	2.3	AmazonS3Client: boolean doesObjectExist(String, String)	0
	2.4	AmazonS3Client: String getBucketLocation(String)	0
	2.5	AmazonS3Client: S3Object getObject(String, String)	0
	2.6	AmazonS3Client: String getObjectAsString(String, String)	0
	2.7	AmazonS3Client: ObjectMetadata getObjectMetadata(String, String)	0
	2.8	AmazonS3Client: String getResourceUrl(String, String)	0
	2.9	AmazonS3Client: URL getUrl(String, String)	0
	2.10	AmazonS3Client: ObjectListing listObjects(String)	0
	2.11	AmazonS3Client: ObjectListing listObjects(String, String)	0
	2.12	AmazonS3Client: ListObjectsV2Result listObjectsV2(String)	0
	2.13	AmazonS3Client: ListObjectsV2Result listObjectsV2(String, String)	0
	2.14	AmazonS3Client: PutObjectResult putObject(String, String, File)	0
	2.15	AmazonS3Client: PutObjectResult putObject(String, String, InputStream, ObjectMetadata)	0
	2.16	AmazonS3Client: PutObjectResult putObject(String, String, String)	0
	2.17	AmazonS3Client: void restoreObject(String, String, int)	0

Table IX: Specific Targeted mBaaS Cloud APIs of Amazon AWS

conference on Data and Application Security and Privacy. ACM, 2012, pp. 317–326.

[46] C. Zuo and Z. Lin, "Exposing server urls of mobile apps with selective symbolic execution," in *Proceedings of the 26th World Wide Web Conference*, Perth, Australia, April 2017.

[47] C. Zuo, W. Wang, R. Wang, and Z. Lin, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, February 2016.

[48] C. Zuo, Q. Zhao, and Z. Lin, "Authscope: Towards automatic discovery of vulnerable authorizations in online services," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, Dallas, TX, November 2017.

[49] J. Chen, X. Cui, Z. Zhao, J. Liang, and S. Guo, "Toward discovering and exploiting private server-side web apis," in *Web Services (ICWS), 2016 IEEE International Conference*, 2016.

APPENDIX

In Table III, we could not report the concrete API definitions of two sets of APIs, and instead we just used 1* and 2* to denote them due to the space limit. The concrete definition of these two sets of APIs are described in Table IX. We can see that there are 7 APIs in 1* and 17 APIs in 2*. We are interested in all of the first parameters, namely the `bucketName` as reported in Table V.