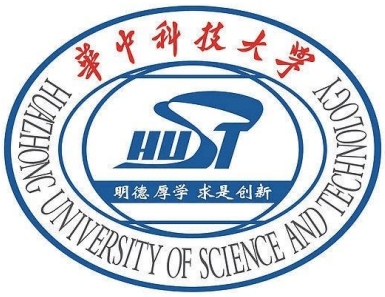
華中科技大學



2022 年度云计算与虚拟化论文阅读报告

Why does your data leak? Uncovering the data leakage in cloud from mobile apps

姓　　名：李欣宇

学　　号：U201911658

班　　级：信安 1901 班

任课教师：李志

论文编号：41

2022 年 12 月 24 日

目录

1 论文拟解决的问题

随着云计算的快速发展，移动应用程序也有了巨大增长，从终端用户角度看，移动应用程序可以被认为是互联网服务的前端，而云是后端。几乎所有我们日常使用的互联网服务都有自己的专用移动应用程序，但是随着云计算作为移动应用的后端，也发生了大量从个人医疗到企业机密的各行各业的云计算的数据泄露，泄漏的数据包含了大量用户信息，包括用户名、密码、电子邮件、电话号码、位置[1]等等，即使是一些知名的公司也不能避免。

本文试图进行系统研究云服务中数据泄露的根本原因并尝试实现一套自动化工具系统识别云服务中的漏洞。通过研究发现，越来越多的应用使用云服务商提供的 APIs(Cloud APIs) 实现数据存储、数据分析、消息推送等功能，在云端和移动应用程序通信时在用户认证的管理和授权过程中对用户权限的错误配置是导致云端数据泄露的根本原因。为解决该问题本文作者从 App 中错误地使用云端 APIs(Cloud APIs) 这一角度出发，设计并实现了一个可扩展且高效的自动化测试工具 LeakScope，其中的程序分析技术包括抗混淆的云 API 识别、字符串分析和零数据泄露漏洞验证实现自动识别移动应用中可能存在的云数据泄露漏洞。

2 论文提出的核心方法

2.1 背景介绍

2.1.1 Mobile Cloud API

在云计算 API 开始流行之前，开发者开发一个移动应用程序必须从头开始建立整个基础设施，包括前端应用程序和后端服务器。此外，即使在发布应用程序后，整个系统也需要很大代价来确保其稳定性和安全性。此外，如果用户逐渐增多为了保证高可用性必须扩大扩大系统以适应潜在的大规模用户增长。在这种移动应用开发者的需求中，主流的云供应商开始提供移动后端作为一种服务（mBaaS），其中包括相应的平台和 API，用于快速移动应用开发。

目前有许多 mBaaS 云供应商，但本文只研究谷歌、亚马逊和微软提供的 mBaaS，每个云供应商提供四个典型的组件，包括数据库管理、用户管理、存储和推送通知以及相应的 API。也正因此，使用 Cloud API 无需搭建后端或租用服务器，也无需考虑鲁棒性和伸缩性，能够实现快速移动应用开发，同时便于维护。

2.1.2 Cloud API 中的 key 机制

App Key（应用密钥）：只具有有限访问权限，云可以根据密钥知道 API 的调用是来自哪个应用，只允许该应用访问公开可用的资源，同时可以隔离资源。

Root Key（根密钥）：具有完整的访问权限，所有属于它的后端资源都可以被访问，便于开发和管理，通常只有系统管理员拥有。

图 1 说明了云供应商的 App Key 和 Root Key 的典型用法以及基于云的移动应用程序的架构。在高层次上，开发者需要首先向云后端注册以获得应用密钥和根密钥，设置后端数据库和存储，并在后

端实现适当的认证和授权，然后，移动应用程序可以通过传递相应的应用程序密钥和其他参数来调用相应的云 API，以访问所需要的云资源。

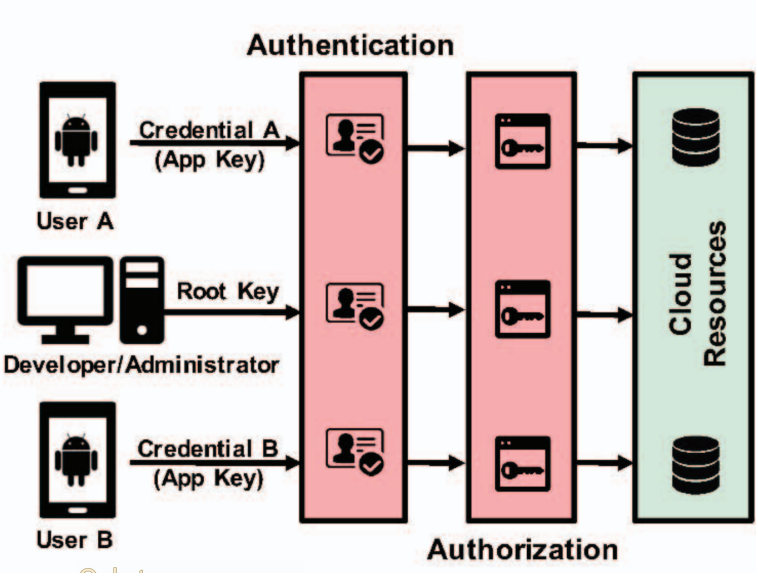


图 1: 使用云计算 API 的移动应用程序典型架构

2.2 误用云 API 的危害

前文提到导致大规模云端数据泄露的根本原因是（1）在认证过程中滥用了两种密钥（2）在授权中错误配置了用户权限，下文将对本文要研究的三个 API 中出现的这两种漏洞进行分析。

2.2.1 认证中滥用密钥

1.Azure 在存储中的密钥误用

Microsoft Azure 云为开发者提供了两种密钥来访问其存储。

- **帐户密钥（Account Key)**

Azure 存储账户提供了一个独特的命名空间，用于存储和访问 Azure 存储中的各种数据，如表、队列、文件和对特定用户收费的虚拟磁盘。这个账户密钥的作用就像一个根密钥，它可以完全访问云存储。

- **共享访问签名（SAS）**

SAS 为属于特定存诸账户的资源提供申请访问。与账户密钥不同，开发人员可以配置 SAS 的权限（例如，读、写），以便 SAS 只能以有限的权限访问某些资源。

在实际开发中，开发者应该在应用程序中只使用 SAS，并对账户密钥进行加密。例如，当一个用户尝试从移动应用中访问一个私人文件时，开发者应该分配一个 SAS，该 SAS 对该用户来说具有访问特定文件的最低权限，但是，通过实验发现，许多开发者直接在移动应用程序中使用账户密钥，这导致账户密钥很容易被攻击者通过逆向工程提取。

2.Azure 在通知枢纽中的密钥误用

通过云端 API，开发者将消息推送给特定的用户或广播给所有用户。Microsoft Azure 提供了一个通知枢纽，如图 2 所示，要使用 Azure 通知枢纽，开发者首先需要注册一个频道，然后移动应用只需

调用 API，使用监听密钥监听开发者注册频道，之后，应用服务器可以向通知枢纽中注册的管道推送通知（使用完整的访问密钥），然后通知枢纽把消息转发给所有应用程序的用户，这里涉及两个密钥。

- **完全访问密钥（Full Access Key）**

  完全访问密钥的工作方式类似于根密钥，它有完整的权限来发送或监听通道上的通知。

- **监听密钥（Listening Key）**

  监听密钥具有有限的权限，只能监听在特定通道中的通知。

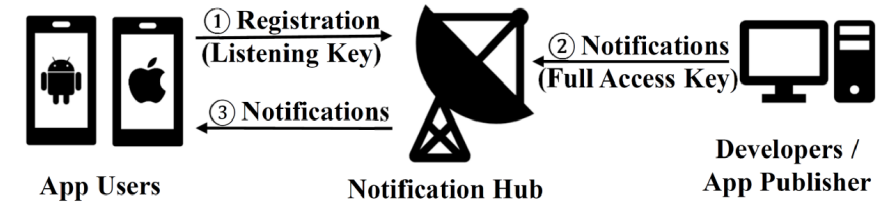


图 2: 使用 Azure 通知枢纽的过程图

显然，监听密钥应该只由移动应用程序使用，而完全访问密钥应该只由应用程序服务器使用。但是在实验过程中发现一些开发者在移动应用中直接使用完全访问密钥，虽然 Azure 通知枢纽中的密钥滥用可能不会直接导致数据泄露攻击，但是一旦攻击者提取了完全访问密钥，就有了极大的权限，间接地可以推送钓鱼信息，窃取其他应用程序用户的数据。

3.AWS 中的密钥误用

亚马逊 AWS 也为移动应用程序提供了一些密钥用来访问 AWS 的资源（例如，S3 存储)，其中一个密钥是根访问密钥，它可以访问特定 AWS 账户下的所有资源。因为这个根密钥拥有所有的权限，使用过程中应该进行加密存储，但是作者注意到这个根密钥被用在移动应用程序中。一旦攻击者提取了这个根密钥，它就可以访问整个存储资源。

2.2.2 授权时用户权限配置错误

通常情况下，认证只告诉系统用户是谁，而决定认证用户可以访问哪些具体资源的则是授权。不授权或不正确的配置会使授权层失去作用，从而导致数据泄露。本文作者发现 (1) 使用 Firebase 的应用程序在后端错误地配置了用户权限；（2）使用 AWS 的应用程序也存在这种错误配置问题。

1.Firebase 中用户权限的错误配置

当使用 Firebase 时，开发者必须定义特定用户的访问控制政策，即“规则”。图 3 是一个“规则”的例子，Firebase 是一个实时数据库，它用 JSON 将数据组织成一个层次结构。根据这个例子的规则，在数据库中有一个名为“users”的节点——当一个用户试图访问 (读/写) 它的一个子节点，该子节点有一个标志参数 $uid，系统只允许这个用户的 $uid 等于子节点地 $uid 时可以进行特定的读写操作，但是，并不是所有的开发者都遵循这种正确的方式来写规则。例如，如图 4 所示，开发者可以编写规则来直接忽略检查，或者只检查用户是否经过验证，这样写意味着任何 (经过认证的) 用户都可以访问整个数据库。

```
{
  "rules": {
    "users": {
      "$uid": {
        ".read": "$uid === auth.uid",
        ".write": "$uid === auth.uid"
      }
    }
  }
}
```

图 3: 正确的 Firebase“规则”授权

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

(a)

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

(b)

图 4: 两个配置错误的 Firebase“规则”

2.AWS 中用户权限的错误配置

通过 AWS，亚马逊提供了身份和访问管理 (IAM)，用于用户管理和权限配置。为了安全地访问特定移动应用程序的资源，开发者需要创建一个 IAM 用户，开发者可以为 IAM 用户配置权限，每个 IAM 用户可以生成两个秘密访问密钥来访问指定资源。虽然使用 IAM 用户似乎是安全的，但事实上开发人员可能会过度配置 IAM 用户的权限，如授予特定存储的完全访问权，这也会导致数据泄露。

2.3 自动化工具——LeakScope

对于上述提到的两种云数据泄露作者设计并实现了一个自动化静态分析框架 LeakScope，对于给定的移动应用，可以自动检测数据泄露的漏洞。

这个框架是基于 dexlib2[2]和 soot[3]实现的，在云 API 识别上使用 dexlib2 能够高效解析 dex 文件并建立函数签名，而字符串值分析是基于 soot 的，用于分析 java 字节码，对于漏洞识别作者写了一个 python 脚本进行请求响应测试。

图 5 是 LeakScope 的框架图，由三个关键部分组成：云 API 识别，字符串值分析以及漏洞识别，下面将分这三个部分进行陈述 LeakScope 的设计与实现。

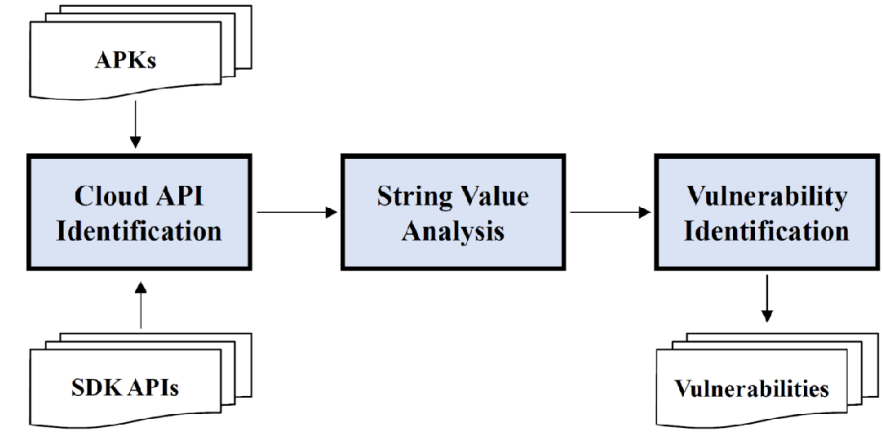


图 5: LeakScope 架构

2.3.1 云 API 识别

首先要识别应用程序使用的 API，但是在实际使用中有些开发者会混淆 API，混淆 API 往往有两种策略

- 将 API 中涉及的名称，如子包名、名、函数名、变量名重命名为一些无意义的字符，这一般使用混淆加密工具实现，例如 Dexprotector , Dexguard 等
- 删除没有使用的函数/API。因为知道一些没有使用过的函数、API 往往可以知道应用程序使用的包和类，通过使用自动化工具，如 Proguard 可以从应用程序中删除这些函数，进一步隐藏 API。

基于此，在云 API 识别方面，作者设计了一种抗混淆的方法识别 API，作者认为对于以上两种混淆策略，为了达到抗混淆的目的，至少要注意到两个不变因素，无论给定的函数是否被混淆，都可以被识别出来：

- **类型和包树的层次结构**

  每个类，函数、参数和变量都有类型，虽然类的名称可以被模糊化，但层次结构不会被改变，如图 6 所示，Callback 始终是被存储在包 com 的第四层。

- **调用与被调用的关系**

  可以通过调用关系，递归地建立每个调用者地签名，然后合并它们建立调用者的签名。因此，可以对这两个不变因素进行编码，并加上它们一起构建一个函数的签名，然后可以在移动应用的字节码中搜索这些签名，找到 API。

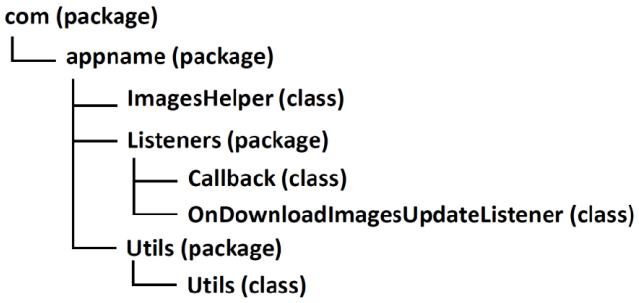


图 6: 图 9 代码的包树层次结构

作者受 LibScout[4]的启发，使用 Merkle 哈希树用来建立一个库的签名，作者改进了 LibScout 的方案，为每个函数生成签名，使用 MD5(fBuf) 哈希值作为作为 API 检测的函数/API 签名，其中 fBuf 是类型和包树层次结构以及调用-被调用关系的编码，如图 7 算法 1，这里使用了（i）函数的类（第 5-6 行），（ii）参数（第 7-8 行），（iii）局部变量（第 9-10 行），（iv）返回值（第 11-12 行），（v）所有被调用者（第 13-18 行）的编码类型字符串 MD5 生成了一个签名（GENFUNSIG）

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)
```

图 7: 生成 fBUf 的算法

图 8 则展示了对类的字符串编码算法。

```
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
```

图 8: 生成 tBuf 的算法

通过上述的 GENFUNSIG 算法，可以为所有的 method 生成签名，包括给定应用程序的云 API，然后可以在移动应用程序中搜索云 API 的签名，如果签名匹配，就能够识别出相应的 API。

2.3.2 字符串值分析

在找到云 API 后就可以确定每个 API 的调用位置，然后需要进一步确定移动应用中包含认证密钥的参数，因为使用的静态分析，不能够进行动态跟踪，所以必须开发一个有针对性的字符串值分析。

这里采用了后向切片＋前向执行的方式。

```
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 ...
```

图 9: 从某 Android 应用程序中获取 Azure 存储的反编译代码

- **后向切片**

首先为每个 method/function 建立一个内程序控制流图（CFG），其中节点代表连续执行的字节码指令，边代表函数内的控制流传输，然后从某个变量开始，例如 CloudStorageAccount . parse 的 v0，在 CFG 中逆向迭代指令：如果有变量有助于 v0 的计算，就把它们添加到的数据依赖图 (DDG) 中，同时把涉及的指令和变量推到字符串计算栈中，这是一个由 LeakScope 维护的内部后进先出数据结构，用来追踪函数的执行顺序；如果有函数调用，会进行上下文敏感的程序间分析并递归分析调用。不断地迭代 CFG，直到达到一个平衡点，即 DDG 不能进一步扩展。在图 9 的运行示例中，所有用红色标出的语句都参与了第 31 行使用的字符串值 v0 的计算。

- **前向执行**

有了跟踪的 DDG 和字符串计算栈，接下来需要计算最终字符串的值。从字符串计算堆栈的顶部开始，根据 CFG 和 DDG 弹出相关的变量和指令，并根据指令语义向前执行相关的字符串操作，直到堆栈为空或字符串值完全确定。前向执行不是真正的执行，而是基于字符串操作的 API 总结。例如，如果涉及的指令是字符串追加 API，就执行字符串追加操作; 如果是 getString，就知道它是用来从 xml 文件中读取字符串的，然后就从 xml 文件中执行指定字符串的读取操作并返回其结果。

回到上图 9 的运行示例，堆栈上最后推送的变量是 storageAccountKey 和 storageAccount-Name。然后，从跟踪的 DDG 开始，在第 9 行和第 12 行通过执行 getResources 和 getString (第 37-38 行) 的 API 摘要，进行前向字符串分析，计算出 this.storageAccountName 和 this.storageAccountKey 的值，然后计算 v0(第 17-21 行)、v1、this.val$conStr 的值，最后在第 31 行计算 v0 的值。

2.3.3 脆弱性识别

在获得由 LeakScope 识别的密钥后，接下来要检测云服务中的数据泄漏漏洞。该检测是针对云的，作者设计了了以下算法来检测使用 Azure、AWS 和 Firebase 的服务的密钥滥用和权限错误配置。

（1）检测 Azure (存储和通知中心) 中的密钥误用

如果移动应用程序中有根密钥或完全访问密钥，那么攻击者可以很容易地使用这些密钥来泄露数据。因此，不需要探测后端来确认漏洞。只要在移动应用程序中识别出根密钥或完全访问密钥，就可以知道云服务有漏洞。

（2）检测 AWS 中的密钥误用

一个 AWS 根密钥可以完全访问相应的 AWS 账户，如果一个应用程序中识别出一个根密钥，该账户下的所有资源都可以被访问。作者发现，根密钥有权限通过以下 rest API 获得一些 AWS 信息 https://ec2.amazonaws.com/?Action=DescribeInstances InstanceId.1=X，其中 X 是目标实例的 ID。而一个应用程序密钥没有这个权限，因此，为了检测 AWS 中的密钥滥用，作者将 X 设置为一个不存在的 ID，当单独发送一个确定了密钥的请求时，会收到错误的提示，如果是 Root key，收到“InvalidInstanceID” Error，如果是 App key，收到“UnauthorizedOperation”Error。

（3）检测 Firebase 中的权限错误配置

如图 4 所示，有两种典型的权限错误配置规则: (a) 没有认证检查 (数据库完全对任何人开放) 和 (b) 没有权限检查 (只检查用户是否通过认证)。使用以下算法来检测它们。

- **检测“开放”的数据库**

  当一个开发者错误地将读取策略指定“.read”：“true”，那么任何人都可以读取数据库。Firebase 提供了一个 REST API 来访问指定“path”中的数据库数据。通过设置“path”，用户可以读取特定的数据。如果将“path”设置为“root”，那么就可以读取整个数据库。然而，我们不一定要读取整个数据库，因为 Firebase 支持用 indexon 字段进行查询，如果在尝试读取“root”路径时将这个字段设置为不存在的值，那么就不会有数据泄露，但作者仍可以确认数据泄露，因为当用户有 root 读取权限时，返回的错误信息是不同的。

- **检测无权限检查**

  当一个数据记录的读取策略是“.read”为“auth!=null”时，必须使用一个经过验证的用户，以便执行基于索引的泄漏测试。作者注意到 Firebase 除了开发者的用户注册方法之外还提供一些用于用户注册的云 API，例如，通过使用电子邮件/密码、电话号码、谷歌/FacebookSSO 等方式。因此可以直接调用这些 FirebaseAPIs 来在相应的服务中注册合法用户，如果被测试的应用程序使用了这些 APIs，之后对注册用户进行认证，然后进行基于索引的测试。

（4）检测 AWS 中的权限错误配置

AWS 密钥用于在特定的访问控制策略下访问指定的资源。在 AWS 中，有几种类型的资源。在这些资源中，作者只关注 S3 存储的权限错误配置的检测，主要是因为最近即几次大规模的数据泄露，（例如，[5]）都来自 S3。为了执行测试，不仅要收集 AWS 的密钥，而且还要收集 S3 存储的名称，因此，必

须对它们都进行字符串值分析，有了确定的 AWS 密钥和存储名称，直接调用 AWS API HEAD Bucket 来验证一个密钥是否有访问存储的权限，如果是，就会检测到数据泄露。

3 论文贡献总结

3.1 系统性的研究

本文对云端数据泄露进行了系统了解，并确定了近些年大规模云端数据泄露的根本原因——在用户认证时误用两种密钥或在用户授权时用户权限配置错误。

3.2 实现自动化分析框架

本文实现了一个静态分析工具 LeakScope，用于自动检测移动应用程序中的数据泄露漏洞，该工具组件主要包括云 API 识别（Cloud API Identification），字符串分析（String Analysis）以及零数据漏洞验证（Zero-data-leakage Vulnerability Verification）。

3.3 实验验证高效性

本文利用 LeakScope 工具对 160 万移动应用程序进行评估，发现了使用 Amazon、Google 和 Microsoft 云服务的 15098 个 App 都存在云服务数据泄露漏洞，并且已与相关 App 开发者联系，对问题进行修复。

4 论文实验设计总结与分析

4.1 实验设计总述

实验环境：

- Intel Xeon E5-2640 CPU
- 24 核 +96G 内存
- Ubuntu 16.04

App 收集：

- 耗时 2 个月
- 使用爬虫脚本在两个月收集到 Google Play 上的 1609983 个免费 App
- 占用空间 15.42TB

实验结果：

- 检测耗时 6894.89 小时
- 中间结果占用 2.56TB
- 检测出 15098 个 App 中的 17299 个漏洞

4.2 实验结果

1. 云 API 识别（Cloud API Identification）

对于收集到的 160 万个 APP 使用 Cloud API Identification 识别产生了 107081 个使用了 Amazon AWS 、Google Firebase、Microsoft Azure 三种 API 的移动应用，其中 20.29%（21724 个）是被混淆的程序，在实验中将实验结果分成两组：未混淆应用程序（85357 个）与有混淆应用程序，以此对照比较混淆的应用程序是否能够更好地保护数据，防止数据泄露。

同时，根据这 1007081 个应用程序使用以上三种 API 的情况，分成了 7 组，分别是只使用了一种 API 的（3 组）、使用了其中两种 API 的（3 组），使用了以上三种 API 的（1 组），具体实验结果如下图，在实验中可以看到，超过 90% 的移动应用程序都使用了 Google Firebase API 服务

	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

图 10: 云 API 检测结果

2. 字符串值分析（String Value Analysis）

在 107081 个应用程序中，静态计算了 631551 个参数字符串，具体数据如下表所示，对于 AWS 中的 bucketName 字符串用来定位相应的 S3 存储，以此来进行用户授权漏洞的验证。对于 identityPoolId 字符串用来检测权限配置错误漏洞，在 Azure 和 Firebase 中字符串也是类似的可以用来验证漏洞。

根据表中第 7 列和第 11 列可以看到无论是混淆的还是非混淆的应用程序，字符串值分析都并不能把所有的参数字符串分析解决，主要原因有两个，其一是很多参数值是联网检索的，而非静态直接检索的，这种情况在 Firebase 中尤其常见，需要进行动态分析；其二是一些应用程序会使用加密函数加密字符串，静态分析的工具也不能解决这个问题。

	String Parameter Name	APIs	Non-Obfuscated #API-Call	#APP	#Resolved Str.	%	Obfuscated #API-Call	#APP	#Resolved Str.	%
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

图 11: 字符串值分析结果

3. 漏洞识别（Vulnerability Identification）

根据前面获得的 key 和字符串，利用零数据泄露策略检测漏洞，共发现了 17299 个漏洞，具体数据如下表。

	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

图 12: 检测到漏洞的应用程序统计

- **密钥滥用漏洞**

主要存在于 Azure 和 AWS 中，在表中可以观察到 907 个非混淆的 Azure 应用程序中有 20.51%（186 个）滥用了密钥，对于被混淆的应用程序，有 13.89%（30 个）包含密钥滥用漏洞；对于 AWS，在 5988 个应用程序中检测到 477 个（7.97%）有根密钥滥用的漏洞。

- **权限配置错误漏洞**

主要存在于 AWS 和 Firebase 云中。

根据表可以看到，漏洞最多的（按百分比）是 AWS 的"S3 存储的权限配置错误"，非混淆应用程序 15.30%，混淆的应用程序 24.44%。从 Azure 中可以观察到，经过混淆处理的应用程序安全性更高（13.89%vs20.51%），但在 AWS 和 Firebase 中，被混淆的应用程序安全性更低，这可能因为错误配置中的错误是针对产品的，与用户的安全专业知识联系甚少。

4.3 实验结果分析

1. 漏洞危害等级分析

本文用有漏洞的应用程序的下载次数来描述其危害性：下载次数越多，漏洞就越严重。作者统计了每个下载类别 (例如，10 亿到 50 亿之间）中存在漏洞的应用程序的下载数量，具体结果如下表所示。在使用上文提到的三个 API 的非常受欢迎的应用（如果一个应用的总下载量超过 100 万，定义为非常受欢迎)，其中 569 个应用受到了数据泄露攻击。在这些应用中，有 10 个的下载量在 1 亿到 5 亿之间，14 个在 5000 万到 1 亿之间，80 个在 1000 万到 5000 万之间。可预见到，如果攻击者利用了这些漏洞，用户敏感数据泄漏量将是数以亿计的。

#Downloads	# Non-Vulnerable Apps Azure	AWS	Firebase	Obfuscated%	# Vulnerable Apps 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

图 13: 每个累计下载类别中使用过云计算 API 的应用程序的数量

2. 混淆与未混淆

从上表中可以看到，混淆通常被用于高下载量的应用程序，一个应用程序下载量越高，被混淆的可能性越高，作者认为这是因为这些应用程序的开发者更具有安全意识，但是遗憾的是，即使这些应用程序被混淆了，仍可以检测到它们有数据泄露漏洞，因此混淆并不能保证数据不会泄露，而要避免数据泄露还是要从根本原因上解决，——正确使用密钥和合适的用户授权。

3. 误报分析

LeakScope 首先使用静态分析来识别相关的字符串（密钥），然后动态分析，通过检查对我们泄露探测请求的相应来确认是否有数据泄露，也就是说，对于所有检测到易受攻击的应用程序，其服务器都存在数据泄露漏洞，但是也会出现开发者故意开放一些不敏感数据，而 LeakScope 探测到的是这样的漏洞时称为假阳性（false positive）作者选取了 10 个最流行且易受攻击的应用程序，手动注册用户账户并对应用程序的代码和流量进行逆向分析，尝试了解存储在云端数据是否具有隐私敏感性，具体实验结果如下表所示，其中 86.7% 都是存在泄露用户照片、上传图片等问题，具有隐私敏感性。

	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	✓

图 14: 对每个云类别的前 10 个易受攻击的应用程序的详细研究，①表示用户名，②表示用户 ID, ③表示用户电子邮件，④表示用户令牌。

5 论文阅读心得

文章针对 Android App 进行大批量分析，而文章中提出的云数据泄露的的根本原因中访问控制导致的漏洞不仅在移动应用数据泄露中有所体现，在物联网或工业互联网、车联网等也有所体现。

总而言之，这个 LeakScope 能够自动识别来自移动应用的数据泄露漏洞，作者对来自 Google Play 商店的 160 多万移动应用程序进行了评估，找到了数以万计的有漏洞的云服务，可以说这个静态分析工具是非常高效的，但是 LeakScope 在实验中虽然有优越的表现，但也因为这是静态分析工具，所以也有其局限性，在字符串值分析中不能识别生成的值，所以也导致了误报或漏报，下一步对这个工具进行改进时可以加入动态分析，对其进行优化。

通过阅读这篇文章我也了解到了云 API 的一些背景以及在应用程序中的调用过程，同时解除了零

数据分析和字符串值分析以及以及在云 API 识别中抗混淆使用层次结构和调用关系定生成编码再签名的思想方法，收获颇丰。

参考文献

[1] SPRING T. Insecure backend databases blamed for leaking 43tb of app data[Z]. 2017.

[2] GRUVER B. Dexlib2[J]. Github.[Online]. Available: https://github. com/JesusFreke/smali/tree/master/dexlib2, 2021.

[3] LAM P, BODDEN E, LHOTÁK O, et al. The Soot framework for Java program analysis: a retrospective [C]//Cetus Users and Compiler Infastructure Workshop (CETUS 2011): vol. 15: 35. 2011.

[4] BACKES M, BUGIEL S, DERR E. Reliable third-party library detection in android and its security applications[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016: 356-367.

[5] MUNCASTER P. Verizon hit by another Amazon S3 leak[J]. InfoSecurity Magazine, 2017.