为什么你的数据会泄漏？揭开移动应用的云端数据泄露的面纱

左朝顺
俄亥俄州立大学

林志强
俄亥俄州立大学

张银倩
俄亥俄州立大学

***摘要*--**
***越来越多*的移动应用（简称应用）将云作为后端，特别是云API，用于数据存储、数据分析、消息通知和监控。不幸的是，我们最近目睹了来自云端的大量数据泄露，从个人身份信息到企业机密。在本文中，我们试图了解为什么会发生这种重大的泄漏，并设计工具来自动识别它们。令我们惊讶的是，我们的研究显示，缺乏认证，在认证中滥用各种密钥（*例如*，正常用户密钥和超级用户密钥），或在授权中错误配置用户权限是根本原因。然后，我们设计了一套自动程序分析技术，包括抗混淆的云API识别和字符串值分析，并在一个名为LeakScope的工具中实现，以根据云API的使用方式来识别移动应用程序的潜在数据泄漏漏洞。我们对来自Google Play商店的160多万个移动应用程序进行了评估，发现了15，098个由亚马逊、谷歌和微软等主流云供应商管理的应用程序服务器受到了数据泄漏攻击。我们已经向每个云服务提供商进行了负责任的披露，他们都确认了我们所发现的漏洞，并正在积极与移动应用开发商合作，为其有漏洞的服务打补丁。**

I. 简介

云已经大大改变了现代计算的面貌。它已被证明是数据存储、数据处理、数据分析以及数据备份和恢复的首选平台，因为它具有高度可用、大规模可扩展、极大地节省成本和可快速部署等巨大的优势。云无处不在，"在全球近70%的企业中，云的存在正成为一种常态，他们至少有一个应用程序在云上运行"[34]。

随着云计算的快速发展，移动应用程序也有了巨大的增长。从终端用户的角度来看，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。通过使用云计算，服务提供商（*例如*，新闻、天气和购物）不必担心他们的后端服务器的可扩展性和可用性，相反，他们可以只关注他们的核心业务逻辑，并开发他们的移动应用程序。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。截至目前，仅在Google Play Store和Apple App Store中就有超过500万个移动应用[20]。不幸的是，随着云计算作为移动应用的后端，我们最近也目睹了来自云计算的大量数据泄露，从个人医疗记录到企业机密。例如，据报道，不安全的移动应用程序的后端数据库暴露了大约2.8亿条敏感的用户记录，包括个人身份信息（PII），如用户名、密码。电子邮件、电话号码和位置[36]。这种泄漏也适用于高知名度的公司，如Verizon，它可能从一个可公开访问的Amazon S3桶中泄漏了100MB的企业机密[33]。

一个传统的智慧是通过使用云来获得更好的安全性，但这实际上导致了大规模的数据泄露。因此，它导致了许多问题需要回答。例如，云中数据泄露的根本原因是什么？它们是由云供应商、应用程序开发人员还是两者都造成的？我们能否系统地识别云服务中的数据泄漏漏洞？他们（*即*，云供应商和应用开发者）如何防止它们再次发生？

在本文中，我们试图进行系统研究并回答这些问题。由于我们没有较低层次的权限来了解每个云供应商如何管理客户的服务，我们只能分析云服务的前端。其中一个前端是移动应用程序。越来越多的人注意到，越来越多的移动应用正在使用云的API来提供各种服务，如认证、授权和存储，而不直接建立和管理这些后端基础设施。因此，通过检查云端API的使用方式，我们可以了解每个移动应用是如何管理其客户数据的，从而找出数据泄露的漏洞。

特别是，我们首先研究了云提供商为移动应用开发提供的典型API，并研究了应用开发者如何利用这些API开发他们的移动应用并管理他们的安全。从这项研究中，我们发现移动应用在与云后端通信时必须执行额外的服务认证，这样云提供商就知道是哪个应用发出的请求，以及这个应用旨在访问哪个资源。用户认证的管理不当和授权中用户权限的错误配置是导致云端各种数据泄露的根本原因。在我们的研究中，我们发现许多云服务都存在着这样的漏洞。

在发现了云端数据泄露的根本原因后，我们也注意到有可能开发一种原则性的方法，通过检查移动应用程序如何使用认证密钥和服务器如何处理用户的请求来自动识别这些数据泄露的漏洞。一个挑战在于如何确保在进行分析时没有客户数据的泄漏，因为我们是第三方，我们当然不能访问任何客户数据。另外，今天有数以百万计的移动应用程序，所以我们必须设计一个可扩展的、自动化的、高效的方法。我们已经解决了这些挑战，并建立了一个名为LeakScope的工具，其中有一套程序分析技术，包括抗混淆的云API识别、字符串值分析和零数据分析。

云服务	供应商	数据库管理	用户管理	储存	通知书交付
AWS 蔚蓝火 碱	亚马逊 微软 谷歌	DynamoDB简易 表 NoSql数据库	识别 阿苏活动目录 Firebase认证	亚马逊S3 Azure 存储 谷歌存储	亚马逊SNS通知枢纽 通知

表一:流行的云供应商为移动应用开发提供的关键组件。

泄露漏洞识别，自动定位移动应用中的云数据泄露漏洞。

我们已经用1,609,983个移动应用程序评估了LeakScope。令人惊讶的是，我们的工具发现了15,098个独特的移动应用程序（其中10个有1亿到5亿用户），其后端服务器--托管在亚马逊AWS、谷歌Firebase和微软Azure等云中--受到了数据泄漏攻击。我们已经向所有的云供应商进行了负责任的披露，他们都确认了我们发现的漏洞。我们还依靠他们进一步通知应用程序开发人员，向他们发送有漏洞的应用程序的清单和其他一些详细信息。云提供商已经认真考虑了我们的发现，并通知了有漏洞的应用程序开发者。云提供商也在积极地修补他们的服务，其中一些可以公开注意到（例如，更新官方文档以纠正SDK中一个错误的例子[8]）。此外，一些流行的移动应用程序也在云供应商通知他们后打了补丁。

简而言之，我们做出了以下贡献。

- **系统性的研究**。我们向系统地了解云中的数据泄漏迈出了第一步，我们的研究已经确定了最近大规模数据泄漏的根本原因。
- **自动化技术**。我们开发了一套程序分析技术，包括新的抗混淆云API识别、字符串值分析和零数据泄露漏洞识别，以自动定位移动应用程序的云数据泄露漏洞。
- **实证评估**。我们用1,609,983个移动应用程序评估了我们的技术，并确定了15,098个独特的移动应用程序，它们在云中运行的服务受到了数据泄漏攻击。我们已经进行了负责任的披露，其中一些服务已经打了补丁。

II. 背景情况

在这一节中，我们介绍了与为什么会有用于移动应用开发的云计算API（§II-A）和如何使用它们（§II-B）有关的背景。

A. 为什么在移动应用开发中使用云API

在云计算API开始流行之前，当开发一个移动应用程序（在Android或iOS上运行）时，开发人员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（尽管他们可以从云服务提供商那里租用一些基础设施，*如*虚拟机或数据库服务器）。此外，即使在发布应用程序后，整个系统也需要付出非同小可的努力来确保其稳定性和安全性。此外，当一个应用程序变得流行时，他们将不得不保持高可用性，并扩大系统以适应潜在的大规模用户增长。在认识到移动应用开发者的需求后，主流的云供应商已经提供了移动后端

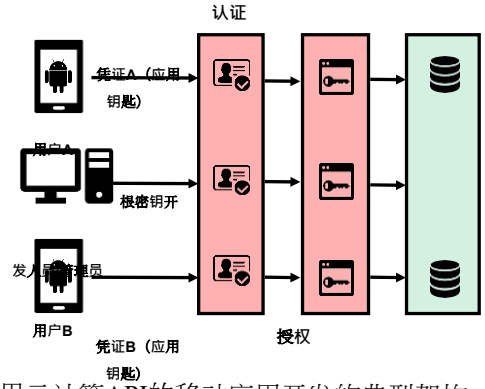

图1：使用云计算API的移动应用开发的典型架构。

作为一种服务（mBaaS），包括相应的平台和API，用于快速移动应用开发。例如，通过几个配置步骤，开发者可以快速建立一个功能齐全的移动应用后端。使用mBaaS云的好处是，开发人员不必担心如何建立和管理后端服务器，以及处理大规模的请求。相反，他们可以只专注于开发他们的前端移动应用程序，并依靠云供应商提供基础设施和管理服务器的安全性、可靠性和可扩展性。

截至目前，有许多mBaaS云供应商，包括亚马逊、Backendless、谷歌、Kinvey、微软、Oracle等。[11].由于我们的人力有限，在本文中，我们只关注亚马逊、谷歌和微软提供的mBaaS云，尽管我们的方法应该也适用于其他云。如表一所示，每个云提供商提供四个典型的组件，包括数据库管理、用户管理、存储和通知交付，以及相应的API，如`setValue`和`removeValue`[13]，`createUserWithEmailAndPassword`[1]。`sendPasswordResetEmail`[10]，`putFile`[17]，列出文件和目录[2]，`CreatePlatformEndpoint`[4]，以及发布[18]。

B. 如何使用云计算API

与开发没有云API的移动应用程序相比，一个重要的区别是，云需要知道谁发出了API调用（出于各种目的，如隔离、问责和安全）。因此，云计算API经常把一个应用程序的密钥作为参数之一。还有一个由云供应商发出的密钥，即根密钥。这两个密钥是完全不同的。

- **应用钥匙**。一个应用程序密钥具有非常有限的权限。有了这个密钥，云就知道API调用来自哪个应用，而且只有属于该应用的公开可用资源才能被访问。由于多个

服务	钥匙类型	例子
阿兹维尔存储	帐户钥匙	DefaultEndpointsProtocol=https。AccountName=*;AccountKey=*
	SAS	https://*.blob.core.windows.net/*?sv=*&st=*&se=*&sr=b&sp=rw&sip=*&spr=https&sig=*
通知枢纽	倾听的关键	Endpoint=sb://*.servicebus.windows.net/;SharedAccessKeyName=DefaultListenSharedAccessSignature。共享的访问键=*。
	完全访问密钥	端点=sb://*.servicebus.windows.net/。SharedAccessKeyName=DefaultFullSharedAccessSignature; SharedAccessKey=*。

表二：使用微软Azure云进行移动应用开发时使用的密钥实例。我们使用符号*对密钥中的敏感数据进行匿名处理。

应用程序可能在后端使用相同的云数据库和存储，云供应商通过使用应用程序密钥来虚拟和隔离这些资源。

- **根钥匙**。根密钥具有非常强大的权限。有了这个密钥，所有属于它的后端资源都可以被访问。通常情况下，开发人员应该对这个密钥保密，因为只有系统管理员才能使用它。

图1说明了云供应商发放的应用程序密钥和根密钥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（如表）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，以访问它所需要的云资源。

III. 我们的发现

在解释了如何使用mBaaS云进行移动应用开发的细节后，我们接下来将揭开应用开发者和云供应商所犯的常见错误，以及由此带来的漏洞。由于云供应商通过使用他们的API使移动应用开发变得更加容易，应用开发者的舞台变得更小。开发者的安全责任减少到正确地执行认证和授权，以便

移动应用程序的用户。这些操作的任何错误都会导致账户受损，如数据泄露和数据篡改。特别是，我们已经确定了两种主要的数据泄露漏洞：在授权中滥用各种密钥（§III-A）和在授权中错误配置用户权限（§III-B）。

A. 滥用认证中的各种密钥

在使用云API时，有必要传递相应的密钥，以便云供应商能够将应用程序与相应的虚拟化后端服务器连接起来。钥匙滥用的根本原因是，开发人员忘记了应用钥匙和根钥匙之间的区别，而且云提供商没有强制执行钥匙的正确使用。更具体地说，在使用云API时，如果一个云接口同时接受应用密钥和根密钥，这将给开发者带来混乱，从而导致漏洞。我们发现，亚马逊和微软的云都存在这个问题，我们已经确定了密钥的滥用，在

```
1 包 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    如果(Utils.isNetworkAvailable(this.imgContext)) {
24      new AsyncTask(v1) {
25        字符串 val$conStr;
26        public AsyncTask(String arg1){
27          this.val$conStr = arg1;
28        }
29        protected void doInBackground(Void[] arg17) {
30          字符串v0 = this.val$conStr;
31          CloudStorageAccount v7 = CloudStorageAccount.parse(v0)。
32 ...
33 包 com.appname.Utils
34 public class Utils {
35   public static String getStringFromResources(Context arg1,
36         int arg2) {
37     资源v0 = arg1.getResources()。
38     String v1 = v0.getString(arg2)。
39     返回v1。
40   }
41 ...
```

图2：从一个真实的Android应用中获取Azure存储的反编译代码样本。

(i) Microsoft Azure Storage, (ii) Azure Notification Hubs, 和 (iii) 亚马逊AWS S3。

(I) Azure存储中的密钥误用。微软Azure云为开发者提供了两种密钥来访问其存储。

- **帐户密钥**。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、Blobs和向特定用户收费的虚拟机磁盘。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。表二的第一行介绍了这样一个密钥的例子。
- **共享访问签名（SAS）**。SAS提供对属于特定Azure存储账户的资源的委托访问。与账户钥匙不同，开发人员可以配置SAS的权限（*例如*，读、写），以便钥匙只能以有限的权限访问某些资源。表二的第二行介绍了一个SAS的例子。

尽管账户密钥和SAS有完全不同的格式和使用情况，但我们发现开发者实际上已经滥用了它们。很明显，开发者应该在应用程序中只使用SAS，并对账户密钥进行保密。例如，当一个用户试图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。然而，根据我们的实验，许多开发者直接在移动应用程序中使用他们的账户密钥。不幸的是，账户密钥很容易被攻击者通过反向工程提取。如图2所示，我们从一个真实的Android应用中提取了代码，其名称为

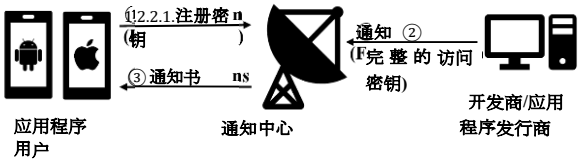

图3：如何使用Azure通知枢纽。

特意进行了匿名处理。该代码用于从云存储中下载一些图片。我们可以看到，这个应用程序中使用的密钥（在第31行）是一个账户密钥（通过查看其在第18行的格式），这意味着任何拥有这个密钥的人都可以
访问分配给这个特定账户的全部存储。

(II) Azure通知枢纽中的关键误操作。通过云端API，开发者可以轻松地将消息发送给特定的用户或广播给所有用户。微软Azure提供了一个通知枢纽，有两个API用于消息通知。如图3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的频道；之后，应用服务器可以向枢纽中注册的频道推送通知（使用完整的访问密钥），这将进一步把这个消息转发给所有频道的监听器。这里涉及两个密钥。

- **完全访问密钥**。完全访问密钥的工作方式类似于"根"密钥，它有完全的权限来发送或监听通道上的通知。表二的第三行显示了这样一个密钥的例子。
- **监听键**。监听键具有有限的权限，只能监听在特定通道中注册的通知。表二中的最后一行显示了一个监听密钥的例子。

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。然而，我们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，但我们必须强调，一旦攻击者提取了完全访问密钥，他们仍然可以获得强大的功能。例如，他们可以很容易地推送钓鱼信息，窃取其他应用程序用户的数据。

(III) AWS中的密钥滥用。亚马逊AWS也为移动应用程序提供了一些密钥，以访问AWS的资源（*例如*，S3存储），并向开发人员收费。其中一个密钥是根访问密钥（或根账户密钥），它可以完全访问特定AWS账户下的所有资源。由于这个根密钥拥有所有的权限，所以它应该被保密。不幸的是，我们也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，她就可以完全访问整个存储。

B. 授权中的用户权限配置错误

除了在认证中滥用密钥外，在授权中对用户权限的错误配置也会导致数据泄露。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。缺乏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。

```
{
  "规则"。{
    "用户"。{
      "$uid"。{
          ".读":"$uid === auth.uid",
        ".write":"$uid === auth.uid"
      }
    }
  }
}
```

图4：一个正确的Firebase授权规则

```
{
  "规则"。{
    ".read": true
    。
    ".write": true
  }
}
```

(a)

```
{
  "规则"。{
    ".读"。"auth != null"。
    ".写"。"auth != null"
  }
}
```

(b)

图5：两个配置错误的Firebase授权规则

虽然授权的错误配置并不是一个新问题，而且已经被研究了很多年，但在使用云API时，这个问题变得更加关键。特别是在开发没有云API的移动应用时，不同的开发者可以用完全不同的方式实现授权系统。即使可能存在漏洞，但对于攻击者来说，系统地利用许多不同的实现方式是相当有挑战性的。然而，当开发者使用云端API时，对手可以很容易地发起攻击；因为许多开发者只使用少数云端服务的接口，攻击者只需关注这些接口就可以系统地发现漏洞。

同时，与认证不同的是，在认证中，开发者在使用云端API时的发挥空间有限（*例如*，错误只来自于密钥的误用），而在授权中，开发者有多种配置方式，这使得授权的配置难度更大，更容易出错。例如，谷歌甚至为开发者提供了一种语言来指定授权中的用户权限。开发者会犯错误并不奇怪：我们发现（i）使用Firebase的应用程序在后端错误地配置了用户权限。此外，我们还发现（二）使用AWS的应用程序也存在这种错误配置问题。

(I) 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4中显示了这样一个规则的例子。请注意，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"用户
"的节点：当一个用户试图访问（读/写）它的一个子节点，该节点有一个相关的`$uid`，系统将只在该用户的uid等于
子节点的`$uid`，用于这个特定的读写操作。不幸的是，我们注意到，并不是所有的开发者都遵循正确的方式来编写规则。例如，如图5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。这些显然是不安全的规则，因为它们意味着任何（经过认证的）用户都可以完全访问整个数据库。

(II) 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。为了安全地访问计入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，这将导致数据泄漏。

IV. 问题陈述和概述

在发现云中数据泄漏的根本原因后，我们希望开发技术来系统地识别这些泄漏。由于我们无法进入mBaaS云的实现（只能进入它们的SDK和API），我们只能从服务的前端（*即*移动应用程序）开始，检查移动应用程序如何使用各种密钥，并根据服务器的响应推断出权限配置。因此，我们必须设计一个移动应用专属的方法。在本节中，我们将概述我们旨在解决的问题以及我们的解决方案和洞察力。我们的工具的详细设计和实现将在下一节介绍。

A. 问题陈述

如图1所示，我们注意到，开发者必须将应用密钥嵌入到应用中，然后用它来调用云端的mBaaS云端API。我们旨在识别的数据泄漏漏洞主要是由密钥滥用和权限错误配置造成的。因此，*我们必须解决的第一个问题是如何系统地识别移动应用所使用的各种密钥。*

起初，通过检查每个云API（例如，图2中第31行使用的`CloudStorageAccount.parse` API）来识别移动应用程序使用的API密钥可能显得微不足道。然而，密钥可能不是直接可见的，可能经过了多个字符串的获取（*例如*，在第9-12行）和连接（*例如*，在第18-21行）。此外，有数以百万计的移动应用程序，所以我们必须设计一个可扩展的方法。因此，*我们必须解决的第二个问题是如何识别移动应用程序使用的相关关键字符串。*

另外，越来越多的移动应用程序使用混淆技术来阻止应用程序的逆向工程和重新包装[45], [43], [42]。应用程序开发者有可能混淆了我们要检查的API。因此，我们不能简单地扫描应用程序的代码以获取API签名（*例如*，`CloudStorageAccount.parse`）；相反，我们必须设计一种抗混淆的方法。考虑到使用混淆技术的应用程序开发人员可能更了解安全问题，了解混淆的应用程序是否容易受到数据泄露的影响也很有趣。因此，*我们需要解决的第三个问题是如何设计一种抗混淆的方法来识别我们感兴趣的云API和关键字符串。*

最后，在我们提取了密钥后，我们必须确定每个密钥的类型（*例如*，根密钥或应用程序密钥）。一旦我们确定了密钥的类型，我们还必须验证应用程序是否滥用了它们，而没有意外地访问存储在云中的任何私人数据。同样地，我们也必须在授权过程中识别用户权限的错误配置，而不泄露任何数据。因此，*我们（作为第三方）必须解决的最后一个问题是设计一个零数据泄漏的验证方法，以确认云中是否存在数据泄漏。*

B. 我们的解决方案

通过云API识别来识别密钥。为了识别一个应用程序所使用的密钥，我们实际上可以从应用程序所使用的知名API的参数中推断出它们。请注意，每个云供应商都在他们的SDK中为移动应用开发提供了一系列的APIs。如表三所示，采取各种键的参数的API实际上是相当有限的，我们根据对相应的SDK的最佳理解来获得这个列表。因此，如果我们能够识别这些API，那么我们就可以识别应用程序在相应参数中使用的密钥。

然而，表三中列出的API是可以被混淆的。有趣的是，我们发现，在API混淆中往往有两种策略。

- **将**API中涉及的**名称**，如子包名、类名、函数名、变量名等，从标准名称**重命名**为一些无意义的字符。这通常是由一些自动混淆工具实现的（*例如*，Dexprotector [7], Dexguard [5]）。
- **移除**从未使用过的**函数/API**。由于不使用的函数/API通常可以揭示应用程序使用的包和类，通过自动工具（*例如*，Proguard [12], [15]）从应用程序中删除这些函数可以进一步帮助隐藏感兴趣的API。

因此，我们建议为APK中的每个函数（包括其库）建立一个抗混淆的函数签名，以识别表III中的云API。我们的签名忽略了包、类、函数和变量的名称。相反，一个函数的签名是由参数、局部变量和返回值的类型以及调用者的签名组成的字符串的散列。

使用字符串分析来识别密钥的值。在确定了我们感兴趣的API后，我们不能直接从应用代码中提取相应参数的值。例如，我们不能直接从图2中的`CloudStorageAccount.parse`提取`v0`的值，因为这个值是由多个字符串操作计算出来的。虽然我们可以使用动态分析来执行应用程序并在运行时提取该值，但这种方法不能很好地扩展，特别是考虑到我们有数百万的移动应用程序。因此，最终我们决定采取静态分析的方法，提出一个有针对性的字符串值分析，以确定使用的键。在高层次上，我们的字符串值分析可以被认为是值集分析的一个特殊案例[24]。它涉及到向后切分和字符串相关的操作分析。

零数据泄露的漏洞验证。在我们检索了云API所使用的密钥值之后，接下来我们必须推断出密钥的类型。对于一些云服务（*例如*，Azure Storage），应用程序密钥和根密钥的格式不同（如表二所示），我们可以很容易地判断出我们感兴趣的密钥是否是根密钥。然而，对于其他一些云服务（*例如*，AWS），应用程序密钥和根密钥具有相同的格式。为了处理这个问题，一个直接的方法是通过使用密钥向服务器发送一个请求，以访问一些根用户的专属数据。如果我们能够检索到这些数据，那么就意味着存在着数据泄露的漏洞。然而，这种方法会违反访问私人敏感数据的道德规范。

云服务	APIs	定义	字符串的索引我们感兴趣的参数
AWS	1*	TransferUtility。TransferObserver downloadUpload(String, String, File)	0
	2*	AmazonS3Client: void S3objectAccess(String, String, ...)	0
	3	CognitoCredentialsProvider: void <init>(String,String,String,...)	1
	4	BasicAWSCredentials: void <init>(String,String)	0,1
蔚蓝	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
火焰基地	9	FirebaseOptions: void <init>(String,String,String,String,String,String)	0,1,2,5
	10	FirebaseOptions: void <init>(String,String,String,String,String)。	0,1,2,5

表三：我们感兴趣的目标mBaaS云的API。总共有32个API。由于空间有限，我们用API 1*和2*来实际代表两组API。这两组API的完整列表见附录中的表九。

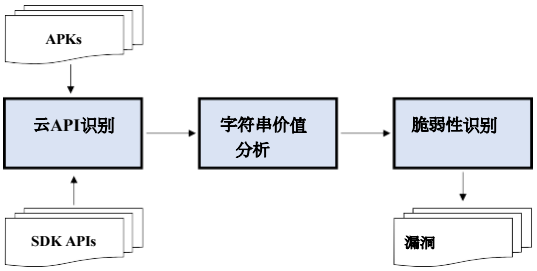

图6：我们的LeakScope的概述。

图7：图2中样本代码的包树的层次结构。

幸运的是，我们还有一个观察结果：我们注意到，当用根钥匙和普通用户钥匙访问云中不存在的数据时，服务器通常会返回不同的响应信息。因此，为了验证一个密钥是否是一个根密钥，我们将用该密钥向云端发送一个请求，以检索一些不存在的数据。如果钥匙是一个应用程序的钥匙，返回的信息通常是 "权限拒绝"的形式，如果钥匙是一个根钥匙，返回的信息通常是 "未找到数据"之类的。在这两种情况下，没有真正的数据被泄露，但我们已经推断出了密钥的类型。我们也可以用同样的方法来验证授权的错误配置。

V. 设计和实施

我们已经建立了一个名为LeakScope的工具，当给定一个移动应用时，可以自动检测数据泄露的漏洞。图6展示了LeakScope的概况。它由三个关键部分组成。云API识别（§V-A），字符串值分析（§V-B），以及漏洞识别（§V-C）。在本节中，我们将描述我们如何设计和实现（§V-D）这些组件。

A. 云API识别

识别应用程序使用的API是很重要的，特别是表三中列出的那些。正如第IV-B节中所讨论的，我们需要设计一种抗混淆的方法来识别云端API。幸运的是，我们注意到至少有两个不变量，无论给定的函数（或方法）的混淆情况如何，都会被保留下来。

- 第一个不变因素是类型和包树的层次结构（或形状，层）。请注意，每个类、方法、参数和变量都有类型（例如，图2中第15行的`arg5`的类型是`com.appname.Listeners.Callback`）。虽然类型的名称可以被模糊化（除了系统类型的名称），但类型的层次结构不会改变。例如，如图7所示，类型`Callback`被存储在从包`com`开始的第四层。
- 第二个不变因素是，在以下情况下，调用者与被调用者之间的关系
一个方法。例如，方法`downloadImage`会调用`StringBuilder.append`和`Util.isNetworkAvailable`等函数。我们可以递归地建立每个调用者的签名，然后合并它们来建立调用者的签名，即本例中的`downloadImage`。

因此，如果我们能对这两个不变量进行编码，并加上他们一起构建一个函数的签名（包括链接的非系统库中的API），然后我们就可以在移动应用的字节码中搜索这些签名，找到我们需要的云API。受LibScout[23]的启发，其中Merkle哈希树被用来建立一个库的签名（用于Android应用程序中的第三方库检测），我们使用编码不变式的哈希（*即*，类型和包树的层次结构，以及调用者-卡列关系）作为API检测的函数/API签名。请注意，我们不能直接使用LibScout，因为它专注于第三方库的检测。此外，LibScout只为每个类生成粗粒度的签名，而我们必须为每个函数生成细粒度的签名。

为每个函数生成签名。更具体地说，如算法1所示，对于一个给定的函数，我们将通过使用(i)函数的归属类（即主类）（第5-6行），(ii)其参数（第7-8行），(iii)其局部变量（第9-10行），(iv)其返回值（第11-12行），以及(v)其所有被调用者（第13-18行）的编码类型字符串的MD5散列来产生一个签名（GENFUNSIG）。如果一个被调用者是一个Android系统函数（未被混淆），那么我们直接

算法1 函数签名生成

```
1：输入：fo：目标函数；Cs：系统类；Fs：系统函数
2：程序GENFUNSIG(fo, Cs, Fs)
3:    L ← ∅
4:    fBuf←∅
5:    to ← GETHOMECLASSTYPE(fo)
6:    fBuf ← fBuf ∪ TYPEENCODING(to , to , Cs )
7:    对于tp∈GETPARAMETERTYPE(fo)做
8:       fBuf ← fBuf ∪ TYPEENCODING(to, tp, Cs )
9:    对于tv∈GETLOCALVARTYPE(fo)做
10:      fBuf ← fBuf ∪ TYPEENCODING(to , tv, Cs )
11:   tr ← GETRETURNTYPE(fo)
12:   fBuf ← fBuf ∪ TYPEENCODING(to, tr, Cs )
13:   对于fi∈GETCALLEE(fo)做
14:      如果fi∈Fs，那么
15:         L←L∪name(fi, argType(fi), retType(fi))
16:
17:      其 L ← LU GENSUNIG fi, Cs, Fs )
他
18:   fBuf ← fBuf ∪SORT(L)
19:   返回MD5（fBuf）。
20：输入：Ch ：家乡类；Ct ：目标类；Cs ：系统类。
21：程序TYPEENCODING(Ch , Ct , Cs )
22:   L ← ∅
23:   tBuf ← ∅
24:   如果ct∈Cs，那么
25:      tBuf ← tBuf ∪ name(ct)
26:   其他
27:      rp ← NORMALIZEDRELATIVEPATH(ch, ct )
28:      tBuf ← tBuf ∪ rp
29:      cp ← GETSUPERCLASS(ct )
30:      tBuf ← tBuf ∪ TYPEENCODING(Ch , cp , Cs )
31:      对于ci∈GETINTERFACES(ct )做
32:         L ← LU TYPEENCODING(Ch , ci , Cs )
33:      tBuf ← tBuf ∪SORT(L)
34:   返回 tBuf
```

在我们的类型字符串中加入这个函数的名称，包括其参数和返回值的类型名称（第14-15行）。

对类型和包树的层次结构进行字符串编码（TYPEENCODING）（第20-行）。34），基本上我们把类型所属的母体类（c_h），目标类（c），都拿出来，计算为输入系统类（C_s来对目标类型c_t进行编码。注意在Java字节码中，所有的类型都被定义为类。如果c_t是一个系统类，我们直接返回该类型的字符串名称（第24-25行），因为它的名称不能被混淆。否则，我们将结合c_t和c_h之间的规范化相对路径（第27-28行）、递归c_t的超类（第29-30行）和c_t的接口（第31-33行）的编码字符串，作为最后的类型字符串。

请注意，为了唯一地编码一个给定的类的类型，我们希望使用尽可能多的可靠信息。同样，我们不能使用任何一个类的名字（除了系统类），我们必须将它们规范化。由于一个类通常有一个超类，我们希望包括其超类的类型编码，递归地。同时，一个类可能已经定义了一些接口函数[19]。注意，一个接口是一个空函数的集合（没有实现体）。任何实现接口的类都会继承这些函数。这就是为什么最终我们的类型编码算法会考虑名称和路径规范化、超类和类的接口。我们在签名中不包括类中的其他信息，比如它的字段和方法，因为它们可以被Proguard[15]等工具删除。

在执行规范化时，我们用符号'X'替换任何非系统定义的名称，类似于LibScout [23]。此外，我们的算法将规范化的相对路径纳入我们的类型字符串。例如，当处理第一个参数的类型编码时的函数`downloadImage`的主类是`ImagesHelper`，我们采取规范化的相对路径来编码`com.appname.Listeners.Callback`类型。更具体地说，根据图7，从主类`ImagesHelper`开始，我们通过以下方式到达`com.appname.Listeners.Callback`：`.../Listeners/Callback`。在将所有非系统名称规范化为'X'后，我们得到一个规范化的路径，字符串为"./X/X"。同时，`com.appname.Listeners.Callback`的超类实际上是`java.lang.Object`，它是一个系统类（不能被混淆）。因此，最终，对于类型编码的`arg5`，我们得到字符串"./X/X#java.lang.Object"

其中#表示字符串连接。同样地，我们得到一个类型字符串，分别用于其第二个参数、局部变量和调用。我们一起为函数`downloadImage`创建了一个独特的哈希值。

用签名识别云API。通过我们的GENFUNSIG算法，我们为所有功能生成签名，包括给定应用程序的云API（最初从SDK库中获取）。然后，我们在移动应用程序中搜索云API的签名；如果签名匹配，我们就能识别出我们感兴趣的相应云API。

B. 字符串价值分析

在确定了我们感兴趣的云端API后，我们就能确定每个API的调用位置，并进一步确定移动应用中包含认证密钥的参数。然而，我们不能直接观察它们的值，因为我们使用静态分析。因此，我们必须开发一个有针对性的字符串值集分析

(VSA) [24]来揭示钥匙的可能值。注意VSA是一种技术，它分析的是寄存器和内存地址在x86二进制代码水平。我们不能直接使用它来解决我们的问题，相反，我们必须定制它来揭示移动应用字节码背景下的字符串值。在高层次上，我们的字符串值分析需要静态地执行以下*程序间的后向切片*和*字符串值计算*。

程序间逆向分片。程序切片[37]是一种广泛使用的程序分析技术，已被用于解决许多重要的安全问题，如软件漏洞诊断（如[38]）和自动补丁生成（如[35]，[44]）。在我们的字符串值分析中，我们必须首先通过应用Java字节码的后向切片来确定与我们感兴趣的最终字符串的计算相关的变量和指令。

更具体地说，我们分析的第一步是为每个方法/函数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。然后从我们感兴趣的变量开始，*例如*API `CloudStorageAccount.parse`的`v0`，我们在CFG中逆向迭代指令：如果有任何变量有助于`v0`的计算，我们就把它们添加到我们的数据依赖图（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。

导致最终字符串值的字符串操作；如果有任何函数调用，我们会进行上下文敏感的程序间分析并递归分析调用。我们不断地迭代CFG，直到我们达到一个固定点，即我们的DDG不能进一步扩展。在我们图2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。

字符串值的计算。有了跟踪的DDG和字符串计算栈，接下来我们需要计算我们感兴趣的最终字符串的值。从字符串计算堆栈的顶部开始，我们根据CFG和我们的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，我们就执行字符串追加操作；如果是getString，我们就知道它是用来从xml文件中读取字符串的，然后我们就从xml文件中执行指定字符串的读取操作并返回其结果。注意，字符串是系统定义的类，相应的API没有被混淆。

回到我们在图2中的运行例子，堆栈上最后推送的变量是StorageAccountKey和StorageAccountName。然后，从我们跟踪的DDG开始，我们在第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。接下来，我们计算v0的值（第17-21行）、v1、this.val$conStr的值，最后在第31行计算v0的值。

C. 脆弱性识别

在获得由LeakScope识别的密钥后，接下来我们想检测云服务中的数据泄漏漏洞。该检测是针对云的，我们设计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。

(I) 检测Azure（存储和通知中心）中的密钥误用。正如第III-A节中所讨论的，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，我们不需要探测后端来确认漏洞。只要我们在移动应用程序中识别出根密钥或完全访问密钥，我们就知道云服务有漏洞。

(II) 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，这意味着如果我们在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对手访问。我们发现，根密钥有权限通过以下rest API获得一些AWS实例信息：https://ec2.amazonaws.com/?Action=DescribeInstances&InstanceId.1=X，其中X是目标实例的ID。相比之下，一个应用程序的密钥没有这个权限。因此，为了检测AWS中的密钥滥用，我们将X设置为一个不存在的ID。当我们单独发送一个确定了密钥的请求时，我们会收到错误的提示消息"InvalidInstanceID"（如果钥匙是根钥匙）或"UnauthorizedOperation"（如果钥匙是应用钥匙）。

(III) 检测Firebase中的权限错误配置。如图5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）。我们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为".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"路径"中的数据库数据。通过设置"路径"，用户可以读取特定的数据。如果我们将"路径"设置为"根"，那么我们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。如果我们在尝试读取"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，但我们仍然可以确认数据泄露，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"。"auth != null"时，我们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。要在每个相应的云服务中注册一个合法的用户，这将是一个巨大的工程挑战。幸运的是，我们注意到Firebase也提供了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/Facebook SSO*等。例如，通过使用电子邮件/密码、电话号码、Google/Facebook SSO等方式*，除了开发者的用户注册方法之外。因此我们可以直接调用这些Firebase APIs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，我们对注册用户进行认证，然后进行基于索引的测试。

(IV) 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，我们只专注于S3存储的权限错误配置的检测。请注意，最近几次高调的数据泄露（例如，[33]）都来自S3。为了执行我们的测试，我们不仅要收集AWS的密钥，而且还要收集S3存储的名称。因此，我们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，我们直接调用AWS API HEAD Bucket [9]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。

D. 实施

我们在dexlib2[6]和soot[16]上实现了LeakScope[1]在dexlib2[6]和soot[16]之上。特别是，为了实现我们的云API识别，我们利用了dexlib2，这是一个轻量级的APK静态分析框架，允许轻松解析dex文件以建立函数签名。我们将字符串值分析建立在Soot之上，这是一个强大的框架，用于分析Java字节码，具有特别有用的功能，如数据流分析。漏洞识别是非常直接的；我们只是写了一个python脚本来发送请求

并解析了

[1] LeakScope的源代码可在https://github.com/ OSUSecLab/LeakScope。

	共计		非混杂的		混淆的	
	#Apps	%	#Apps	%	#Apps	%
与云计算API	107,081	-	85,357	79.71	21,724	20.29
仅限AWS	4,799	4.48	4,548	5.33	251	1.16
仅限Azure	899	0.84	720	0.84	179	0.82
仅限Firebase	99,186	92.63	78,475	91.94	20,711	95.34
与AWS和Azure合作	3	0.00	2	0.00	1	0.00
使用AWS和Firebase	1,973	1.84	1,427	1.67	546	2.51
使用Azure和Firebase	210	0.20	174	0.20	36	0.17
三种服务	11	0.01	11	0.01	0	0.00

表四：我们的云API检测结果。

响应。LeakScope总共由大约6,000行我们自己的Python和Java代码组成。

VI. 评价

在本节中，我们将介绍我们的评估结果。我们首先描述了我们的实验设置，包括我们如何在§VI-A中收集移动应用程序，在§VI-B中描述了我们的详细结果，最后在§VI-C中提供了对识别的漏洞和我们的方法的误报的分析。

A. 实验设置

收集移动应用程序。我们专注于在Google Play上发布的Android应用程序。为了从Google Play获得一个应用程序，我们必须提供应用程序的包名。因此，我们首先在`scrapy`[14]框架上开发了一个python脚本来抓取每个应用包的名称。经过两周的爬行，我们在2017年5月检索到了大约*190*万个应用程序名称。然后我们在两个月内抓取了1，609，983个免费安卓应用的全部包内容。请注意，由于付费应用或某些应用只在某些国家可用等限制，我们无法下载所有*190*万个应用。总的来说，这160万个移动应用程序消耗了我们的硬盘上有15.42TB的空间。

环境设置。我们的实验是在七个工作站上进行的，每个工作站都配备了英特尔至强E5-2640处理器，有24个内核和96GB内存，运行Ubuntu 16.04.我们不需要任何真实的移动设备，因为LeakScope主要是一个静态分析工具，只有漏洞识别组件需要使用动态分析与云服务器通信。我们所有的实验数据，包括目标应用程序和中间结果都存储在一个拥有34.90TB硬盘空间的网络附加存储器中。

B. 实验结果

LeakScope总共花费了6,894.89个单CPU计算小时来分析这1,609,983个应用程序，这消耗了2.56TB的存储来存储中间结果。在被测试的应用程序中，最终LeakScope检测到15,098个独特的移动应用程序（总共有17,299个漏洞），其云服务器受到了数据泄漏的攻击。在下文中，我们根据LeakScope的每个组件的表现，介绍详细的结果。

1) **云API识别。**我们首先用测试过的1,609,983个应用程序评估了我们的云API识别。由于我们的方法具有抗混淆性，它显然也适用于非混淆的应用程序。总的来说，我们的云计算API识别产生了39,617,809,277个函数签名，并识别了107,081个使用了部分API的移动应用。
我们感兴趣的32个云计算API。在这些应用程序中，有21,724其中（20.29%）是被混淆的。因此，如表四所示，我们将实验结果分为两组：没有混淆的应用程序（85个，357个应用程序），和有混淆的应用程序，以了解混淆的应用程序是否能更好地保护数据泄漏攻击。

我们还报告了移动应用程序使用的云服务的详细分布情况。特别是，在这107,081个应用程序中，4,799个（4.48%）专门使用亚马逊AWS。
899个（0.84%）专门使用微软Azure，99，186个（92.63%）专门使用谷歌Firebase。还有3个应用同时使用AWS和Azure，1，973个使用AWS和Firebase，210个使用Azure和Firebase。有趣的是，也有11个应用程序使用所有三种云服务。非混淆的应用程序和混淆的应用程序的详细分类分别从第4列到第7列报告。

令人相当惊讶的是，绝大多数（超过90%）的移动应用程序实际上使用了Google Firebase的后端服务，至少根据表四中报告的结果是如此。我们在进行负责任的披露时咨询了Google Firebase团队。他们告诉我们，部分原因是可能有相当一部分移动应用程序使用了亚马逊或微软的云，但没有使用他们的mBaaS云（*例如*，他们可能使用他们的IaaS云来代替）。我们的mBaaS云API识别不能 识别这些云。

2) **字符串价值分析。**在107,081个应用程序中，我们的字符串值分析静态计算了631,551个我们感兴趣的参数字符串，我们的详细性能结果报告在表五中。特别是，我们在第二栏报告了我们感兴趣的字符串参数，然后在第三栏报告了每个字符串参数所属的相应API（这些API的原始定义见附录中的表三和表九）。然后，对于非混淆和混淆的应用程序，我们在第4和第8栏报告有多少相应的API被调用，在第5和第9栏报告有多少应用程序调用了这些API，在第6和第10栏报告有多少参数字符串的值最终被解决，在第7和第11栏报告相应的百分比。

我们可以从表五中观察到，我们对AWS中2套不同的API中一个名为`bucketName`的参数感兴趣。这是因为我们需要`bucketName`来定位相应的S3存储以进行授权漏洞验证。我们还对`identityPoolId`感兴趣，它被用来检测AWS中的权限错误配置漏洞，而`accessKey`和`secretKey`显然也是我们直接感兴趣的。对于Azure，我们对参数`appURL`、`connectionString`和`appKey`感兴趣。对于Firebase，我们对`google_app_id`、`google_api_key`、`firebase_database_url`和`google_storage_bucket`感兴趣。应用程序可能在不同的地方一次或多次调用这些API。

根据应用程序代码对字符串的使用方式，String值分析已经解决了绝大部分的字符串值，如表五中第7和第11列所示，对于非混淆和混淆的应用程序。了解为什么不是所有的参数字符串都能被我们的字符串值分析所解决也是很有趣的。我们的进一步调查显示，这有两个原因。第一个是，这些参数的许多未解决的值实际上是从互联网上检索的。这种情况在Firebase中特别常见，因为Google实际上

	字符串 参数名称	APIs	非混杂的 #API-调用	#APP	#已解决的斯特拉。	%	混淆的 #API-调用	#APP	#已解决的斯特拉。	%
AWS	bucketName 桶名	1*	2,460	1,229	2,190	89.02	398	1,229	321	80.65
	identityPoolId 访	2*	2,069	1,703	2,045	98.84	444	439	442	99.55
	问密钥 secretKey	3	3,458	3,458	3,315	95.86	291	291	266	91.41
		4	3,280	1,769	2,650	80.79	277	203	199	71.84
		4	3,280	1,769	2,646	80.67	277	203	197	71.12
蔚蓝	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
	连接字符串	7	700	513	643	91.86	207	189	200	96.62
	connectionString	8	345	97	303	87.83	29	21	22	75.86
火焰基地	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

表五:我们对感兴趣的参数进行弦值分析的结果。

	根本原因	非混杂的 #Apps	%	混淆的 #Apps	%
蔚蓝	滥用账户钥匙	85	9.37	18	8.33
	滥用完全访问钥匙	101	11.14	12	5.56
AWS	根密钥滥用 "开放	477	7.97	92	11.53
	"的S3存储	916	15.30	195	24.44
火焰基地	"开放 "的数据库	5,166	6.45	1,214	5.70
	没有许可检查	6,855	8.56	2,168	10.18

表六:检测到漏洞的应用程序统计表

建议开发者从远程服务器检索密钥[3]。如果不对应用程序进行动态分析，我们就无法推断其价值。第二个原因是，一些应用程序使用加密函数来保护字符串，我们无法用静态分析来解决这个问题。

3) **漏洞识别。**有了确定的密钥和我们感兴趣的字符串，我们的第三个组件，**漏洞**识别，然后根据我们在§V-C中描述的零数据泄露策略检测漏洞，总共发现了17，299个漏洞。请注意，一个应用程序可能有多个数据泄露漏洞，我们根据有漏洞的服务来计算漏洞。

- **密钥滥用漏洞。**正如§III-A所讨论的，这些漏洞主要存在于Azure和AWS云中。根据应用程序的密钥值，以及密钥的格式，如果我们注意到应用程序的密钥是一个账户密钥或完整的访问密钥，我们就可以直接检测到Azure中的漏洞。表六（前两行）列出了Azure中易受攻击的应用程序的统计数据。我们可以看到，在907个非混淆的Azure应用中，有186个（20.51%）滥用了密钥；对于216个被混淆的应用程序，其中30个（13.89%）包含数据泄漏漏洞。对于AWS根密钥的滥用，我们在5，988个应用程序中检测到477个有漏洞的应用程序（7.97%）。
  如表六第三行所示，在798个非混淆的AWS应用程序中，有92个（11.53%）是混淆的应用程序。
- **权限错误配置漏洞。**这种类型的漏洞主要存在于AWS和Firebase云服务器。正如表六第4行所报告的，我们检测到5个中的916个有漏洞的应用程序，988个（15.30%）非混淆的应用程序，195个中的798个（24.44%）被混淆的应用程序。对于Firebase中的 "开放"数据库，我们在80,087个（6.45%）非混淆的应用程序中检测到5,166个脆弱的应用程序，在21,293个（5.70%）混淆的应用程序中检测到1,214个。对于Firebase的无权限检查漏洞，我们在80,087个非混淆的应用程序中检测到6,855个（8.56%），在21,293个混淆的应用程序中检测到2,168个。
  的21，293个（10.18%）混淆的应用程序。

我们可以从表六中注意到，最脆弱的类别（按百分比计算）是来自AWS的 "开放"S3存储的权限错误配置：非混淆的应用程序为15.30%，混淆的应用程序为24.44%。在Azure中可以观察到，经过混淆处理的应用程序往往不那么脆弱（13.89%对20.51%）。然而，在AWS和Firebase中，被混淆的应用程序甚至更加脆弱（除了Firebase的 "开放"数据库）。这可能是因为错误配置的错误是针对产品的，与用户的安全专业知识联系较少。

C. 漏洞分析

严重性分析。接下来，我们想研究一下我们发现的移动应用程序中的漏洞的严重程度。我们用有漏洞的应用程序的下载次数来描述其严重性：下载次数越多，漏洞就越严重。为此，我们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。这一结果在表七的最后四栏中报告。对于非常受欢迎的应用（如果一个应用的总下载量超过100万，我们就定义为非常受欢迎），这些应用使用了云API，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。显然，我们研究的数据泄露漏洞是相当令人担忧的。如果攻击者利用了这些漏洞，那么数十亿的敏感数据记录可能已经被泄露了。

混淆与非混淆。既然我们能够区分非混淆和混淆的应用程序，我们也想了解混淆对应用程序安全的影响。值得注意的是，混淆通常被应用于顶级下载的应用程序。如图七所示：一个应用程序的下载量越高，就越多

#下载	# 非易受攻击的应用程序				# 脆弱的应用程序			
	蔚蓝	AWS	火焰基地	混淆%的	蔚蓝	AWS	火焰基地	混淆%的
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

表七：在每个累计下载类别中使用过云计算API的应用程序的数量。

	应用程序名称	应用程序描述和功能	被掩盖了？	数据库/存储器中的数据	对隐私敏感？
AWS	A1	发送具有多种花式功能的信息		用户照片	
	A2	用神奇的增强手段编辑用户照片		用户照片	
	A3	用特色专业编辑用户照片		用户照片；发布的图片	
	A4	允许用户组织和上传照片	×	用户上传的图片	
	A5	帮助用户计划和预订旅行		用户照片	
	A6	一个建造和设计有吸引力的酒店的游戏应用	×	用户备份	
	A7	一个表达对游戏NPC的报复的游戏应用程序	×	高级插件	×
	A8	推送新闻并允许用户报道新闻	×	用户上传的图片和视频	
	德普普	帮助用户管理和联系他们的联系人		用户语音信息	
	A9	推送新闻并允许用户报道新闻	×	用户上传的图片和视频	
蔚蓝	A10	帮助用户开始节食并控制体重		用户照片；发布的图片	
	A11	计算和跟踪人类健康的卡路里	×	用户照片	
	A12	从通讯员工具包中显示生育状况	×	用户上传的图片	
	A13	帮助用户轻松地玩一个流行的游戏	×	关于游戏的配置	×
	A14	一个实时的翻译工具，用于通话、聊天等。	×	用户照片; 聊天记录	
	A15	显示各国纪念币的图片		钱币图片	×
	A16	一个方便的工具，用丰富的内容来做笔记		用户上传的图片	
	A17	为用户安排出租车的便捷工具	×	司机照片	
	A18	允许用户购买/续保一般保险	×	检查视频	
	A19	提供准确的当地天气预报		设备信息（IMEI等）。	
火焰基地	A20	编辑和增强用户的照片和自拍	×	用户信息（CD@）；用户私人信息	
	A21	允许用户猜测有关音乐的信息		音乐细节	×
	A22	允许用户销售和购买多种产品	×	用户信息 (@@); 交易	
	照片拼贴	用个人照片创建照片拼贴画		用户信息 (@@)	
	A23	帮助用户翻译和学习语言		用户信息（CD）；测验数据	
	A24	为卡通头像编辑带效果的用户照片	×	用户信息（CD）；用户图片	
	A25	帮助用户学习如何绘制人体		用户信息（CD@@）；用户图片	
	A26	一个带有文本和音频的离线圣经学习应用程序	×	用户信息 (CD@@)	
	A27	嘻哈混音带和音乐的音乐平台	×	用户信息（CD@@）；播放列表	
	A28	帮助用户学习绘制不同的东西		用户信息（CD@@）；用户图片	

表八：对每个云类别的前10个易受攻击的应用程序的详细研究。请注意，符号CD表示用户名，@表示用户ID，@表示用户电子邮件，@表示用户令牌。

可能是被混淆了。我们认为这是因为这些应用程序的开发者更有可能拥有更好的安全心态。然而，即使这些应用程序可能已经被混淆了，我们仍然可以检测到它们的数据泄露漏洞。(事实上，许多被检测出有漏洞的顶级应用程序都被混淆了，如表七所示）。这是因为我们的关键技术是抗混淆的，无论一个应用程序是否被混淆，我们仍然能够解决我们感兴趣的绝大多数字符串，如表五的第7和第11列所示。因此，混淆并不能帮助开发者打败数据泄露攻击；他们必须实施适当的认证和授权，以防止这些攻击。

假阳性分析。LeakScope首先使用静态分析来识别感兴趣的字符串（例如，应用程序使用的各种密钥），然后使用动态分析，通过检查对我们泄漏探测请求的响应来确认数据泄漏。有

在确定存储在云中的数据是否会被泄露方面没有误报。也就是说，对于我们检测到的所有易受攻击的应用程序，其服务器都会受到数据泄露的攻击。然而，可能会有这样的情况：开发者可能故意让他们的数据开放。要真正决定LeakScope在这方面是否有误报，我们必须看一下数据本身。如果相应的泄露的数据对隐私不敏感，那么它就是一个假阳性。

为此，我们从应用程序中手动注册了一个用户账户到相应的云服务器，并对应用程序的代码和网络流量进行逆向工程，以了解存储在云中的数据是否具有隐私敏感性。由于我们的人力有限，而且对被混淆的应用程序进行逆向工程也是一个巨大的挑战，我们无法确认所有的应用程序数据，因此我们只关注前10个最流行的易受攻击的应用程序，我们对每个测试的云端都有最好的了解。

表八中报告了每个应用程序的详细报告和可能被泄露的数据。请注意，我们希望保持应用程序名称的匿名性，因为并不是所有的应用程序都已经打了补丁，因此我们只报告Google Play中显示的应用程序的名称（第2列），如果其漏洞已经被修补（截至2018年5月，有两个应用程序的服务器已经被修补。），然后是应用的描述和功能（第3栏），这个应用是否已经被混淆（第4栏），这个特定应用的云服务器可能泄露的具体数据（第5栏），最后是这些数据是否是隐私敏感数据（最后一栏）。

对于使用过AWS的前10个易受攻击的应用程序，我们注意到许多都存储了用户照片：要么是用户头像，要么是用户拍摄的照片。还有一些其他文件，如视频和配置。有趣的是，有两个新闻应用使用AWS进行存储（但新闻内容没有存储在AWS中）。特别是，它们允许用户报告新闻，如目击的事故，同时允许用户附加有关报告新闻的图片或视频。显然，攻击者可以很容易地抓住这些文件。我们还必须强调，攻击者也可以篡改存储在AWS中的文件的完整性。我们对使用Azure的脆弱应用程序也有类似的观察。大多数数据是隐私敏感的，如用户照片、聊天记录和视频。例如，第9个应用程序，一个与汽车保险有关的应用程序，允许用户拍摄和上传汽车检查的视频。我们认为这些文件显然应该得到保护。

与AWS和Azure不同的是，Firebase是一个数据库，包含各种数据记录。正如表八第5栏所报告的，我们根据这些数据记录的类别，如用户名、用户ID和用户电子邮件，对其进行了总结。虽然大多数数据是隐私敏感的，但我们注意到有一个应用程序只在数据库中存储与音乐有关的数据。特别是第2个应用，一个与音乐相关的应用，允许用户猜测歌曲的信息。其数据库中的所有数据都与音乐有关，如音乐ID、音乐下载URL和歌手。虽然它不包含任何隐私敏感数据，但任何人都可以通过一个HTTPS请求获得整个数据库。我们相信这不是开发者的初衷（*例如*，竞争对手可以很容易地用这些数据建立一个类似的系统）。

总之，LeakScope能自动发现的是存储在云后端的数据可能被泄露的情况。要真正确定LeakScope是否有误报，最终用户和服务提供商都需要对数据是否对他们很重要、是否对隐私敏感进行分类。目前，我们还没有一种自动技术，尽管我们对30个易受攻击的应用程序进行的人工分类表明，这些应用程序的数据中有86.7%确实是隐私敏感的。

VII. 讨论

我们的研究发现了数以万计的包含云数据泄漏漏洞的移动应用程序。这些应用程序的累计下载量在40亿到140亿之间。因此，这是一个非常严重的安全问题。在这一节中，我们将进一步讨论为什么存在这样的漏洞和对策（§VII-A），局限性和未来的工作（§VII-B），最后是我们在研究中如何处理道德问题（§VII-C）。

A. 根本原因和对策

云中的数据泄漏有很多原因。第一个是"隐蔽的安全"。开发者可能认为没有人可以找到他们的（根）钥匙。但不幸的是，通过对（混淆的）移动应用程序的简单逆向工程，对手可以很容易地提取各种密钥并直接使用它们与服务器通信。第二个原因是使用SDK时缺乏安全培训。例如，使用根密钥与服务器进行通信绝对是一场安全灾难。在访问特定资源时不验证用户的身份也是一个巨大的错误。

对此，向开发者提供安全培训是缓解这一问题的直接措施。云提供商应该在其手册中清楚地记录下开发者可能犯的各种错误及其后果，最重要的是提供使用密钥的正确方法（而不是错误方法）。事实上，非常令人惊讶的是，我们发现Azure文档中的一个官方例子实际上误用了根密钥（而不是使用SAS密钥），从移动应用中与云后端进行通信。这也解释了为什么在Azure中有那么多的密钥误用。

更重要的是，云供应商还应该提供更好的安全工具和SDK来帮助开发者。例如，SDK应该进行类型检查，云后端也应该拒绝不正确的使用钥匙。SDK还应该使安全策略规范更容易，特别是对于谷歌Firebase，可以提供更多的模板或GUI界面，使开发者更容易遵循。最后，云供应商也可以开发安全工具来检测数据泄漏（例如，通过定期检查云后端不安全的访问控制策略）。

B. 局限性和未来工作

虽然LeakScope检测到了许多来自移动应用的云端后端的数据泄漏漏洞，但显然它并不完美，有很多局限性。首先，它有假阴性。这是因为对漏洞的检测是基于表III中列出的API。如果有任何其他的API在其参数中也涉及到应用程序或开发者凭证，LeakScope就会错过对这些字符串的识别。在我们未来的工作中，我们想把重点放在更系统地检查云提供商SDK的所有API上。

第二，我们的字符串值分析不能识别直接生成的值，例如从远程服务器收到的值，因为我们使用静态分析而没有实际执行应用程序。例如，有404个应用程序，LeakScope未能从中提取我们感兴趣的字符串。请注意，这也是造成假阴性的原因之一。为了处理这些应用，我们计划使用动态分析来运行这些应用，钩住我们感兴趣的API，并提取相应的参数。这是我们未来的另一项工作。

第三，由于道德方面的考虑（§VII-C），我们只用最大的努力来验证数据泄露的漏洞。例如，我们只在Firebase中发现了9，023个无权限检查漏洞。事实上，这是因为在我们的数据集中，使用Firebase的101，380个应用程序中，我们只能自动注册13，506个用户。如果有任何其他方法可以绕过其余87,874个应用程序的认证，或者有来自云端的协作，我们应该能够发现更多的漏洞。

供应商。增加我们分析的覆盖面是我们未来工作的第三个途径。

最后，LeakScope只关注使用云API来开发移动应用程序的应用程序。显然，有相当一部分应用在其后端直接使用了其他类型的云服务（*如*，IaaS云）。如何以一种有原则的方式识别这些应用程序及其易受攻击的云服务，就像LeakScope为mBaaS所做的那样，是我们未来工作的另一个方向。

C. *伦理学*

由于我们的工作旨在识别云中的数据泄漏漏洞，我们必须确保我们的研究不会直接泄漏任何客户的数据。正如第V-C节所述，我们考虑到了道德问题，并通过考虑服务器如何根据不同的用户角色来响应客户的请求，设计了一种零数据泄露的漏洞识别方法。尽管这种方法限制了我们可以识别的漏洞的数量，但它对客户数据来说是安全的。

此外，我们已经向每个云供应商进行了负责任的披露，并通过他们可以接触到移动应用程序的开发者。所有的云供应商都在积极解决我们报告的问题。例如，我们从谷歌了解到，他们已经立即警告了有漏洞的Firebase用户，并且正在监控漏洞的修补过程，特别是对于那些超级流行的应用程序（w/在1亿到5亿用户之间）。

此外，在过去的几个月里，我们也一直在与云计算供应商接触，探讨如何检测、缓解和防止这些数据泄露的漏洞。更重要的是，作为我们研究后果的一部分，谷歌已经计划在配置用户授权权限时，提供更方便的开发者SDK。Azure在其最近的git提交中纠正了关于如何使用正确的密钥与云通信的文档[8]。

VIII. 相关的工作

移动系统的漏洞识别。开发移动应用程序类似于开发传统软件，开发人员可能会犯错误，从而导致各种安全漏洞。在过去的许多年里，大量的工作都集中在识别移动应用程序的各种漏洞上。早期的努力集中在识别隐私泄露上，因为用户的GPS坐标、地址簿等都可能被意外泄露。TaintDroid[28]、PiOS[27]和AndroidLeaks[29]是这些努力的例子。他们利用动态分析来跟踪敏感信息（*例如*，地址簿）是否会被泄露，或者利用静态分析来识别泄露。

除了移动应用程序的隐私泄露，还有其他安全漏洞。例如，组件劫持漏洞[32]允许攻击者劫持流程并执行未经授权的读写操作，代码注入漏洞[30]使攻击者能够将恶意的Javascript代码注入移动应用程序，而悬挂属性引用漏洞[21]允许恶意应用程序获得关键的系统能力。相应地，一些工具，如CHEX[32]和Harehunter[21]已经被开发出来以识别它们。

最近，也有一些努力来确定移动应用程序的服务器端漏洞。比如说。如果服务器没有跟踪用户的登录尝试次数，就会出现密码暴力攻击[47]；如果商家服务器没有验证支付信息，就会出现免费购物[40]；如果服务器没有检查来自应用程序的请求，就会出现SQL注入和服务器API滥用漏洞[46]，[49]，以及在服务器授权中使用不安全的用户令牌（*例如*，没有随机性）[48]。也有相应的工具，如AutoForge[47]和AuthScope[48]来识别它们。

错误配置的漏洞检测。复杂的软件系统，如mBaaS云，很难配置和管理，因此，各种配置错误可能会被引入。不正确的访问控制配置，如LeakScope旨在发现的权限错误配置，显然会导致安全漏洞的出现。与由应用程序开发人员的错误造成的关键误用漏洞不同，权限错误配置是由系统管理员造成的。

为了检测访问控制系统（如防火墙）中的权限错误配置，FIREMAN[41]使用防火墙配置的符号模型检查来推断政策违反和不一致的情况。在典型的应用系统（*如*医疗）中，Bauer等人[25]将关联规则挖掘应用于访问控制日志，以推断出预期的政策和错误的配置。在一个企业网络中，Baaz[26]通过监控访问控制元数据的更新和观察对等体之间的不一致来推断权限的错误配置。

也有许多努力通过配置测试来检测软件系统中的错误配置。ConfErr[31]是一个黑盒配置测试工具，它通过注入拼写错误、结构错误和语义错误来暴露配置错误。ConfAid[22]是一个白盒配置诊断工具，它探索与错误行为有关的控制和数据流，以配置文件中的特定标记。SPEX[39]也是一个白盒配置测试工具，它根据配置参数的读取和使用方式产生配置错误。

作为一个黑盒测试工具，LeakScope只探索服务器响应信息的差异，以推断云服务器是否正确配置了用户的权限。我们相信云供应商肯定可以超越黑盒测试，相反，他们可以开发白盒方法来主动检测权限的错误配置。

IX. 结论

我们研究了为什么最近有这么多来自云端的私人数据泄露的问题，我们发现移动应用认证中各种密钥的滥用和authorization中用户权限的错误配置是能够导致云端大量数据泄露的两个根本原因。我们设计并实现了LeakScope，以自动识别可能包含来自移动应用的数据泄漏漏洞的云服务。我们对来自Google Play商店的*160*多万个移动应用程序进行了评估，发现了数以万计的有漏洞的云服务，包括来自Google、Amazon和Microsoft的云服务。我们已经向每一个有漏洞的服务提供商进行了负责任的披露，他们都确认了我们发现的漏洞，并且正在积极与移动应用开发商合作，为他们有漏洞的服务打补丁。

鸣谢

我们感谢匿名审稿人的宝贵反馈。我们还要感谢Erick Bauman和Atanas Rountev对本文早期草案的有益评论。这项工作得到了AFOSR的FA9550-14-1-0119资助，以及NSF的1718084、1834213和1834215资助。任何意见、发现、结论或建议都是作者的，不一定是AFOSR和NSF的。

参考文献

[1] "在安卓上使用基于密码的账户用Firebase进行认证，" https://firebase.google.com/docs/auth/android/password-auth。

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

[3] "安全使用api密钥的最佳实践，" https://support.google.com/cloud/answer/6310037。

[4] "创建平台端点并管理设备令牌"，http://docs。aws.amazon.com/sns/latest/dg/mobile-platform-endpoint.html。

[5] "Dexguard 安卓 混淆器"，" https://www.guardsquare.com/dexguard。

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

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

[8] "免责声明 关于 的声明使用 的使用 帐户 键，" https://github.com/Azure/azure-storage-android/commit/d90c3a49312e77c2cc911c8f55a37be9947454e4.

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

[10] "管理Firebase中的用户，" https://firebase.google.com/docs/auth/android/manage-users。

[11] "移动后端作为一种服务，" https://en.wikipedia.org/wiki/Mobile_backend_as_a_service。

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

[13] "在安卓上读写数据，" https://firebase.google.com/docs/database/android/read-and-write#updating_or_deleting_data。

[14] "Scrapy | 一个快速而强大的刮擦和网络爬行框架，" https://scrapy.org/。

[15] "缩减你的代码和资源，" https://developer.android.com/studio/build/shrink-code.html。

[16] "Soot--分析和改造java和android应用程序的框架，" http://sable.github.io/soot/。

[17] "在安卓上上传文件，" 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] "什么是接口？" https://docs.oracle.com/javase/tutorial/java/concepts/interface.html。

[20] "统计门户网站。移动应用的使用，" https://www.statista.com/topics/1002/mobile-app-usage/，2017年12月。

[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年，第1248-1259页。

[22] M.Attariyan和J. Flinn，"利用动态信息流分析自动进行配置故障排除"，在*第九届USENIX操作系统设计与实施会议论文集*，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年，第356-367页。

[24] G. Balakrishnan和T. Reps，"分析x86可执行文件中的内存访问"，载于《*编译器结构*》。Springer，2004，第2732-2733页。

[25] L.Bauer, S. Garriss, and M. K. Reiter, "Detecting and resolving policy misconfigurations in access control systems, " *ACM Trans.Inf.Syst.Secur.*，第14卷，第1期，第2：1-2：28页，2011年6月。

[26] T.Das, R. Bhagwan, and P. Naldurg, "Baaz:一个检测访问控制错误配置的系统，"在*第19届USENIX安全会议论文集*，ser.USENIX Security'10, Washington, DC, 2010.

[27] M.Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios:检测ios应用程序中的隐私泄露，"在*NDSS*，2011年。

[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.

[31] L.Keller, P. Upadhyaya, and G. Candea, "Conferr:用于评估对人类配置错误的复原力的工具"，载于*Dependable Systems and Networks With FTCS and DCC, 2008。DSN 2008。IEEE国际会议*。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被另一个Amazon S3泄漏击中"，https://www. infosecurity-magazine.com/news/verizon-hit-by-another-amazon-s3/，2017年9月。

[34] M.OLSON，"2017年值得关注的云计算趋势"，https://apiumhub.com/tech-log-barcelona/cloud-computing/，2017年4月。

[35] M.C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, "通过故障遗忘计算增强服务器的可用性和安全性。"在*OSDI*, 第四卷, 2004, 第21-21页。

[36] T. 春天。 "不安全的 后台 数据库 被指责为泄漏的原因43tb 的 应用程序 数据，" https://threatpost.com/ insecure-backend-databases-blamed-for-leaking-43tb-app-data/ 126021/, 2017年6月。

[37] M.Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*.IEEE出版社，1981年，第439-449页。

[38] J.Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "自动诊断和响应内存损坏漏洞"，在*第12届ACM计算机和通信安全会议上*。ACM，2005，第223-234页。

[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 implementation 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, "火人：火力全开"。Chuah, and P. Mohapatra, "Fireman: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:一个用于安卓应用重新包装检测的自动化工具，"在*第25届国际程序编译会议论文集*，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:To- wards 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:自动生成针对漏洞的补丁，防止安卓应用中的组件劫持。"在*NDSS*，2014年。

[45] W.Zhou, Y. Zhou, X. Jiang, and P. Ning, "检测第三方安卓市场中重新包装的智能手机应用", in *ACM*

云服务	API	定义	字符串的索引我们感兴趣的参数
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

表九:亚马逊AWS的特定目标mBaaS云API

数据和应用安全与隐私会议。ACM，2012，第317-326页。

[46] C.Zuo和Z. Lin, "Exposing server urls of mobile apps with selective symbolic execution," in *Proceedings of 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.

附录

在表三中，我们无法报告两组API的具体定义，由于篇幅所限，我们只用1*和2*来表示它们。这两组API的具体定义在表九中描述。我们可以看到，1*中有7个API，2*中有17个API。我们对所有的第一个参数感兴趣，即表五中报告的`bucketName`。