Understanding data leakage in cloud from Mobile Apps

Devin Sunil Lilaramani‡
School of Computing
Clemson University
Clemson, South Carolina
dlilara@g.clemson.edu

Shriya Reddy Surusani‡
School of Computing
Clemson University
Clemson, South Carolina
shriyas@g.clemson.edu

Anoop Kakkireni‡
School of Computing
Clemson University
Clemson, South Carolina
akakkir@g.clemson.edu

***Abstract*-Over the past few years, there have been many instances of data breaches in the cloud exposing millions of user data. This paper focuses on understanding why data leakage occurs in the cloud and aims to develop suitable tools to detect vulnerabilities in mobile applications with respect to data leakage. To understand why data leakages occur, it is essential to understand the backend cloud services adopted by mobile applications. In short the backend services are termed as Mobile Backend as a Servie (mBaaS). There are two major identity and access management principles involved in data leakage – authentication and authorization. Authentication is a way for the backend service to identify the user whereas authorization defines the level of access that the authorized user has. In this paper, we explore mobile applications hosted on mainstream cloud platforms such as Amazon AWS, Google Firebase and Microsoft Azure. We identify the root causes of data leakage to be the misuse of various keys in the authentication process and the misconfiguration of user permissions in the authorization rules. We analyze LeakScope which is a tool to automatically identify vulnerabilities in mobile applications. The tool has three major processes involved – Cloud API Identification, String Value Analysis and Vulnerability Identification. The Cloud API Identifcation step takes the .apk files as the input and returns the Cloud APIs which use various keys. The final step was to determine the data leakage potential in the mobile apps without actually causing a data breach by invoking the Cloud APIs using the detect keys to request for non-existent data. The response of this request was used to categorize the keys in use as either a 'root key' or an 'app key'. If the response was 'Unauthorized operation', it means that the key used was an app key and the response 'Data does not exist' indicates the use of the 'root key'. We validated the usefulness of the tool LeakScope by analyzing 15 applications which use different cloud providers. Over 5 applications were found to be vulnerable after being detected by tool LeakScope and manually verified. We cross verified the effectiveness of the tool by manually calling API endpoints to check and make sense of the response.**

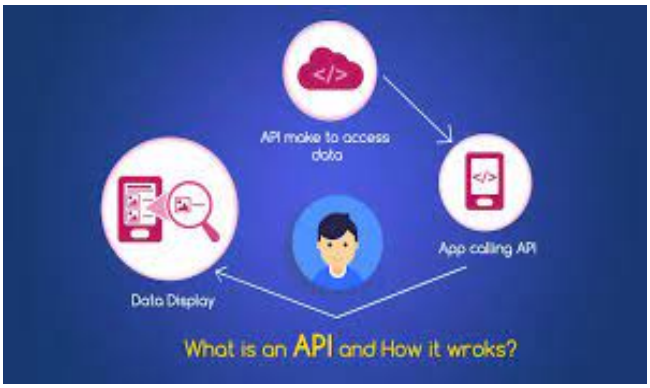
I. Introduction

With organizations moving their infrastructure to the cloud, there comes in a major question in to play, What about data security? A cloud leak is when sensitive business data stored in a private cloud instance is accidentally exposed to the internet.[20] The cloud is part of the internet. The difference is that "the cloud" offers pockets of privatized space that can be used to carry out enterprise scale IT operations.[17] Our project focuses on a tool called "Leak Scope" which uses the process of reverse engineering to find vulnerabilities in applications that use cloud APIs to develop the mobile apps.

II. Background

What is an API? API, Application Programming Interfaces, is a software to software interface that allows different applications to talk to each other exchange information or functionality.[21] This allows a business to access another business's data, piece of code, software or services in order to extend the functionality of their own products.

For example: Microservices communicate with each other using an API.



An API Functions using a Request Verb:

- **GET:** Used to retrieve a resource.
- **POST:** Used to create a new resource.
- **PUT:** Used to edit or update an existing resource.
- **DELETE:** Used to delete a resource.

Fro example, let's say you use a fuel stations API want to see a list of the nearest fuel stations

in Clemson, SC, you would then have to make a GET request that will look something like this.

Standard GET Response:

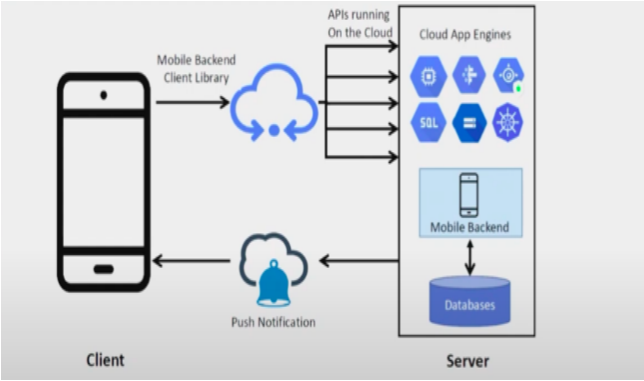
GET https://developer.nrel.gov/api/alt-fuel-stations/v1/nearest.json?api_key=XXXXXXXXX&location=Clemson+SC

IF exists : HTTP Response 200(OK)

IF Doesn't exist: HTTP Response 404(Not Found)

This tells the server to search the database to find a list of alternative fuel stations in Clemson. If that list exists, the server will send back a copy of that response in XML or JSON an HTTP response code of 200(OK) if that list doesn't exist, then it will send back the HTTP response code 404(NOT FOUND)

Widespread Adoption of Cloud APIs:



Before the advent of cloud technologies, the entire mobile application infrastructure was built from scratch which comprised of the front-end and back-end servers.[21] Even after releasing the app, ensuring the availability and scalability of the application to accomodate varying user traffic was a manual task. Cloud service providers then started providing Mobile Back-end as a Service (mBaaS) to automate this process. With it came the cloud APIs to interact with the back-end.[25] Different types of keys are now used to identify the source, authenticate the request and check access level of the source of the API calls. Most cloud providers make use of two different keys namely:

Root key: It is the master key which essentially has permissions to access all the back-end resources. It is advisable to keep this key a secret and only resevre it for use by the system adiministrators.[30]

App key: These keys have limited permissions and it is a way for the cloud to identify the source of the API calls and only provide the information that is publicly available for that particular app.

With the introduction of cloud APIs, althougth the work of a developer has been minimized, so has the room for error. The developers are responsible for the proper authentication and authorizations to prevent security breaches.

Here are some of the biggest cloud data leaks in recent history

- Verizon Partner Nice Systems Exposes 6 Million Customer Records to Internet : This information, leaked from an exposed Amazon S3 bucket, included customer details such as name, address, and phone number, as well as some account PINs, used to verify identity with Verizon.
- RNC Vendor Data Root Analytics Exposes 198 Million Voter Records : One of the largest leaks of all time was discovered when an exposed cloud system was found, containing both collected and modeled voter data from Data Root Analytics, a firm contracted by the RNC for data driven political strategy. Over 198 million unique individuals were represented in the data set, with personal details, voter information, and modeled attributes including probable race and religion.
- Accenture: In its Cyber Risk survey, the world's first Cyber Resilience startup UpGuard discovered that Accenture left at least four AWS S3 storage buckets unsecured in 2017.
  The breach included unbridled authentication details, confidential API data, digital certificates, decryption keys, user data, and meta info.
  The security analysis by UpGuard discovered 137GB of data was available for public access. As a result, cyber attackers used this data to defame and extort money from users. Some compromised information also found its way onto the dark web.

III. Problem

As data flows through the APIs security is of utmost importance to prevent data leakage.[31] Also, since APIs are like doors into an application, they're the obvious entry point for attackers who want to break into the system.

Field-Level Access: Field Level access control answers to question, "What should be visible to whom?" As part of the API design, the developer has to decide how to handle requests to resources that aren't allowed. This is the crux of the problem, since many cloud developers do not prepare for this which gives rise to unauthorized access. While we check the authorization on a resource level, we also need to check for it on the field level.[15] Hence, we not only need to check for HTTP status codes or other shallow response information, we also need to inspect the JSON structure of the response.

There are two major problems that are associated with the use of mBaaS from a data leakage standpoint. The first problem arises from the misuse of keys in authenticaing an API call. Developers tend to overlook the differences between an app key and a root key and this improper assignment leads to unwanted data leaks. This issue can also arise from improper guidance from the cloud providers. The second problem is associated with the misconfigurations of user permissions. Authentication only identifies who the user is whereas authroization decides what resources the user has to. Since there are multiple access levels defined by the developers, impropoer authorizations are highly susceptible to attackers and incrase the chances of data leakage.

Relationship between Mobile apps and their backend: Today, the leading mBaas cloud providers, to name a few are Amazon AWS, Microsoft Azure and Google Firebase. Data leakage is essentially an access control problem

which regulates which user can access which resource. In particular, access control consists of two procedures here, i) Authentication: which verifies the identity of the user, typically by username and password and ii) Authorization: this defined the relationship between the users and the resources. In Cloud backend, there are two types of users, the normal customers and the app developers. Here, customer A and Customer B are granted permission based on the credentials(App key)provided by the user after being authenticated by the cloud backend. The Developer/Administrator needs to provide a root key to access the cloud resources as seen in the image below.

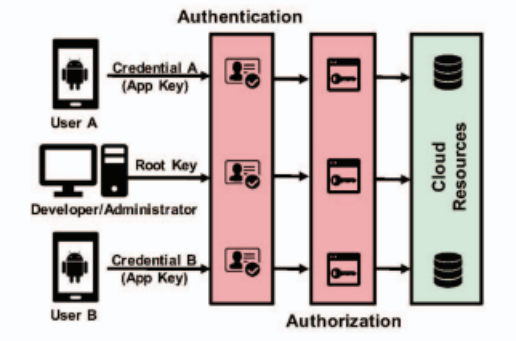


fig 1. User Authentication Authorization

Misuse of Root keys: The below image depicts the App key and the Root keys being displayed in the APK of an application hosted on Microsoft Azure, these keys can be misused to access sensitive information or even lead to information theft.

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=*

IV. Related Work

We have studied various papers that align with our line of research. TaintDroid [5], PiOS [6], and AndroidLeaks[7] focused on privacy leakage such as GPS coordinates and address book. CHEX [8] and Harehunter [9] have been developed to identify component and code hijacking vulnerabilities. ConfErr [10], ConfAid [11] and SPEX[12] are configuration testing tools to reveal misconfigurations and improper authorizations. We intened to take inspiration from most of these studies to improve the tool "Leak Scope" and to better understand the vulnerbilities in mobile applications hosted on the cloud.

V. Our Plan

From the paper that we are reproducing [1], we see that the tool developed by the authors is used to scan the mobile apps statically though automation in such a way that the results of the following are directly achieved:

- Identifying Cloud API's used by the app. Along with the obfuscated API's
- Returning the key used in the mobile app
- Sending requests with the key to an invalid ID in the cloud service used by the app, with which they determine what type of key is the app hosting.

Leakscope: Leakscope consists of 3 major components, i) Cloud API identification that takes cloud APKs and cloud APIs as input to identify cloud APIs that are used in the mobile app. ii) String Value Analysis: this tries to get the content of the key. iii) Vulnerability Identification : This stage uses the keys present in the APKs to detect vulnerabilities.

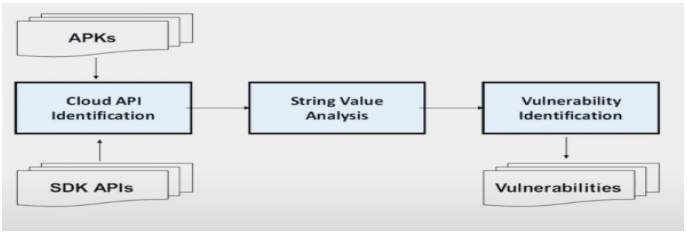


fig 2. Leakscope Architecture

We aim to use the tool and reproduce our own experimental results, with a much lesser dataset than what has been used in the original paper[1]. Our focus would be on evaluating the working of the tool along with checking for any possibility of alternate approaches to perform each step that the tool is intended for. How we plan on moving forward with this is as follows:

- Finding a small dataset (few mobile apps) that use Azure, Microsoft or firebase along with the ones which do not use any cloud service for the backend.
- Execute the first part of the tool to find the API's in our dataset and to check for any false positives or false negatives.
- The second part of the tool that uses static analysis to identify the keys in the apps and retrieves it. We plan to manually do the dynamic analysis on the apps to find the key manually and compare if this approach has a better chance on finding the key than the static analysis.
- Next we can move on checking if the key is root key or app key by using their python script to Check if it's a trial and error hit.

VI. Timeline

Checkpoint 1: First we gather the requirements to execute the code followed by understanding the underlying implementation of it. Then, collect the sample set of Mobile apps using AWS, Azure, Firebase or any other cloud services for backend along with the ones which do not use any cloud services.

Checkpoint 2: Checking which cloud Mbaas service the app is using, with the help of API's that are retrieved from the apps. Followed by getting the key from the app manually, dynamically [2].

Checkpoint 3: Finally, Check if the key is root key or app key [4] and evaluate how effective the tool is regarding this and check if this key from the mobile app could causes the data leakage in cloud. Then, we evaluate how effectively the authorization rules are written for them [3]. Followed by writing the final paper.

VII. Methodology

Initially, we categorized the list of requirements needed to get our experimental results to proceed with our evaluation of the test results presented by the authors. One of our most extensive and time-consuming processes was finding the list of android applications that use the cloud for their baas. This was particularly difficult because we did not have the hardware resources that the authors mentioned for collecting massive data and also because the source code of these apps is not open-sourced though they are publicly available for the users to interact with. For this reason, we decided on collecting a small number of applications that would serve as our data on which the code can be used. We first crawled through the web for the names of some widely used android applications that were recently deployed for public use. In particular, we crawled through the play store, which fetched some significant findings. For doing this, we used the tool called scrapy.

Once we found a couple of the latest, widely used android applications hosted for free in the play store, our task was to get the .apk files for those applications. Since the tools code that we intend to test scans the applications code statically, we did not need to install these android applications found. Our usage of scrapy shell for fetching the .apks is found below:

```
(base) PS C:\Users\sshri> scrapy shell
2022-03-13 15:06:05 [scrapy.utils.log] INFO: Scrapy 2.6.1 started (bot: scrapybot)
2022-03-13 15:06:05 [scrapy.utils.log] INFO: Versions: lxml 4.6.3.0, libxml2 2.9.10, cssselect 1.1.0, parsel 1.6.0, w3
b 1.22.0, Twisted 22.2.0, Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)], pyOpenSSL 20.0.1 (
penSSL 1.1.1k  25 Mar 2021), cryptography 3.4.7, Platform Windows-10-10.0.22000-SP0
2022-03-13 15:06:05 [scrapy.crawler] INFO: Overridden settings:
{'DUPEFILTER_CLASS': 'scrapy.dupefilters.BaseDupeFilter',
 'LOGSTATS_INTERVAL': 0}
2022-03-13 15:06:05 [scrapy.utils.log] DEBUG: Using reactor: twisted.internet.selectreactor.SelectReactor
2022-03-13 15:06:05 [scrapy.extensions.telnet] INFO: Telnet Password: 68d6410e50cd3866
2022-03-13 15:06:05 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.corestats.CoreStats',
 'scrapy.extensions.telnet.TelnetConsole']
```

```
In [2]: fetch("com.cardinalblue.piccollage.google_66410_apps.evozi.com.apk")
2022-03-13 15:44:48 [scrapy.core.engine] DEBUG: Crawled (200) <GET file:///C:/Users/sshri/com.cardinalblue.piccollage.go
ogle_66410_apps.evozi.com.apk> (referer: None)

In [3]:
```

Now that we fetched our .apks, we went ahead and ran the tool presented by the authors against these .apks after installing all the requirements of the tool and noted the applications that had vulnerable code that could be exploited to cause data leakage. The tool also detected which application uses cloud service as BaaS.:

- SHA1: 175bfcb0e03ba4907ae400e31836c926df90b813
- SHA1: 6faa126aa8588b4f1b674e62ef51b119999a9f2f
- SHA1: 5a09a2eae7f98d51e0c2b139e7e1d8bc62845275

The apps mentioned above are the three types of apps that we collected among 12 others that use firebase, AWS, and Azure as BaaS. Further details regarding the vulnerable code found and how it can be used to extract the key required to gain privileges to the database are explained in detail in the latter part of this report.

We wanted to validate that the tool does this detection and check for any false positives if present. Our intention to validate these scanned results where because the authors claimed to have no false positive detection by the tool presented, and in order to confirm that, we reverse engineered all the .apk files, and the screenshots of the vulnerable code found for three of the apps among others is presented below:

```
/* renamed from: d */
private static int m15766d() {
    SecureRandom secureRandom = new SecureRandom();
    byte[] bArr = new byte[4];
    byte b = 0;
    while (b == 0) {
        secureRandom.nextBytes(bArr);
        b = ((bArr[0] & Byte.MAX_VALUE) << Ascii.CAN) | ((bArr[1] & 255) << Ascii.DLE) | ((bArr[2] & 255) << 8) | (bArr[3] & 255);
    }
    return b;
}

public static KeysetManager withEmptyKeyset() {
    return new KeysetManager(Keyset.newBuilder());
}

public static KeysetManager withKeysetHandle(KeysetHandle keysetHandle) {
    return new KeysetManager((Keyset.Builder) keysetHandle.mo50333f().toBuilder());
}

@Deprecated
public synchronized KeysetManager add(KeyTemplate keyTemplate) throws GeneralSecurityException {
    addNewKey(keyTemplate, false);
    return this;
}
```

```
KeyProvider23() {
}

public synchronized Key retrieveKey(String str) throws KeyNotFoundException {
    Key key;
    try {
        KeyStore instance = KeyStore.getInstance(ANDROID_KEY_STORE_NAME);
        instance.load((KeyStore.LoadStoreParameter) null);
        if (instance.containsAlias(str)) {
            Log log = logger;
            log.debug("AndroidKeyStore contains keyAlias " + str);
            logger.debug("Loading the encryption key from Android KeyStore.");
            key = instance.getKey(str, (char[]) null);
            if (key == null) {
                throw new KeyNotFoundException("Key is null even though the keyAlias: " + str + " is present in " + ANDROID_KEY_STORE_
            }
        } else {
            throw new KeyNotFoundException("AndroidKeyStore does not contain the keyAlias: " + str);
        }
    } catch (Exception e) {
        throw new KeyNotFoundException("Error occurred while accessing AndroidKeyStore to retrieve the key for keyAlias: " + str, e);
    }
    return key;
}

public synchronized Key generateKey(String str) throws KeyNotGeneratedException {
```

```
/* renamed from: a */
private synchronized boolean m15763a(int i) {
    for (Keyset.Key keyId : this.f20581a.getKeyList()) {
        if (keyId.getKeyId() == i) {
            return true;
        }
    }
    return false;
}

/* renamed from: b */
private synchronized Keyset.Key m15764b(KeyTemplate keyTemplate) throws GeneralSecurityException {
    KeyData newKeyData;
    int c;
    OutputPrefixType outputPrefixType;
    newKeyData = Registry.newKeyData(keyTemplate);
    c = m15765c();
    outputPrefixType = keyTemplate.getOutputPrefixType();
    if (outputPrefixType == OutputPrefixType.UNKNOWN_PREFIX) {
        outputPrefixType = OutputPrefixType.TINK;
    }
    return (Keyset.Key) Keyset.Key.newBuilder().setKeyData(newKeyData).setKeyId(c).setStatus(KeyStatusType.ENABLED).setOutputPrefixType(ou
}
```

By going through these vulnerable codes found along with the APIs used by these apps, we confirmed that these applications were rightly detected by the tool and the tool had no false positives as per our findings.

Now that the Cloud APIs were detected, our next step

was to extract the string values of the various keys from all the applications which was one of the major step to further evaluate the key. The tool presented by the authors had a segment of the code that performed this operation. It took us a while to find the specific code regarding this operation, since the whole code of the tool had no documentation explaining it's flow and there was no way of knowing how to feed the 15 applications that we collected to the tool as input for extracting the key strings. We ran the code multiple times and re-framed it with proper comments. This had to be done at this stage of the project because the initial way of running the tool against the apps only detected the cloud providers APIs and those apps that had the vulnerable code in them. If the application had app key or root key hidden in the back-end code of the app, the app was flagged vulnerable but it was truly vulnerable only if the developer used the root key in the app. This true vulnerability could only be confirmed after the key was extracted and checked which type it belonged to and can say that the application is truly vulnerable to causing data leakage in cloud only if the app contained the root key in it.

After understanding the flow of the code, we figured that the code for extracting the key strings was stored in "api_key_detector" folder and that we had to place our applications in that folder such that the tool can access and scan them for it. This was because the authors has hard-coded the location of the input to be it's immediate directory.The directory and all the apps are shown below:

```
C:\Users\sshri\Desktop\Tool>cd api_key_detector

C:\Users\sshri\Desktop\Tool\api_key_detector>ls
Boilr.apk            Gardine.apk      LeafPic.apk    Paseo.apk          Trekarta.apk          WaniDoku.apk
Calendula.apk        Gen_authtoken.py  Metronome.apk  PoetAssistant.apk  TriPeaks.apk          Yokatta.apk
FediPhoto-Lineage.apk  JitsiMeet.apk    Nonocross.apk  SimplySolid.apk    ValueSetAnalysis.java

C:\Users\sshri\Desktop\Tool\api_key_detector>
```

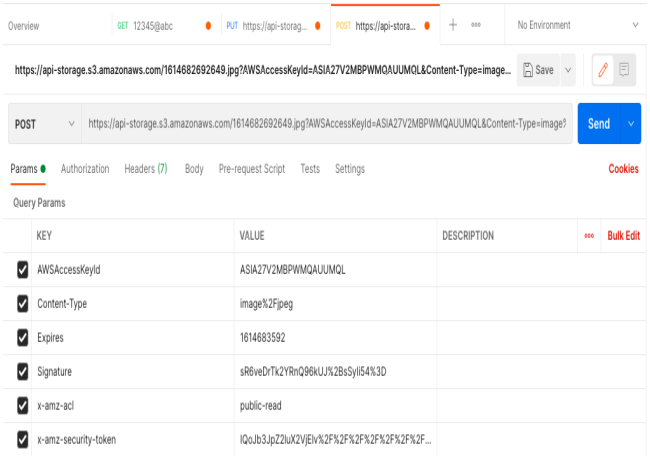
After placing the applications in the required folder and running the tool. All the keys were extracted from the input apps. The same is shown below:

```
C:\Users\sshri\Desktop\Tool\api_key_detector>javac ValueSetAnalysis.java

C:\Users\sshri\Desktop\Tool\api_key_detector>java ValueSetAnalysis
0.06817095,0.00014711,0.00681442,ISSN8yYzHk84Y6PfMd_1BvhNCBzgfQxcSP5bx--ZMil0Qmkh/VU2bBUpVulVyxgwh5nY5s
0.06817095,0.00014711,0.00681442,c487ba1b6d536aec1f3e48bffbe3532c1f78b1a2
0.06817095,0.00014711,0.00681442,e1d0c9ce505b1822d97a34ade0107d241d82ec74fd522e4963e602c
0.06817095,0.00014711,0.00681442,oxBmfms2SrMeEB5iaQUFR7VCn1YFxNY3jLHPGYXMd7fe81e4f62df3e
0.06817095,0.00014711,0.00681442,AuhXCurI1rioe8xbkcHpXnZQ3D993ibcHXaCJsc56_7OSm1vTYgeXHOuo/BbXz-tnC1C85
0.06817095,0.00014711,0.00681442,075b4a0782c168c8c197691365e8a6c65ed13d410fd10a68a2068ba
0.06817095,0.00014711,0.00681442,NHzD9JoV24aI5sDmZo/Kbex7Fa6d8eJz7DBSRI0g
0.06817095,0.00014711,0.00681442,dfdf3928dd6d77c62d290162ccdb61bbfda041ddfb6d70bed45eef6
0.06817095,0.00014711,0.00681442,0ad60a37fb035d1d9c1c0094c1a558848a156cb167c33a36d6e2cff
0.06817095,0.00014711,0.00681442,0cdcUO7Pn0OFHAHuDdZcwQDfAHNnARUT7CRIWJ9m
0.06817095,0.00014711,0.00681442,eyJ0eXAi0iJKV1QiLCOhbGciliJIUzI1NiJ9.eyJlbwFpbCI6ImFkb2
0.06817095,0.00014711,0.00681442,zS/3eq7qguYNaNw83NmfY7fIcCkqw_kPwOn8lZVybaxNK_VsIh7Q7Xw93cZw4GLAUBJs4l
0.06817095,0.00014711,0.00681442,KFsukQJ8FpgQu9u8qiga0qqDDydNh5K5o9vzQM6JpJwvBaD7mPgp7RW5ZLZQer6GDr_QtQ
0.06817095,0.00014711,0.00681442,YfPlvOe6pzN9h3Ceq0sO23/dra747RxXP0xyNcJG
0.06817095,0.00014711,0.00681442,Led8vxbES1fs1Lv_0jY10-y1lsCi07W0sutuHdOJgUbcVbur/Bl0aMZ-w0vXSikmQwMKfF

C:\Users\sshri\Desktop\Tool\api_key_detector>
```

The three different cloud providers in consideration have different formats for the keys they employ. The format for the root key and app is also quite distinguishable as in the case of Amazon AWS. We browsed through various official API documentations to understand the formats of the keys and to try and differentiate them directly after the extraction. However, this was not very successful considering the limited number of apps that we had selected for our experiment.

The next step was to check for vulnerabilities in the apps. There are two kinds of vulnerabilities that were analyzed: key misuse and permission misconfiguration. In case of key misuse, the use of root keys is a direct indication of vulnerability as it is supposed to be kept secret and used in the everyday flow of things. For permission misconfigurations, we tried to request the server for certain information using endpoints of relevance and passing the extracted keys with them. Based on the response from the server, we could decide if the app was vulnerable or not. In an ideal scenario, if we want to check the details of an up and running EC2 instance on Amazon AWS, the request made using the app key must be denied. This is showcased in the figures below. This proves the fact that the access key used in this case was indeed an app key. We also tried to make certain GET calls to the Amazon simple DB using the keys that were extracted. As mentioned before, the access should be denied unless and until the root key is being used for the query. We received a response from the server for one of the keys that were extracted, and hence it was concluded to be a root key.

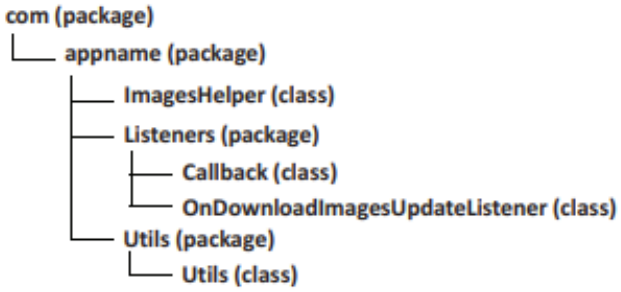
The evaluation result of each app if it was truly vulnerable or not is presented in the evaluation section of the report. After all our findings we confirm that the tool rightly detects the cloud provider of each app and extracts the keys as expect from the apps. The minor flaw in the tool is that it also flags the app as vulnerable if it detects the presence of key in the back-end code. This is particularly flawed because the app is truly vulnerable only if it contains the root key. One other observed minor flaw was that the tool's flow of code has no proper documentation or comments to understand each functions working. But apart from these, the tool's functionality works as expected.



VIII. Vulnerability

After we tested the tool's legitimacy for a few applications by cross-checking the output of the code and the manually

reverse engineered .apk files, we moved on to study the working of the tool and its behavior when tested on vulnerable apps. The first step was to understand how the tool identifies the cloud API. The algorithm used in the tool LeakScope uses in-variants in a function (or method) in the .apk file to detect the Cloud APIs in use. The algorithm disregards obfuscated cloud APIs as the invariants are preserved even if obfuscation techniques were used.



The first invariant is the hierarchical structure or the pattern observed for the different packages and types. It was observed that each class or method has a unique type depending on the function of the API. Although the types can be obfuscated, these classes' structure remains the same. The second invariant that LeakScope takes advantage of is the caller-callee relation. All the methods call certain functions, such as StringBuilder, which appends various parameters to the endpoint based on certain conditions. These parameter values can be recursively stored and clubbed together to build the endpoint of concern, referred to as the caller's signature. We analyzed the internal functioning of this algorithm with a test .apk file. The code used to build and detect the caller signatures is shown below.

```
package com.example.vsa.valuesetanalysisexample;

import ...
public class VsaTest {

    String keypart1;
    String keypart2;

    public void init(Context arg3){
        keypart1 = getHardcodedStr();
        keypart2 = arg3.getResources().getString(R.string.key_part2);
    }

    public String getHardcodedStr(){
        return "hardcode";
    }

    public CloudStorageAccount getAccount() throws URISyntaxException, InvalidKeyException {
        String key = "part1:";
        key += keypart1;
        key += "|part2:";
        key += keypart2;

        //target function
        return CloudStorageAccount.parse(key);
    }
}
```

The sample output when LeakScope is run against this .apk file is shown below. We can see that the target endpoint of concern is a known endpoint used in Microsoft Azure-based apps. We can see how the target endpoint was identified and how the signatures were built. Each method has multiple dependencies, and the string is recursively built to form the final signature. A custom stack, a last-in-first-out data structure, is maintained by LeakScope to keep track of the order of the string computations to generate the final string values.

```
$ java -jar ValueSetAnalysis.jar ./libs/android.jar ./example/example.json
May 20, 2019 8:53:14 PM brut.androlib.res.AndrolibResources loadMainPkg
INFO: Loading resource table...
Using './libs/android.jar' as android.jar
com.example.vsa.valuesetanalysisexample[CG time]:10035
com.example.vsa.valuesetanalysisexample[CG time]:10774
...
com.example.vsa.valuesetanalysisexample==========87519500==========
Class: com.example.vsa.valuesetanalysisexample.VsaTest
Method: <com.example.vsa.valuesetanalysisexample.VsaTest: com.microsoft.azure.storage.CloudStorageAccount get
Target: $r3 = staticinvoke <com.microsoft.azure.storage.CloudStorageAccount: com.microsoft.azure.storage.Clou
Solved: true
Depend: 422751563, 1674086835,
BackwardContexts:
  0
    $r1 = new java.lang.StringBuilder
    specialinvoke $r1.<java.lang.StringBuilder: void <init>()>()
    virtualinvoke $r1.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("part1:")
    $r2 = $r0.<com.example.vsa.valuesetanalysisexample.VsaTest: java.lang.String keypart1>
    virtualinvoke $r1.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($r2)
    $r2 = virtualinvoke $r1.<java.lang.StringBuilder: java.lang.String toString()>()
    $r1 = new java.lang.StringBuilder
    specialinvoke $r1.<java.lang.StringBuilder: void <init>()>()
    virtualinvoke $r1.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($r2)
    virtualinvoke $r1.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("|part2:")
    $r2 = virtualinvoke $r1.<java.lang.StringBuilder: java.lang.String toString()>()
    $r1 = new java.lang.StringBuilder
    specialinvoke $r1.<java.lang.StringBuilder: void <init>()>()
    virtualinvoke $r1.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($r2)
    $r2 = $r0.<com.example.vsa.valuesetanalysisexample.VsaTest: java.lang.String keypart2>
    virtualinvoke $r1.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($r2)
    $r2 = virtualinvoke $r1.<java.lang.StringBuilder: java.lang.String toString()>()
    $r3 = staticinvoke <com.microsoft.azure.storage.CloudStorageAccount: com.microsoft.azure.storage.CloudStc
ValueSet:
   |0:part1:hardcode|part2:fromres,

com.example.vsa.valuesetanalysisexample==========422751563==========
Field: <com.example.vsa.valuesetanalysisexample.VsaTest: java.lang.String keypart2>
Solved: true
Depend: 1568161955,
ValueSet:
   |-1:fromres,

com.example.vsa.valuesetanalysisexample==========1674086835==========
Field: <com.example.vsa.valuesetanalysisexample.VsaTest: java.lang.String keypart1>
Solved: true
Depend: 38076305,
ValueSet:
   |-1:hardcode,

com.example.vsa.valuesetanalysisexample==========1568161955==========
Class: com.example.vsa.valuesetanalysisexample.VsaTest
Method: <com.example.vsa.valuesetanalysisexample.VsaTest: void init(android.content.Context)>
Target: $r0.<com.example.vsa.valuesetanalysisexample.VsaTest: java.lang.String keypart2> = $r2
Solved: true
Depend:
BackwardContexts:
  0
    $r2 = virtualinvoke $r3.<android.content.res.Resources: java.lang.String getString(int)>(2131427369)
    $r0.<com.example.vsa.valuesetanalysisexample.VsaTest: java.lang.String keypart2> = $r2
ValueSet:
   |-1:fromres,

com.example.vsa.valuesetanalysisexample[{"0":["part1:hardcode|part2:fromres"]}]
```

IX. Evaluation

We present the step by step approach for our implementation of the tool LeakScope. This section also provides the results obtained and the analysis of the vulnerabilities identified by the tool LeakScope.

Mobile App collection: Initially, we tried to look for some of the apps that were mentioned in the paper. We spent quite some time to try to discover the .apk files of those apps and which cloud service providers were being used by those apps. We slowly realized that these apps were mostly built for commercial use and the source code was not publicly available. We faced a similar problem with the identification of the cloud service providers as well. We also realized that

the authors made the ethically right decision of informing the cloud providers about the vulnerabilities in the apps that were tested. The cloud providers in turn informed the app owners and thus all these apps were most probably patched and immune to LeakScope. After this realization, we made the decision to look for applications which were open-source and were developed by the community. This way we would get to test the apps which were never subjected to any vulnerability tests. For our experiment, we chose 15 apps in total. This was done by web scrapping the app names of the widely used customer centred applications. The screenshots of using scarpy for downloading the apk files of them is shown in the methodology section of this paper. We Then categorized these applications based on the cloud providers that they were using which were, for the scope of our experiment, Amazon AWS, Microsoft Azure and Google Firebase. The experiment was conducted on our personal computers as LeakScope can be used like a Java library. We first accessed the GitHub repository provided in the paper to find the source code for LeakScope. We made sure the jar files were in our classpath.

Cloud API Identification: The first part of LeakScope is the API identification. We ran the tool by selecting about 6 APIs of our interest, 2 of each different cloud provider. In order the select the most optimum APIs for our experiment, we browsed though the official documentation provided by the cloud providers and looked for the most commonly used features and endpoints based on the app description. The tool generated function signatures and identified 14 apps which use the APIs of our interest.

Interestingly, the tool LeakScope accounts for apps which adopt obfuscation techniques as well using invariant in the code to generate function signatures. However when we ran the tool, we were not able to uncover if the apps were obfuscated or not. The detailed explanation of this feature was not provided by the original authors and hence we failed to spot if the app was obfuscated or not. Regardless, we did manage to get the output for almost all the apps and this step was touted to be a success.

It is important to note that the tool identifies the mBaaS clouds that are being used by the apps and not the IaaS clouds.

String Value Analysis: The second step of the tool LeakScope is the String Value Analysis. The whole purpose of this step is to pinpoint where each API is called and the arguments that include the authentication keys in the mobile app after we've identified the cloud APIs we're interested in. However, because we are using static analysis, we will not be able to directly observe their values. Hence we'll need to create a targeted string value set analysis uncover the probable key values.

Among the 15 apps that we chose, the tool computed the strings of the parameters of our interest for 14 apps. The parameters were carefully chose by referring to the official API documentation provided by the cloud service providers itself. We targeted APIs for parameters such as 'bucketName', 'accessKey', 'appKey' covering all the three cloud providers, AWS, Azure and Firebase.

It is also important to understand when and why the tool fails to compute the strings. Many of the unresolved values for these parameters were obtained via the Internet. This is especially prevalent with Firebase, because Google actively advises developers to retrieve keys from distant servers. We couldn't infer their values without a dynamic study of the apps. The second problem is that certain apps use cryptographic routines to protect the string, which static analysis cannot resolve.

Vulnerability Identification: Now that we have our keys and strings identified, we can move on to the final and decisive step of vulnerability identification. The 3rd and final component of LeakScope basically detetcs multiple types of vulnerabilitites such as Key misuse and permission misconfiguration. This is determined keeping a zero data leakage policy in mind. The tool detected multiple vulnerabilities, most of which were associated with pemission misconfiguration. To verify these results, we tried caling the API endpoints that were detected using the extracted keys in the previous step. We found some insightful results. Based on the value of the key and it's format, we could differentiate between the root key and the app key. When we called the API endpoints using these keys, it revealed the misconfigurations in permissions. We found that 1 out of the 15 extracted keys was a root key and as many as 5 apps were having vulnerabilities from a permission misconfiguration standpoint.

This section contains the true vulnerability evaluation of each app that we chose as the input for the tool. Our whole experimentation and process used to evaluate them is mentioned in the methodology part of the report.

Evaluation for the Apps using AWS: While checking for vulnerabilities in apps that make use of AWS mBaas services, we found that 2 out of 5 apps are Privacy Sensitive. One of these apps were Tripeaks, which is a classic Solitaire Puzzle game, which displays a table of the top 10 highscorers, we noticed during our evaluation using Postman that sensitive user data was being passed via one of the APIs.

Name	Cloud Service Used	App Description and Functionality	Privacy Sensitive
Yokatta	AWS	Flashcards based language learning app	✗
Simply Solid	AWS	Pick a solid color as your homescreen background color	✓
Boilr	AWS	Android app which monitors cryptocurrencies	✗
Metronome	AWS	Professional tool to help all musicians play with flawless accuracy	✗
TriPeaks	AWS	Classic solitaire puzzle game	✓

Evaluation for the Apps using Azure: While checking for vulnerabilities in apps that make use of Microsoft Azure mBaas service, we found two Apps namely Trekarta and Gardine which are Privacy Sensitive.

Name	Cloud Service Used	App Description and Functionality	Privacy Sensitive
FediPhoto-Lineage	Azure	Quickly post photos	✗
Trekarta	Azure	Designed for hiking, geocaching, off-roading, cycling, boating and all other outdoor activities	✓
LeafPic	Azure	A fluid, material-designed alternative gallery	✗
Gardine	Azure	A minimalistic one-touch app switcher	✓
Jitsi Meet	Azure	Instant video conferences efficiently adapting to your scale	✗

Evaluation for the Apps using Google Firebase: While checking for vulnerabilities in apps that make use of Google Firebase mBaas service, we found 1 app named Celendula, an app used to manage the medical presciptions of the user which we named as privacy sensitive as sensitive data was being passed via the API which can be easily visible to an attacker.

Name	Cloud Service Used	App Description and Functionality	Privacy Sensitive
Calendula	Firebase	Manage your medical prescriptions through a simple and intuitive interface	✓
WaniDoku	Firebase	Japanes leaning app	✗
Poet Assistant	Firebase	A set of offline tools to help with writing poems	✗
Nonocross	Firebase	Simple number puzzle game based around grids	✗
Paseo	Firebase	Step counting app	✗

X. Future Work

Finally, LeakScope is exclusively interested in apps that construct mobile apps using cloud APIs. Clearly, other forms of cloud services (e.g., the IaaS cloud) have been used directly in the back-end of a major fraction of the apps. Another area for future research is how to identify these apps and their vulnerable cloud services in a principled manner, like LeakScope has done with mBaaS.

Though LeakScope works as expected apart from its minor flaws which are mentioned at the end of methodology section of this report, it's working is very limited to android mobile applications using cloud as Baas. With respect to the tool, further developments can be made to documenting it's code flow and correcting the vulnerability detection part of the tool to be more specific to detecting the truly vulnerable applications without having to check the type of key the app holds manually. This could elevate the number of users of the tool, since the accessibility of it would be increased. This would in-turn encourage the industrial use of the tool for getting major development changes in the apps.

Apart from further development of the tool itself, more scope lies in adapting the flow of LeakScope to incorporate the same principles in developing a tool for ios mobile applications and web applications and these tools should be able to be deployed over the web for easy access.

XI. Team Contribution

The contribution of the team members are as follows:

Devin Sunil Lilaramani:

- Checking and installing all packages required by the tool to execute. Finding the hardware resources for our experimental set-up.
- Extracting the keys from the apps.

Shriya Reddy Surusani:

- Crawling through the web for finding the applications to feed the tool as the inputs.
- Reverse Engineering the apk files.

Anoop Kakkireni:

- Querying and finding the API end points for each app to determine the type of key.
- Debugging the issues occurred with code execution.

Every team member put in the efforts equally and helped each other in each task.

References

[1] Zuo, Chaoshun, Zhiqiang Lin, and Yinqian Zhang. "Why does your data leak? uncovering the data leakage in cloud from mobile apps." 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019.

[2] Serrão, Francisco Miguel Gouveia. "SMART: Static and Dynamic Analysis to Reverse Engineer Android Applications." (2021).

[3] Masood, Rahat, et al. "Cloud authorization: exploring techniques and approach towards effective access control framework." Frontiers of Computer Science 9.2 (2015): 297-321.

[4] Wang, Hui, et al. "Vulnerability assessment of oauth implementations in android applications." Proceedings of the 31st annual computer

security applications conference. 2015.
[5] 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
[6] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in NDSS, 2011.
[7] 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.
[8] 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.
[9] 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.
[10] 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.
[11] 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.
[12] 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.
[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 failureoblivious 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