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

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. 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. 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

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. 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. 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.
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

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.

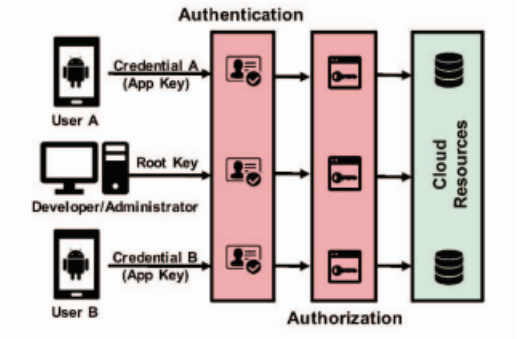


fig 1. User Authentication Authorization

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.

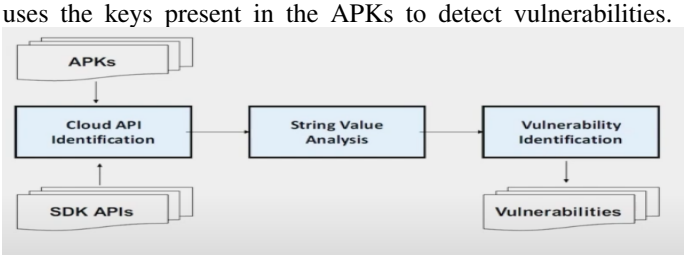


fig 2. Leakscope Architecture

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. Progress

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, w
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.g
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 and noted that the following applications 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 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 these three .apk files, and the screenshots of the vulnerable code found for the same 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;
}
```

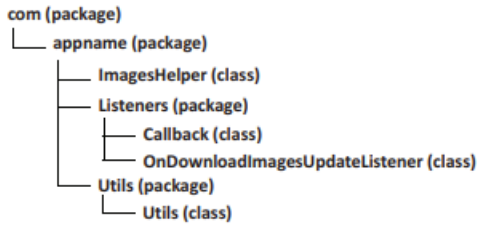
```
/* renamed from: a */
private synchronized boolean m15763a(int i) {
    for (Keyset.Key keyId : this.f20581a.getKeyList()) {
        if (keyId.getKeyId() == i) {
            return true;
        }
    }
    return false;
}

/* 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.

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==========875195900==========
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. Next steps

Once the Cloud APIs are detected, the string values of the various keys need to be extracted. These keys need to segregate into either a 'root key' or an 'app key.' The authors needed to design an approach to detect data leakage without actually causing a data leak themselves to prevent private data leakage. This step involves invoking a non-existent endpoint by choosing a faulty 'instance id' and observing the response from the backend server. If the root key was used to send the request to the server, the expected response is 'InvalidInstanceID'. If the app key was used to send the request, 'UnauthorizedOperation' response is observed. The detected keys can be labeled either a root key or an app key. The use of the root key in Cloud APIs reveals the potential for data leakage.

Our next steps involve extracting the keys, verifying the functionality of the tool LeakScope against random applications, and checking if they are vulnerable or not based on the output of the code.

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.