Credit Karma: Understanding Security Implications of Exposed Cloud Services through Automated Capability Inference

Xueqiang Wang[1][*], Yuqiong Sun[2][*][†], Susanta Nanda[3], and XiaoFeng Wang[4]

[1]University of Central Florida, *xueqiang.wang.01@gmail.com*
[2]Meta, *yuqiong@fb.com*
[3]ServiceNow, *susanta@gmail.com*
[4]Indiana University Bloomington, *xw7@indiana.edu*

Abstract

The increasing popularity of mobile applications (apps) has led to a rapid increase in demand for backend services, such as notifications, data storage, authentication, etc., hosted in cloud platforms. This has induced the attackers to consistently target such cloud services, resulting in a rise in data security incidents. In this paper, we focus on one of the main reasons why cloud services become increasingly vulnerable: (over-)privileges in cloud credentials. We propose a systematic approach to recover cloud credentials from apps, infer their capabilities in cloud, and verify if the capabilities exceed the legitimate needs of the apps. We further look into the security implications of the leaked capabilities, demonstrating how seemingly benevolent, unprivileged capabilities, when combined, can lead to unexpected, severe security problems. A large-scale study of ∼1.3 million apps over two types of cloud services, notification and storage, on three popular cloud platforms, AWS, Azure, and Alibaba Cloud, shows that ∼27.3% of apps that use cloud services expose over-privileged cloud credentials. Moreover, a majority of over-privileged cloud credentials (∼64.8%) potentially lead to data attacks. During the study, we also uncover new types of attacks enabled by regular cloud credentials, such as spear-phishing through push notification and targeted user data pollution. We have made responsible disclosures to both app vendors and cloud providers and start seeing the impact—over 300 app vendors already fixed the problems.

1 Introduction

The ever increasing popularity of mobile applications (apps) has led to a rise in demand for scalable backend services, such as user authentication, push notification, data storage, and so on. Modern cloud platforms have become the natural home for such backend services due to their high availability, scalability, and low cost. This change in paradigm has also prompted the attackers to shift their focus to cloud services. The overwhelmingly huge amount of data generated by mobile app users, the value of the data, combined with the open nature of cloud services have kept attackers' interest level alarmingly high. As a result, incidents such as leaks and theft of users' personally identifiable information (PII) ([55, 60, 68]) and corporate/government secrets ([41, 59]) are on the rise.

Implications of cloud credential exposure. Cloud service providers have often gone extra miles to secure their services, but the clients of those services (e.g., the mobile apps) are often seen as the weaker links. A number of previous studies ([29, 67, 71, 74]) have reported identification of residual cloud credentials that may lead to insecure cloud access. Particularly, Zuo et al. [75] have conducted a systematic study on cloud service credentials embedded in mobile apps, and concluded that the misuse of root credentials is the major reason behind the data leakage problems in clouds. As a result of these findings, major cloud service providers, such as Google Cloud, Amazon Web Services (AWS) and Microsoft Azure, updated their security guidelines to prevent the use of root credentials in mobile apps [13, 16]. Instead, they encourage app developers to adopt regular cloud user credentials (e.g., IAM user) when accessing cloud services in order to restrict the data access capabilities from apps. However, the effectiveness of this approach depends entirely on the app developers' ability to correctly tailor the capabilities of the regular user credentials, which, if history is any indication, is often a daunting task. In particular, cloud services have notoriously complex sets of capabilities and even minor mistakes may result in severe data leaks on app users. Even worse, such data leakage can be more subtle and harder to detect, compared with the usage of root credentials. Hence, there is a need for analyzing the capabilities owned by cloud service credentials at a *finer granularity* and better understanding the security implications of the capabilities when they are exercised by apps even though they are *regular cloud user credentials*.

Analyzing exposed cloud capabilities. In this paper, we report a systematic study on exposed cloud credentials. By answering three key questions, we aim to contribute to the better understanding of how mobile apps use cloud credentials (especially regular ones) and its security implications: first, what capabilities do mobile app developers grant to cloud credentials when accessing a cloud service? Second, do developers often grant more-than-needed capabilities (i.e., over privileged[1])? Third, when would more-than-needed

[*]The two lead authors contributed equally to this work, and are corresponding authors.

[†]Part of the work was done while the author was with NortonLifelock.

[1]To a large extent, the root credential studied in previous work [75] is a special type of over privileged credential

capabilities be harmful and how (i.e., can one non-privileged, extra capability lead to large scale attacks on clouds when combined with other seemingly benevolent capabilities)?

To answer these questions, we need to address several challenges. First, unlike the prior studies that evaluate app permissions on their host operating systems ([9, 33, 40, 56, 57, 69]), how to identify capabilities that apps own on remote cloud services remains largely under-addressed. Lack of visibility into cloud services leaves analysis of mobile apps the only viable approach, but analyzing an app alone can hardly draw a complete picture of all the capabilities the app owns on remote cloud. Second, identifying capabilities at *fine granularity* is instrumental in identifying subtle attacks involving misuse of individual capabilities. However, accurately inferring such capabilities is difficult due to the complexity in cloud access control models and the opacity of their policies. Finally, due to the large number of mobile apps available today, the analysis must be highly scalable and automated. To address the first challenge, we complemented static app analysis with a dynamic probing approach where we dynamically probe cloud services and analyze their responses. Specifically, we leverage a novel approach called `induced transaction failure` (Section 3.6) to indirectly infer capabilities on remote cloud services without risking access or modification to actual users' data. To address the second challenge, we eliminated the abstraction of specific access control models and policies of cloud services by building the list of capabilities owned by a cloud credential represented in the most basic form of access control — a tuple expressed by the combination of $< subject, object, operation >$. Combining these ideas with a suite of program analysis techniques, we built a tool named *PrivRuler* that allows us to systematically investigate cloud credentials used in mobile apps and identify the ones that may lead to user data compromises in the cloud.

Our findings. We evaluated *PrivRuler* against two popular types of mobile backend service, *storage* and *notification*, on three popular cloud platforms: AWS [2], Microsoft Azure [46], and Alibaba Cloud [1]. Our empirical study consists of 1,358,057 mobile apps crawled from Google Play [34], out of which 11,891 apps are identified to be using aforementioned mobile backend services. We found that 2,572 apps contain cloud credentials with more-than-needed capabilities (i.e., over privileged). Majority of these apps are relatively new (with 70.0% updated on Google Play after April 2019), confirming our hypothesis that simply encouraging app developers to not use cloud root credentials would not solve the problem. We further identified 1,667 apps (out of 2,572, 64.8%) whose extra capabilities can potentially lead to large scale attacks on mobile app users. Examples of such high profile attacks include spear phishing attacks where an attacker can target any or all mobile app users with a crafted in-app notification message, large scale data leakage where an attacker can harvest user PIIs, personal photos, legal recordings, travel logs and so on, and data pollution attacks where an attacker can distribute malicious content to mobile app users. In total, these apps have been downloaded over 765 million times.

Responsible disclosure and ethical research. We have made responsible disclosures to mobile app developers, cloud vendors and Google Play, and started seeing the impact of the work — 317 apps were already updated to newer versions with changes to their cloud credentials and 56 mobile app developers formally acknowledged our efforts via email (see more details online [5]).

During the course of this research, ethics has been our top priority. In particular, we conducted a thorough risk-benefit analysis of the dynamic cloud probing approach and ensure from design that we do not access or modify any users' data, nor impact the operation of cloud services. Refer to Section 3.6 and Section 6 for more details.

Contributions. To summarize, we make the following contributions:

• We build *PrivRuler*, an automated tool that combines static app analysis and dynamic probing to accurately infer cloud capabilities owned by mobile apps. We have open sourced *PrivRuler* [5].

• We conduct a large-scale, empirical study on cloud capabilities owned by mobile apps and their security implications. The findings indicate the pervasiveness and seriousness of security risks caused by excessive cloud capabilities, particularly the risks posed by combination of regular credentials.

• We propose several heuristics for identifying mobile apps that may enable large scale attacks against mobile app users. This approach reveals a number of novel attacks, such as spear phishing attacks, large scale data leakage, data pollution, etc.

2 Background

Cloud services and cloud identities. Backend cloud services are typically integrated into mobile apps in the forms of mobile SDKs, which in turn are composed of a number of cloud service APIs corresponding to a variety of cloud operations. App developers perform cloud operations (e.g., store/retrieve data) by invoking the APIs, similar to the use of other libraries in mobile apps. On the cloud side, app developers need to configure cloud identities and grant these identities capabilities to operate on the cloud resources. They will need to generate credentials for the identities from their cloud management console, and deliver them to the apps in order for the apps to be able to authenticate and authorize themselves for performing cloud operations. Since the cloud identities are owned by app developers instead of individual app users, the app users typically share the same set of cloud credentials and capabilities.

There are two types of cloud identities. One is root (i.e., account owners), who basically own full access to all cloud resources under the same cloud account. Previous study [75] has found that misuse of root cloud credential is one of the major causes for data leaks in cloud. Since then, cloud vendors have been discouraging usage of root credentials in mobile apps [13, 16]. The other type is regular user identity (e.g., IAM users [17] in AWS), for whom app developers need to

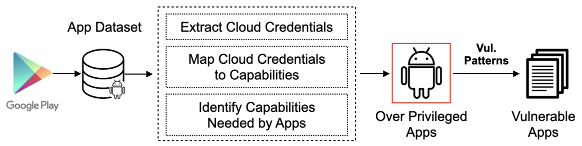

App Dataset
Google Play
Extract Cloud Credentials
Map Cloud Credentials to Capabilities
Identify Capabilities Needed by Apps
Over Privileged Apps
Vul. Patterns
Vulnerable Apps

Figure 1: Overview of *PrivRuler*

properly configure its capabilities. In this work, we focus on investigating excessive capabilities granted to regular user identities, but our approach applies to root identities as well.

Scope of the study. This research focuses on the mobile apps that use cloud platforms as their backend services, for the purpose of analyzing their cloud service usages to determine whether they expose sensitive cloud capabilities. We assume that the apps being analyzed are benign, and do not consider the evasive and malicious apps that deliberately hide cloud services in their background (e.g., via dynamic code loading). Instead, we focus on the evaluation of data attacks that can happen to benign apps and these apps' legitimate usages of cloud services to help the developers mitigate the threats to their data in the cloud.

3 Design of *PrivRuler*

Overview. Figure 1 presents an overview of *PrivRuler*. The first component is the App Dataset, which stores $\sim$1.3 million apps crawled from Google Play. *PrivRuler* scans this raw dataset to identify apps that use cloud services covered in this study. After these apps are identified, *PrivRuler* extracts cloud credentials from them. To decide whether an app is over-privileged, *PrivRuler* first determines the *capabilities that a cloud credential enables*. This is done by dynamically probing cloud services to perform different cloud operations using the credentials extracted from the app. The output is a *capability map*, represented by access control tuples. Next, our tool identifies the capabilities that are *needed by the app*. It statically analyzes the app to find cloud operations performed and cloud objects accessed, also in the form of access control tuples. Then it compares the capabilities that an app has with the capabilities that the app needs to decide whether it is *over-privileged* and what the extra capabilities are.

Over-privilege is the root cause for many security issues and should be avoided whenever possible. However, not every "extra" capability enables massive scale attacks against app users. Thus, in the last step, our tool utilizes a set of vulnerability patterns to decide when *over-privilege is likely to become a vulnerability* that an attacker may exploit to launch attacks at scale against app users. This guides prioritization of vulnerability confirmation, report, and fix. In the remainder of this section, we describe each step of our tool in detail.

3.1 Building the App Dataset

We build an app dataset by crawling Google Play [34] with a crawler using googleplay-api [51]. For this purpose, we firstly gathered the package names of all apps (3.4 million as of April 2020) with google-play-scraper [53], and randomly shuffled the package names before downloading apps. Till this work is done, we were able to download and analyze 1.3 million apps (See Section 4.1 for more details).

With the raw app dataset, we focus on the apps that use cloud services as backends. The core challenge is to detect *at scale* (i.e., millions of apps) if an app uses a cloud service. To address this challenge, we adopted a two-stage pipeline design. In the first stage, we screen the app dataset by simply testing the presence of cloud service SDKs. We unpack an app and check if certain packages used for communicating with cloud backends are present in the unpacked file structure of the app (e.g., `com/amazonaws/`). Such package names are often *unobfuscated* which makes the SDK presence detection straightforward. This stage produces fast but less accurate results.

In the second stage, we detect with more accuracy if a cloud service is indeed being used by an app by identifying if specific cloud service APIs are invoked in the app (i.e., as part of cloud service SDKs). Specifically, we first compile a list of cloud APIs of our interest from cloud service documentation. Then we run FlowDroid [8] to construct a global call graph for the app, and search through the global call graph for the reachable cloud APIs. One challenge we encounter is app obfuscation where class and method names are replaced by random strings. To address this challenge, we built *API signatures*, which consist of a series of invariants that are *not changed* by the obfuscation tools, including parameter types and return values, method modifiers, signature of callees, and constant strings that are referenced on the API sub-graph, etc. Instead of comparing method invocations by their names, we compare their API signatures to identify the presence of cloud APIs in the event that an app is obfuscated.

The output of the pipeline is a much smaller dataset ($\sim$1%) composed of apps that use cloud services as backends. Additionally, it tells us the exact cloud APIs that an app is using.

```java
public class ProfileService {
  public static String BUCKET = "profile"; //Container
  public static String A_K_ID;  //AWS Access Key
  public static String S_K;     //AWS Secret Key

  static {
    StringBuilder sb = new StringBuilder();
    A_K_ID = sb.append("AKI*").append("7*").toString();
    StringBuilder sb2 = new StringBuilder();
    S_K = sb2.append("2M**").append("T1**").toString();
  }

  public void deleteProfileImage(...) {
    AmazonS3Client client = new AmazonS3Client(
        new BasicAWSCredentials(A_K_ID, S_K));
    StringBuilder objKey = new StringBuilder();
    objKey.append(userId).append("/").append(fileName);
    client.deleteObject(BUCKET, objKey);
  }
}
```

Figure 2: The code snippet from a real-world mobile app.

3.2 Extracting Credentials from Apps

Next, *PrivRuler* extracts the cloud service credentials from the apps. Recall from Section 2 that such credentials are required by cloud services to both authenticate and authorize an app when it performs a cloud operation. To extract these credentials, we apply a suite of program analysis techniques.

We start with regular expression matching. Cloud credentials often exhibit certain patterns, e.g., AWS access key complies with `AKIA[0-9A-Z]{16}` (More credential examples are available in Table 5). By searching through the decompiled app code for such patterns, we were able to extract many credentials that were directly built into the app.

In many cases, credentials are results of transformations, such as string operations, encryption/decryption procedures, etc. Regular expression matching is limited in reconstructing credentials in these cases. Figure 2 shows an example where credentials `A_K_ID` and `S_K` are built from string concatenation (Line 7-10) before being used to create an AWS S3 client (Line 14-15). To address this problem, we adopted a program analysis approach that combines program slicing and dynamic execution. Specifically, we start with the app entry points computed by FlowDroid, and build a backward program dependency graph through a whole app analysis [73]. Then, we mark the credential parameter of a cloud API as the value of interest and conduct a backward program slicing to identify all instructions and variables that may impact the credential parameter. An obvious challenge is that once a slice becomes inter-procedural, the complexity of the analysis increases dramatically or even becomes untractable. To reduce the complexity, we borrow the idea of *differential analysis* from the prior work [21]. The core of the idea is that since a cloud credential is static and invariable, its value should originate from some *source of invariants* within the program (e.g., constants, resource and manifest files). We can thus prune the search paths to remove those unrelated to the sources of invariants to reduce the analysis complexity.

After the program slices are created, we dynamically execute the slices to reconstruct the credential. Using our prototype, we construct a reduced method for each method on a program slice by re-assembling the program statements that affect credential construction. Then we create an executor method to invoke these reduced methods following the order of the dependency graph. After that, we rewrite the app's *launcher* activity to ensure that the executor method is invoked when the app is being started. At the same time, we add the cloud APIs as logging points, and instrument the reduced methods to ensure that the cloud credentials are logged right before the cloud APIs. Once the instrumented app is ready, we execute the program slices by directly launching the app in an unmodified Android emulator. The advantage of using an Android emulator is that it can effectively address most of the program dependencies, such as framework APIs. However, there are cases that may prevent us from executing the slices and gathering the credentials, e.g., when the slices are dependent on user input environmental conditions. Fortunately, in our evaluation (Section 4.5), we found that such cases are rare, so we leave the development of their solutions to the future research.

Note that we analyze *all* cloud service APIs in an app in order to extract a complete set of credentials. We observed in several cases where different APIs in an app use different credentials. For example, in a *creative service* app, we extracted two credentials, one for testing and one for production. The test credential is never invoked when the app runs, but it was embedded in the app by presumably a developer's mistake. Interestingly, the test credential is more privileged (i.e., with more capabilities) than the production one.

3.3 Mapping Credentials to Capabilities

The next step is to decide capabilities, i.e., authorized cloud operations, of the credentials. Given a credential, recovering its capabilities in the original policy format is infeasible due to the blackbox nature of cloud services. To address this challenge, our solution is to express capabilities in its primitive form—access control tuples. An access control tuple <OP, OBJ>[2] states *what cloud operation* (OP) can be performed over *which cloud object* (OBJ). It captures a capability at its finest possible granularity, while treating the specifics of access control policies as a blackbox. A *capability map* contains a list of access control tuples enabled by the credential.

To build the capability map, static app analysis is not sufficient since the extra capabilities that a credential has but does not use cannot be recovered from the app. To solve the problem, we use a dynamic probing technique [50, 72, 75] that probes cloud services using the credential and inspects how cloud services respond. There are two major challenges involved in this technique. First, how to construct the <OP, OBJ> tuples from probing. Second, how to safely and ethically probe cloud services without impacting individual app users. We focus on the first challenge in this section and discuss solutions to the second in Section 3.6.

Re-constructing *operations* (OP) is straightforward since possible cloud operations are simply exposed in forms of the cloud APIs in mobile SDKs. We could thus enumerate them by probing if the cloud services authorize the invocations of the cloud APIs. The more challenging part is to decide which cloud *objects* (OBJ) can be accessed by those cloud operations. Unlike operations, the cloud objects are not known, and without knowing them, the probes will simply fail as probes without valid objects are considered to be malformed by cloud services.

To solve this problem, we observe that cloud objects mainly fall into two conceptual categories, *containers* and *elements* [14, 15, 45, 47]. A container object is often statically created in the cloud by app developers prior to app launch, and its goal is to "contain" all data produced by the app users (e.g., "bucket" in AWS S3). In contrast, element objects are often dynamically created, accessed and deleted at runtime by individual app instances—they belong to individual app users. Examples of such element objects are the "object" in AWS S3 and the "platform endpoint" in AWS SNS. This distinction enables us to treat them separately. For container objects, their static

[2]We omitted subjects in the tuple as in the app-cloud context, the subject is always the app itself.

nature determines that they need to be embedded inside of the apps. We can therefore extract their values (i.e., object names) in a way similar to how we extract cloud credentials from apps, as discussed in Section 3.2. Element objects, on the other hand, cannot be recovered from apps statically. However, we found that their values do not impact whether or not an operation is authorized, due to the fact that cloud policies are often specified at the granularity of container objects. As a result, we could plug-in invalid element objects with valid cloud operations and container objects to construct a valid dynamic probing request. By sending the probing requests to cloud services and inspecting their responses, we could thus determine if the credential is authorized to perform the operation, and therefore build the capability map for the credential (See Section 3.6 for more details).

3.4 Detecting Excessive Capabilities

Next, we decide if the credential is over-privileged by comparing the capabilities *the credential has* with the capabilities *the app requires*. We rely on static app analysis to identify the capabilities that an app requires. Specifically, we use the cloud APIs identified in Section 3.1 as the set of required cloud operations. For each operation that the app performs, we extract the name of the container objects with the dynamic execution technique. The result is a list of capabilities that an app requires to function, expressed in access control tuples.

One problem that we did not discuss sufficiently in Section 3.3 is how do we enumerate container objects that an app *can but does not* access. Such objects cannot be recovered from static app analysis but they are a key to identify the "over privileges" regarding objects. We combine three methods to address this problem. First, we utilize the special "listContainerObjects" APIs (e.g., `s3:ListAllMyBuckets`) to get a list of container objects that a credential has access to. Such APIs take no parameter except the credential itself and returns a list of container objects associated with a cloud project. Although straightforward, this method is limited in practice because the "listContainerObjects" API itself corresponds to a capability—a particularly powerful one that many app developers intentionally remove from credentials.

The second method we use is guilt by association where we gather credentials and container objects from all the apps of the same developer on Google Play, and cross validate their accesses. The rationale is that a developer tends to have the same or associated cloud settings for different apps [6, 70]. As a result, capability escape and misconfiguration are likely to happen. In practice, we found this method very useful in identifying many objects that credentials have access to, which we would never learn by analyzing a single app. As an example, we identified an over-privileged AWS S3 credential in a smart doorbell app. The credential not only allows access to all users' data of the doorbell, but also users' data from other apps in the family, such as indoor and outdoor cameras (See Section 4.2 for more details).

The third method we use is also based on guilt by association but from an object's perspective. Specifically, We build an undirected graph whose vertices are <`app, container object`> pairs. The container objects are added to the graph as long as their names are non-generic values (i.e., strings that contain at least two vocabulary words or contain non-vocabulary words, such as `viozer-videos-raw` instead of `photo` or `data`). We then draw an edge between two vertices if 1) the vertices have the same container object name; or 2) the vertices correspond to the same app. After that, we cross validate accesses to container objects within each connected subgraph. Not only does this method allow us to find more hidden capabilities of a credential, it also enables a few interesting findings where apps that are published by different developers and have different credentials share access to the same container objects (See Section 4.2 for more details).

By comparing the capabilities that a credential has with the capabilities the app requires, we can thus identify mobile apps that are over-privileged.

3.5 Identifying Vulnerable Apps

We firmly believe over-privilege is a severe security issue that should be fixed whenever possible. Therefore, we have provided responsible disclosures to all developers whose apps are flagged by this study. However, at the same time we recognize that not all extra capabilities are equally harmful. To guide prioritization of vulnerability confirmation and patching, *PrivRuler* features another component that decides when *over-privilege may lead to a vulnerability* that can be exploited by attackers to launch large scale attacks against app users.

The approach we took is driven by heuristics. Specifically, we investigated the over-privileged apps to understand how extra capabilities can be abused (especially when they are combined with other capabilities required by the app) to carry out attacks against app users and summarize the common attack patterns into heuristics. We then apply the heuristics to identify apps that might be vulnerable to similar problems at scale. Due to limited resources, we have focused on a few high profile cloud operations and potential attacks around them, e.g., *spear-phishing attack*, where attackers can launch large scale (and targeted) phishing campaign against app users, *data leakage attack*, where attackers can harvest app users' sensitive data at scale, and *data pollution attack* where attackers can arbitrarily modify app data that other app users depend on. Section 4.2 and Section 4.3 describe attack examples and heuristics in detail.

3.6 Probing with Induced Transaction Failure

While dynamic probing enables us to decide the capabilities that an app credential owns on cloud services, it creates safety and ethical concerns. Any cloud operation that reads (or writes to) a cloud object may lead to data leakage (or pollution) attacks, especially if the object corresponds to app user's personal data. Thus, one significant challenge we face is to ensure that no data leakage or modifications are made due to dynamically probing cloud services.

To address this problem, we introduce a technique called *induced transaction failure* to dynamic probing, inspired by the Zero-data-leakage Vulnerability Verification mechanism in [75]. The core idea is to model cloud operation(s) as a transaction. By injecting failures into the transaction, the cloud operations eventually fail to execute, thus causing no materialized access or change to app users' data. However, through strategically injecting failures, we could indirectly infer the capabilities of the credential in use by inspecting the error code of the cloud operation. The reason we could do this is because, despite looking simple, a cloud operation is implemented as a series of checks with authorization check being one of them. Naturally, failing authorization check (i.e., indicating no capabilities) would return a very different error code from failing other checks. The actual failure we inject differs from operation to operation, but they share a common nature that they are only being evaluated after the cloud credential clears the authorization check.

We illustrate this idea through two examples. In the first example, we show how `Get-` and `Delete-`like cloud operations are protected by the technique. We found that cloud services typically evaluate `Get-` and `Delete-`APIs in the following order[3]: correct container object name -> authorized credential -> correct element object name. The reason is that access control policies in the cloud are often specified at the granularity of container objects. Thus the authorization often comes *after* evaluating the correct container object but *before* evaluating individual element objects. We leverage this artifact to inject failures by providing correct container objects with non-existent element objects. For example, in Figure 2, when an non-existent *objKey* is given to the *deleteObject* API (Line 18), the operation fails silently with no error as long as the credential is authorized, otherwise an *Access Denied* error will be thrown. We could thus indirectly infer if the capability is present by simply looking at the error code. In addition to the obvious benefit that we do not risk accessing individual user's data since the operations failed to execute at cloud side, we also avoid the need to know the exact element objects that are mostly generated at run time by individual app instances, as mentioned in Section 3.3. The way we ensure that the element object is non-existent is quite simple too. We use a fixed hash value, `sha256(author_emails)`, to replace the element object name wherever it is needed.

Next, we show that the technique also protects `Put-`like cloud operations. For most cloud services (e.g., AWS SNS), `Put` needs to operate over a specific endpoint (e.g., publish a message to a specific phone number). By providing an non-existent endpoint, the operation would fail in a similar way as `Get` and `Delete`. One notable exception is the storage service (e.g., AWS S3), where objects can be newly created if an non-existent name is provided to the `Put` API. This could introduce data or state changes to cloud services which we aim to avoid. The way we inject failure in this case is to leverage the transaction-aware APIs itself. For example, to evaluate if a credential has `s3:PutObject` capability, instead of performing a `PutObject` operation, we initiate a multi-part upload operation by using the `initiateMultipartUpload` API. We then immediately abort the upload through `abortMultipartUpload`, signaling a failure to the cloud service, and not uploading any actual data parts. Cloud treats these two APIs as an non-committing, failed transaction, cleaning up temporary state changes (if any). We can however still infer the presence of `s3:PutObject` capability through the status return code of `initiateMultipartUpload`.

Although the specific failures we inject are dependent on the cloud operations under evaluation, the gist is the same: we make sure that the dynamic probing would eventually fail—thus not causing any data leakage or modifications to the app users in cloud, while gathering the signal if a capability is present. The signals here look very much like a "side-channel", but we note that it is inherent to the cloud API design. Cloud services are supposed to be informative to authorized users, in particular when failures and errors happen, but generic and obscure for unauthorized users. We leverage this artifact to probe cloud services in-field without risking accessing users' data or impacting normal cloud operations.

Discussion. We used a semi-automatic approach over a testing cloud account to study how to introduce transaction failures in the cloud APIs. Specifically, we first build cloud API models, such as parameter semantics and related cloud resources, by manually analyzing related cloud documentations. Then for each API, we set up the required cloud resources (e.g., buckets), and create authorized/unauthorized cloud roles to operate on such resources. In the last step, we curate API parameters with (in)valid cloud resource information (based on the API model), and perform automatic API fuzzing with the credentials of both authorized and unauthorized roles. We then conduct differential analysis on the API responses to determine whether there are failures that will reveal the authorization status.

We need to manually verify the failures before using them to probe other cloud accounts. This is a safety measure we need to take, especially because cloud vendors can use inconsistent API responses [31] that may cause inaccuracies to our study. In total, we checked 36 cloud APIs (with ~90 API variants see Table 7 for a full list) of the six cloud services, which took us about one week. While this semi-automatic approach might be enough for analyzing the small number of cloud services in the spotlight, we believe that a more automated solution (e.g., automatically generate cloud API models with NLP-based documentation analysis) is necessary to scale our study to the other cloud services, which we plan to study in future research.

4 Results

4.1 Results Overview

App Dataset ($D_{1.3M}$). We collected a set of 1,358,057 apps from Google Play using the method described in Section 3.1.

[3]There are often much more checks implemented as part of a cloud operation, which gives us even more options to inject failures.

The app size ranges from 579KB to 155MB, with an average size of 26MB. In total, it took us $\sim$1,130 hours to automatically analyze the apps in $D_{1.3M}$ on a Red Hat Linux Server with an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz and 16 GB RAM.

We believe the app dataset ($D_{1.3M}$) represents an approximation of the entire Google Play store due to random sampling. To verify this claim, we gathered the meta-data of all apps from Google Play with google-play-scraper [53], and compared the app distributions. The distribution shows that $D_{1.3M}$ indeed shares a similar distribution with the entire Google Play across all app categories (see [5] for details). Therefore, we can take $D_{1.3M}$ as a representative set to evaluate how prevalent the exposed cloud services are in the real world.

In this study, we focus on two general types of cloud services, *notification* and *storage*, on three cloud platforms, *AWS*, *Azure* and *Alibaba*. Thus, in total we work with six different cloud services, namely AWS S3 and SNS, Azure Blob Storage and Notification Hubs, Alibaba Cloud OSS and Message Service. Table 1 shows the statistics of $D_{1.3M}$: there are 24,300 apps (out of 1,358,057, $\sim$1.8% [4]) containing at least one of the three cloud platform SDKs. However, after the second stage of filtering described in Section 3.1, the dataset is reduced down to 11,891 apps. Among them, 1,663 (14.0%) are found to obfuscate cloud APIs.

Table 1: App dataset ($D_{1.3M}$)

Total Number of Apps			1,358,057
AWS SDK	18,181	S3	9,336
		SNS	1,339
		S3 or SNS	10,473
Azure SDK	4,291	Blob Storage	130
		Notification Hub	1,031
		Blob Storage or Notification Hub	1,161
Alibaba SDK	1,958	OSS	265
		Message Service	6
		OSS or Message Service	267
Any SDK	24,300	Any Service	11,891

Credential Extraction. We have successfully extracted 10,863 credentials from 9,417 (79.2%) apps. Among them, a vast majority are valid (10,578 credentials, 97.4%). Note that there are more credentials than apps due to some apps using multiple cloud services or having additional credentials embedded. Table 5 shows the number of credentials and container objects (extracted via the same method as credential) for each cloud service and sanitized examples from real-world apps.

Although we observe that having the credentials embedded in the apps is the prevalent way for accessing cloud services, it is interesting to explore alternative methods in-use, especially more secure ones. One less common method we observe is for apps to store credentials at a remote server or cloud token service (e.g., Alibaba STS [24] and AWS STS [18]) and fetch them dynamically at app runtime. At its first glance, this seems to offer additional protection. However, a runtime analysis shows that in almost all cases (17 out 20 apps we analyzed) credentials are still shared across app users and can be recovered

[4]This is in line with the statistics shown in AppBrain [7]

Table 2: Overall results ($D_{1.3M}$)

Cloud Service	# Apps with Credentials	# Over-Privileged Apps
AWS S3	7,410	2,239 (30.2%)
AWS SNS	1,266	438 (34.6%)
AWS S3 or SNS	8,569	2,401 (28.0%)
Azure Blob Storage	118	46 (39.0%)
Azure Notification Hub	664	110 (16.6%)
Azure Blob Storage or Notification Hub	779	153 (19.6%)
Alibaba OSS	75	32 (42.7%)
Alibaba Message Service	3	0 (0.0%)
Alibaba OSS or Message Service	75	32 (42.7%)
Total	9,417	2,572 (27.3%)

through runtime app inspection. This is worth highlighting because we saw that some cloud vendors are promoting the design of dynamic credential retrieval (e.g., Alibaba Cloud asks app developers to distribute cloud tokens via private application servers [24]). In such cases, it is important for the cloud vendors to be explicit in their message that without having separate credentials and cloud identities for different app users, such design does not significantly improve security. Rather, it is merely for the convenience of app developers to manage credentials. During our analysis, we often see such false sense of security assumed by the app developers.

A more secure but even less commonly used method is the identity-based cloud policies (e.g., policies associated to Amazon Cognito users [10]) where each app user is granted an independent cloud identity with its own credential and capabilities. In order to onboard to this approach, app vendors are required to properly configure and orchestrate a number of cloud services involving authentication, authorization, and cloud identity and resource management. We hypothesize that the complexity is one of the main reasons why the method is not popular among apps—through a study of apps using AWS, we found that only 312 apps (out of 10,473, 2.9%) are adopting the identity-based policies. Another possible reason is the cloud documentation. Many times cloud documentation (e.g., [12, 23, 44]) demonstrates bare minimum working examples for a quick start. App vendors, especially less security conscious ones, are not spending additional efforts in working out a more sophisticated configuration despite being more secure.

Above mentioned two methods, along with other limitations of the tool (discussed in Section 4.5) represent 20.8% (2474 out of 11,891) of apps where we did not extract cloud credentials despite using cloud services.

Over-Privileged Apps. We found that *2,572 apps (out of 9,417, 27.3%) are over-privileged*, with a total of over 765 million installs. Also, the vast majority (2,297, 89.3%) of these apps use regular user credentials and only 275 (10.7%) use root credentials. This finding proves our hypothesis—encouraging developers to not use root credentials is merely a first step towards improving security. Additional research that helps developers to properly configure credential capabilities is necessary to mitigate the root cause. Table 2 (third column) shows a detailed breakdown of over-privileged apps by cloud services.

Although results from different cloud services should not represent a direct comparison between clouds, we do notice a few interesting things behind the data. First, Azure Notification Hub has a lower over-privilege rate (16.6%) than its AWS counterpart SNS (34.6%). We suspect it is because of the less number of APIs/capabilities that Azure Notification Hub exposes. Moreover, Azure Notification Hub offers two different types of credentials—*full* from *listen*-only (i.e., *connectionstring*), with semi pre-configured capabilities, and explicitly asks mobile developers to use the *listen*-only over *full* to avoid over-privilege [44]. As a result, most apps using Azure Notification Hub use *listen*-only credentials in their code. We do believe that in security sometimes more freedom is not a blessing. A simplified set of APIs/capabilities with a few pre-configured, commonly used capability combos could simplify the security decisions to be made by app developers thus improving security (see our suggestions in Appendix 5).

Another interesting comparison is between Alibaba OSS and AWS S3. The two services are very close to each other from all aspects, e.g., similar authorization models ([11, 25]), an almost one-to-one mapping of cloud APIs [22]. Therefore, we would expect similar over-privilege rates. However, Alibaba OSS had a higher rate than AWS S3 (42.7% vs. 30.2%). By studying the over-privileged apps on Alibaba OSS, we noticed that a larger portion (78.1%) of them are capable of performing any operations on the cloud. We suspect that the difference is mainly driven by cloud documentation: Alibaba Cloud uses credentials with full access to cloud storage as a prominent example on its official website [25, 26], while AWS recommends custom IAM roles with only required cloud capabilities.

Additionally, we make a few interesting observations from the data analysis. First, contrary to the common belief that newer, popular apps (i.e., apps with more users) are more secure, we did *not* observe an inverted correlation between app popularity and if it is over-privileged, as shown in Figure 4. Also, we observed that 70.0% of the over-privileged apps were updated on Google Play within one year of this study, indicating that the risk of excessive cloud capabilities has not been fully addressed despite of the prior efforts [29, 43, 62, 67, 75]. Second, we found that not all cloud capabilities are abused equal. We studied the use-to-grant ratio of different capabilities and identified several capabilities that are less often used but more often granted (Table 7). As an example, the `s3:ListAllMyBuckets` capability is only used by 92 apps, but is granted to another 1,025 apps. As another example, the `sns:ListPhoneNumbersOptedOut` is only used by one app, but is granted to as many as 361 apps. This prompts a security proposal: if a sensitive capability is rarely used by mobile apps, cloud vendors may consider removing (i.e., not expose) it from mobile SDKs to avoid its misuse by the developers.

Vulnerable Apps. Utilizing a few vulnerability patterns, we further identify over-privileged apps that are more likely to be vulnerable. As discussed in Section 3.5, we are targeting three different kinds of attacks: spear-phishing attack, data leakage attack, and data pollution attack. In total, we have identified *1,667 apps (64.8% of 2,572 over-privileged apps) that are very likely to be vulnerable to these attacks* and disclosed our findings to the app developers. We describe case studies and the vulnerability patterns in detail in Section 4.2 and Section 4.3.

4.2 Case Studies

Spear Phishing in Cloud Notification Services. A rising amount of apps are using cloud notification services to send messages to app users (e.g., notify users of an app update), or collect real time data from them. A credential with excessive capability will allow an adversary to abuse the notification services, such as sending a fraudulent message to app users. As an example, `Anonymized Energy` is an energy monitoring app that homeowners use to monitor home energy consumption through deployed sensors. The app uses AWS SNS as the cloud backend. We found that the app has a number of SNS capabilities that it *never requires*. One excessive capability is the `sns:ListEndpointsByPlatformApplication`. This capability allows anyone to list all the users of the app (represented by endpoints with random UUIDs). Another excessive capability is the `sns:Publish` that allows anyone to send notification messages to a specific endpoint. Combining these two capabilities allows anyone to list and pick any endpoint and send it a notification message with arbitrary content. Moreover, since an endpoint is often created with custom user data (e.g., email, name), the phishing can be very targeted (see our attack demo online [5]). We did not launch a phishing campaign against real app users due to ethical reasons. However, we want to highlight that such phishing is more convincing because the messages are delivered by the mobile OS in the context of the app (see Figure 5 for the screenshot), instead of generic ways like SMS. Given victims are already users of the app, they are very likely to fall for these phishing messages that are from such "trusted sources" [27, 38].

In our analysis, we found that the above excessive capabilities are very common for apps that use cloud notification services, leading to spear phishing attacks. For example, `Anonymized Smart Building`, `Home Automation` and `DVR` apps, just to name a few, all have the similar issue. We have summarized such capability patterns into a heuristic to help us better identify potentially vulnerable apps (See Section 4.3 for details). In total, we have identified and reported 96 apps that are very likely to suffer from the spear phishing attack. They are installed for over 4.97 million times, representing a large potentially vulnerable user base.

Data Leakage in Cloud Storage Services. `Anonymized Parenting` is an app that allows parents to record and share baby's developmental milestones in the form of logs, photos and videos. The app is very popular—it has been used by over 13 million families in total (according to the app description). The app uses AWS S3 as its storage backend. It creates two separate S3 buckets, one for storing users' photos (i.e., `parenting-photos`) and one for videos (i.e., `parenting-videos`). These two buckets are *shared across app users*. As discussed in Section 3.3, capabilities in cloud

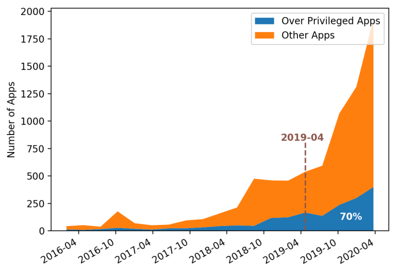


Figure 3: Over privileged app distribution over app update time.

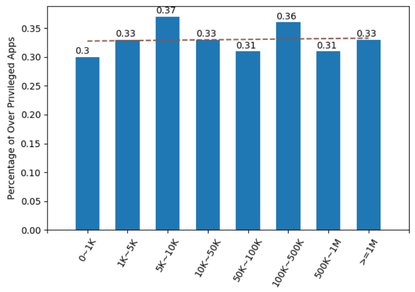
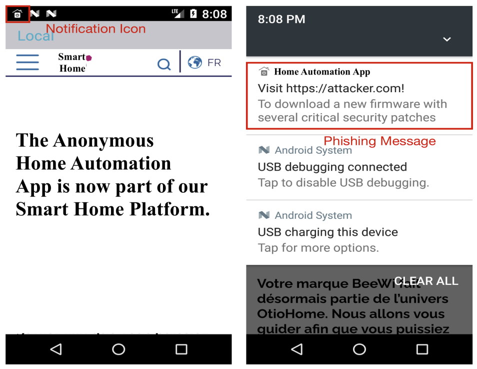


Figure 4: Over privileged app distribution over app installs.

Table 3: Vulnerability patterns

Vulnerability Patterns	# of Apps
P1: Get + Excessive Put	146
P2: Put + Excessive List	1,345
P3: Put + Excessive List + (Excessive) Get	925
P4: Put + Guessable Object + (Excessive) Get	62
Total	1,667



The Anonymous Home Automation App is now part of our Smart Home Platform.

Home Automation App
Visit https://attacker.com!
To download a new firmware with several critical security patches

Android System
USB debugging connected
Tap to disable USB debugging.

Android System
USB charging this device
Tap for more options.

Votre marque BeeWi fait désormais partie de l'univers OtioHome. Nous allons vous guider afin que vous puissiez

Figure 5: Phishing UIs of `Anonymous Home Automation App`

are typically set over container objects (i.e., buckets in this case), and element objects such as individual users' photos and videos do not have separate capabilities associated with them. As a result, credentials embedded in the app have the capability to read and write any user's data. This sounds like a broken design, but with one caveat—an app instance needs to provide a correct name for the element object that it is trying to access. `Anonymized Parenting` leverages this condition to provide user data isolation. It creates pseudo-random subdirectories for each user and organizes user's data under the subdirectory. Since pseudorandom subdirectories are part of path names for element objects, one user should not be able to access another user's data even though the credential she uses has the capability to do so. Such design is used widely in practice, and is considered secure if the pseudorandom subdirectory name can be kept secret to its own user. However, we found one excessive capability in credential that broke this assumption and rendered the whole design pointless. Specifically, the credential embedded in `Anonymized Parenting` has the `s3:ListBucket` capability. This capability is never used by the app, but it effectively allows an adversary to enumerate all the subdirectories under `Anonymized Parenting` buckets and thus accessing any user's data. In our analysis, we found many apps suffer from a similar problem—excessive capabilities such as `s3:ListBucket` causes leakage of all app users' data. Some of these apps actually deal with extremely sensitive data. For example, `Anonymized Dictation` is an app that provides transcription services for voice recordings. The target audiences are, as the app suggests, "research, medical, legal or any other purpose" which indicates the sensitivity of the data stored in the cloud.

The problem is even worse for apps whose storage design is inherently insecure. In our analysis, we found that instead of creating pseudorandom strings, some apps simply use guessable strings or even users' PIIs as subdirectory names. This, combined with the excessive capabilities such as `s3:ListBucket`, allows adversaries to efficiently target individual users. For example, `Anonymized Travel` is a travel log book app for recording car odometer readings, travel routes, stops, geo-locations, gas/meal receipts, etc. These data are stored in an AWS S3 bucket directly using users' emails as subdirectory and file names. An adversary with some background knowledge about an user's email can easily read and write that users' data, and not surprisingly, credentials in the app have the excessive `s3:ListBucket` capability that allows the adversary to easily check if a specific user is using the App. What is ironic is that `Anonymized Travel` claims itself to be HIPAA compliant (we quote "HIPAA Compliance: `Anonymized Travel` encrypts protected health information (PHI) and personally identifiable information (PII) in a secure environment").

Data Isolation for 2B Apps. Excessive capability does not necessarily mean additional operations such as `s3:ListBucket`, but can also be operations that are applied to a broader scope than what is required by the app. This is another source of data leakage in cloud storage services. Although many poorly designed apps suffer from this problem, we found it is particularly common among a specific type of apps, i.e., to-business (2B) apps, which merits a separate discussion.

2B apps target businesses, but are eventually used by end users (i.e., employees of businesses). This introduces another layer of complexity in managing data access as the apps need to not only provide isolation for users but also isolation for businesses. Moreover, there are additional requirements for business owners to access employees' data, adding more complexity to the access control. Not surprisingly, we found that a number of credentials in 2B apps are over-privileged—they allow cloud operations to be performed on a broader scope of users' data. The result is often a broken data isolation across employees or businesses. The latter is particularly worrisome as different businesses using the same app are often competitors with a conflict-of-interest.

As an example, `Anonymized Deskless` is a 2B app that provides a reporting platform for deskless workforce (e.g., security, janitorial, facility). Deskless employees use the app to record and report their routine tasks (e.g., area is patrolled/-cleaned) and their managers (i.e., business owners) can keep track of the tasks, approve requests, etc. What we found is that app users' data is stored in several AWS S3 buckets and the credential embedded in the app is able to access all of them. As a result, not only employees of a business can access each other's data, one business can access another business's data as well. The data under risk here can be extremely sensitive, e.g., locations of security guards, pictures of facility internals.

Cross App Data Leakage. Using the method discussed in Section 3.4, we found several cases where over-privileged credentials in one app allow access to data that is generated by another app. Such cases largely fall into three categories.

In the first category, the *same credential* is used by several different apps developed by the same developer. For example, `Anonymized Barber` is an app for helping barber shop owners to manage their customer base (e.g., manage contacts, automatically send out reminders). The developer of `Anonymized Barber` has a whole series of similar apps, customized for skin salons, nail salons, tattoo shops, etc. All these apps share the same credential and not surprisingly, the credential is over-privileged—it enables any of the apps to access data belonging to other apps. As another example, the largest group of apps we identified that share the same credential contains 150 `teacher-parent communication` apps. This app group is similar to 2B apps: instead of having a single app, the app developer decided to provide one app for each school. The intention of the developer seems to be providing business isolation, but the effect of using the same credential invalidated any isolation provided through having different apps. As a result, one school can access data belonging to any other schools, despite that they are using "different" apps.

In the second category, *different credentials* are used by different apps, but they still lead to cross app data access. As an example, developer of `Anonymized Cam` (a home monitoring camera app) has a family of apps for different types of smart home devices. All these apps have their own credentials and data containers in cloud, and most of them are isolated from each other. However, we found one app, `Anonymized Bell` (a smart doorbell), contains a credential that enables data access to all the other apps in the family. A deeper analysis revealed that all the app credentials in the family are under the same cloud account, but they belong to different IAM users that have different capabilities. This shows that the developer clearly had isolation in mind and she successfully configured capabilities for most of the apps in the family. However, a mistake was made for the credential in `Anonymized Bell` that it was granted excessive capabilities to all the other data containers belonging to other apps. As a result, an `Anonymized Bell` user can access data from all the other apps in the family.

The third category of the apps are owned by different parties, but use the same cloud storage. An example is the group of taxi and ride sharing apps: `Anonymized Ride`, `Anonymized Cab` and `Anonymized Driver`. These apps utilize different cloud credentials and are submitted to Google Play by three different companies in India and Bulgaria. However, we find that they upload passenger data, such as photo, all to the same bucket "voilacabsproduction" under a common cloud account. Further inspection of these apps' code reveals that the cloud-related component very likely has all been developed by the same Indian company (i.e., `techintegrity.in`). This raises a serious concern since outsourcing cloud service development may not only result in data leakage across app developers, which could be completely unaware to them, but also enable data sharing across the regions (i.e., between India and Bulgaria in this case) under different privacy regulations.

Data Pollution Attack. The other side of the coin to the data leakage is data pollution. The essence of the attack is that an adversary may leverage excessive capability to "pollute" the data in cloud storage that individual app instances read from. As an example, `Anonymized Santa` is an app that downloads pre-recorded Santa videos from AWS S3 and plays the videos to children. Credential in the app only requires `s3:GetObject` capability to download videos, but it is granted with `s3:PutObject` capability that allows it to write to the cloud storage. As a result, an adversary may deliver any videos to the app users. Although the attack mechanism is simple, the effect can be dangerous (i.e., exposing young children to video messages from random guys on the Internet). Even worse, since cloud storage can be utilized to distribute executable objects (e.g., firmware, JavaScript), the potential damage can be more severe. For example, we found a cryptocurrency trading app, `Anonymized Crypto`, that has its webview resources (e.g., javascripts) hosted in AWS S3. These resources are meant to be static and only modifiable by the app vendor. However, credential in the app actually has the `s3:PutObject` capability that allows adversaries to modify them in arbitrary ways. More interestingly, these webview resources are shared with normal desktop users as well, making the problem even worse.

4.3 Common Vulnerability Patterns

Granting more capabilities than needed should be avoided whenever possible. However, not all of them may lead to real vulnerabilities. In this section, we look for more indicating signals of when an over-privileged app becomes vulnerable. Specifically, we explore how the combinations of capabilities lead to severe security issues, and summarize our findings into a few vulnerability patterns in Table 3. In total, we have identified 1,667 potentially vulnerable apps, with 913 apps exhibiting multiple vulnerability patterns.

***P1*: Get + Excessive Put.** The first vulnerability pattern is the combination of a `Get` with an excessive `Put`. Here the `Get` and `Put` are generalized capabilities that allow read and write operations on any cloud service, respectively. When an app exercises the `Get` capability, it means the app actually reads data from the cloud. In contrast, an excessive `Put` capability means that the app does not write to the cloud although it has the capability to do so. The combination of these two capabilities is a strong indicator of data integrity vulnerabilities. An adversary may abuse the excessive `Put` capability to pollute the data source where other app instances read from. This type of attack is found in 146 apps (out of 1,318 apps that use `Get` but not `Put`) where app resources are mainly public and read-only, like the the data pollution attack mentioned in Section 4.2.

***P2*: Put + Excessive List.** A variant of the above pattern is the combination of `Put` and excessive `List`. Instead of public app resources, the fact that an app has and uses `Put` capability indicates that the app produces user-specific data and stores them in the cloud (the most prevalent pattern used by at least 6,100 apps). The excessive `List` capability allows an adversary to enumerate data stores related to all app users, and then pollute the data of any individual user. We found that this pattern affects the most number of apps (1,345) in our study. One special case of this vulnerability pattern is the spear phishing attack in cloud notification services (e.g., AWS SNS). In the attack, an app creates an endpoint in the cloud service by providing an app token with `Put`. An adversary can list all the available endpoints (i.e., app users) and push notifications to any of them.

***P3*: Put + Excessive List + (Excessive) Get.** Another type of attack that can happen to the 6,100 apps that store user-specific data in the cloud is data leakage attack. Assuming the data that an app produces is private, then any cloud capability that could potentially enable one app user to access another app user's data is a strong signal for data leakage vulnerability. <`List`, `Get`> is exactly such a pair of capabilities. As discussed in Section 4.2, the prevalent design for cloud storage is to allow app users to share the same container (e.g., bucket), and rely on pseudo-random pathnames to isolate users. The excessive `List` capability effectively breaks this isolation and allows adversaries to enumerate pathnames under a container and thus access individual users' data with `Get`. We found 925 apps that are potentially vulnerable to such a data leakage attack. Note that the `Get` capability needs not to be excessive. We found that 1,527 apps use/require `Get` to retrieve user data from the cloud. However, 293 of the apps are potentially vulnerable since the required capability can still be abused by adversaries.

***P4*: Put + Guessable Object + (Excessive) Get.** Similar to above, this vulnerability pattern involves `Put`, which indicates that an app stores private data in the cloud. However, instead of using pseudo-random object names for user isolation, the app uses predictable information, such as phone number and user birthday. Therefore, an adversary may still access user's data by guessing the object names, although `List` capability is not granted. To determine whether an app uses guessable object names, we collected 21 keywords for predictable user data from previous studies [37, 49], and tracked these keywords in the app code to determine if the predictable data flows into object names. Using this method, we were able to identify 62 apps that are vulnerable to this data leakage attack.

Vulnerability Confirmation. Patterns discussed above help us identify 1,667 different apps that are *likely to be vulnerable*. To confirm the result, we try to launch concrete attacks against these apps in an ethical way. Specifically, we create test users and perform attacks against dummy data of the test users. This is largely a manual effort. Till the time of the submission, we have validated results for a total of 100 apps that are randomly drawn from the 1,667 apps. Results suggest that above vulnerability patterns are very indicative: among the 100 tested apps, 98 are confirmed to be vulnerable. The only 2 false positive cases (`learning vehicle game` and `local business` app) are both flagged by the pattern P3 where the apps produce data and store them in cloud. However, the data produced by these two apps are not meant to be private (i.e., game scores and business cards respectively), thus breaking the assumption for the vulnerability pattern.

4.4 Vendor Responses

We have made responsible disclosures to app developers, cloud vendors and Google Play. For app developers, we contacted them via their registered emails on Google Play. In total, we have sent out 2,572 emails for all the over-privileged apps, each with a detailed description of the excessive capabilities, possible attacks if any, and our suggestions on the fix. Our emails were opened by 1,347 app developers, bounced from 451 (i.e., email address no longer valid), and remain unopened for the rest 737. Out of the 1,347 app developers that opened our email, only 56 of them responded. The rest are either automatic responses or no response at all.

We were able to make some preliminary observations from the 56 responses. First, several app developers responded to us that they feel confused about the capabilities offered by the cloud services. Second, it often takes multiple rounds of revisions before the issue gets completely fixed. As an example, we found that the developer of a `navigation app` has introduced another credential with limited capabilities in new app versions, but failed to revoke or restrict the capabilities for the original credential, leaving the doors open for attackers. In general, our experience interacting with app developers suggests that existing cloud services are not really mobile app friendly. This motivates us to make suggestions as discussed in Appendix 5. We are also interested in why many app developers (1,291 out of 1,347) opened but did not respond to our emails. We therefore launched another analysis against these apps 30, 90 and 120 days after the initial reach-out. To our surprise, we found that after 30 days, 161 apps have either made changes to the credential capabilities directly, or revoked the over-privileged credentials completely in newer app versions. The number of fixed apps increases to 228 and 317 after 90 and 120 days, respectively. This indicates our reports are indeed getting attention from a broader set of app developers. Appendix Table 6 shows a few apps that were patched due to our report. We have also released some anonymized responses online [5].

We reported our findings to the cloud service providers and received quick responses from both Azure and AWS. They appreciated our effort to secure cloud services, but due to the fact that they do not own the apps, they recommended reaching out to app developers directly. We also provided the list of vulnerable apps to Google Play, and asked their help to contact the app owners. Google Play is still investigating security risks of the reported apps and yet to determine their next steps.

4.5 Evaluation

Ground-truth dataset (D_{200}). We randomly sampled 200 apps from the 24,300 apps that contain any cloud SDK, and used them to build a ground-truth dataset. We conducted both static and dynamic app analysis to confirm the cloud service usages. Specifically, we identified the cloud APIs and the entries (e.g., activities, buttons) to trigger them using JADX [4]. For the apps that have any cloud API, we installed them on a Nexus 6P phone, activated them and manually interacted with the apps to cover as many app functions as possible. At the same time, we hooked the cloud APIs using Frida [3] to collect the container objects and cloud credentials. In total, we found that 85 apps use at least one of the six cloud services (most of the other apps use different services such as AWS Kinesis and Azure Analytics), with 1,061 cloud API callsites (including 45 obfuscated ones), 137 cloud credentials, and 88 container objects. Out of the 85 apps, 25 apps are confirmed to be over-privileged. This process took us around 67 hours.

Effectiveness of *PrivRuler*. We ran *PrivRuler* on D_{200} and compared the findings to the ground-truth data. Table 4 shows the results. Below we discuss the false positives (FPs) and false negatives (FNs) of identifying cloud APIs, cloud credentials and containers, and the overall results of over-privileged apps.

• *Cloud service APIs*. *PrivRuler* successfully identified 1,009 (95.1%) cloud API callsites, with 972 (95.7%) non-obfuscated and 37 (82.2%) obfuscated ones. It failed to locate 52 callsites (FNs) in five apps for a few reasons. First, *PrivRuler* failed to discover 24 reachable callsites in two apps due to the incomplete modeling of system callbacks (such as `onBindViewHolder`) in FlowDroid [8]. Second, *PrivRuler* missed 19 callsites in another two apps because of the underlying parsing errors in Soot. Moreover, our tool identifies obfuscated cloud APIs using a list of API signatures generated from cloud SDKs. Unfortunately, we missed nine callsites for obfuscated APIs since their API signatures were not included in our prototype. In addition to FNs, we incorrectly identified 11 cloud invocations (FP) in one app: the cloud APIs will not be invoked since they appear in a private `Service` that is never triggered. Nevertheless, they are still flagged because *PrivRuler* treats all registered services as entry points regardless of whether they are launched or not (this is an inherited tradeoff made by FlowDroid to accommodate ICCs in mobile apps).

• *Cloud credentials*. *PrivRuler* identified 110 (80.3% of 137) cloud credentials. It failed to extract 27 credentials from 10 mobile apps (FNs) for a few reasons. First, in three apps, the cloud credentials are first stored in a local storage (e.g., shared preference), and later retrieved from the storage and used in cloud service APIs. *PrivRuler* falls short in such cases because it can not accurately model data flows for local storage. Second, in two apps, *PrivRuler* incorrectly flagged the cloud APIs that consume cloud credentials non-reachable, and thus didn't initiate the reconstruction of the credentials. Third, *PrivRuler* failed to generate fully executable program slices for reconstructing the cloud credentials in the remaining apps. This is due to the challenges to accurately build *PDG* and interacting with the highly sophisticated apps. One typical example is the app that fetches its credentials from a remote server by providing a valid `userid`. Our tool was not able to generate the `userid` variable and therefore failed to interact with the server. Notably, such FNs will only lead to an under-estimation of the number of over-privileged apps since, without the credentials, we were not able to report whether the apps are over-privileged.

• *Cloud containers*. *PrivRuler* successfully extracted 74 (84.1% out of 88) container objects. It missed 14 container objects (FNs) almost for the same reasons as in cloud credential extraction. In the cases where a cloud API is identified while its corresponding container object is not, we made an over-estimation that the app requires the cloud capability (represented by the cloud API) to access *all containers*. Therefore, the FNs will not lead to false alarms (but rather under-estimates) of over-privileged apps. Besides, we noticed that 87 (98.9%) container objects are indeed hard-coded in the apps and shared by app users, and the remaining app `home inspection service` uses random UUIDs as the names of container objects. Therefore, none of the container objects in our ground-truth dataset reveals any personally identifiable information (PII) of the app users. We can use them to construct cloud probes with very low privacy risks to individual app users.

• *Overall results*. *PrivRuler* is highly accurate in identifying cloud service usages—no FPs for cloud credentials and container objects, and only 11 FPs (1.1%) for cloud APIs. However, we indeed missed at least one cloud APIs in five (5.9% of 85) apps. For three of the apps, we failed to extract their cloud credentials either and therefore were not able to determine whether they are over-privileged or not. But for the remaining two apps, we incorrectly reported them as over-privileged because the cloud capabilities (i.e., APIs) are indeed required by the apps while we mistakenly marked them as being excessive. In other words, the FNs in cloud API extraction leads to 8.0% false alarms of over-privileged apps in D_{200}. We expect such false alarms to be mitigated by a better app analysis technique and a more comprehensive list of API signatures.

5 Countermeasures

We believe there are several suggestions that we can make to both cloud vendors and app developers to help mitigate the problem.

Cloud Vendors. Perhaps what bothers app developers most is the complexity in the cloud capability model itself. To date, there are 83 distinct capabilities in AWS S3 and 34 capabilities in AWS SNS. In the process of issue disclosure, we noticed

Table 4: Evaluation results on ground-truth app set D_{200}.

	# Apps			# Cloud APIs			# Credentials			# Containers		
	Groundtruth	PrivRuler	%	Groundtruth	PrivRuler	%	Groundtruth	PrivRuler	%	Groundtruth	PrivRuler	%
AWS	66	61	92.4%	963	919	95.4%	111	90	81.1%	65	55	84.6%
Azure	16	15	93.8%	74	66	89.2%	20	16	80.0%	19	16	84.2%
Alibaba	3	3	100.0%	24	24	100.0%	6	4	66.7%	4	3	75.0%
Overall	85	79	92.9%	1061	1009	95.1%	137	110	80.3%	88	74	84.1%

that many app developers are confused about when and how to use these capabilities. One response we quote is "amazon has too many services I am always getting lost in that console"[5].

Fine grained capabilities enable cloud services to support diverse functions and we do not intend to diminish its value. However, we want to highlight that many mobile apps, especially the ones from small/medium vendors, follow a few usage patterns of cloud services. Instead of relying on app developers to cherry pick capabilities, it may be better to provide a few dedicated capability templates for mobile developers. Some cloud service vendors are already making efforts in this direction, e.g., AWS provides a few capability templates (termed *Policies* [17]). But these templates are mostly for illustration purposes and they cannot meet functional needs of mobile apps (as shown in Section 4.3). How to summarize common functional scenarios of mobile apps into well tailored capability templates and educate developers to pick the most suitable one represents an interesting research opportunity for the future.

In addition, we observed that many cloud capabilities are barely used in mobile apps, e.g., `ListPhoneNumbersOptedOut` in AWS SNS (See Table 7 for more examples). Given that these APIs may lead to potential security/privacy issues, it might make sense to consider removing them from the offering of mobile cloud SDKs to eliminate the possibility of being misused by app developers.

App Developers. The first and foremost task for app developers is to review the cloud service usages and only grant the minimal required capabilities to their apps. In particular, for apps that handle user personal data in the cloud services, the developers should consider app user isolation, e.g., requesting users to authenticate to the cloud services, and create isolated cloud resources and credentials for different users. This helps to mitigate the cross-user data leakage/pollution attacks. Additionally, we observed many cases where apps of the same developer share or use associated cloud resources/credentials, which leads to cross-app (or cross-business) attacks (Section 4.2). In such cases, it might be reasonable for app developers to use different cloud service accounts that provide inherent data isolation.

6 Discussion

Ethical Considerations. We pay special attention to ensure that we respect legal and ethical boundaries. Our work was reviewed by IRB, who determined that the work does not involve human subjects and is exempted from further review. We also made responsible disclosures to all parties in the ecosystem, including app developers, cloud service vendors, and Google Play (see Section 4.4 for their responses). However, due to the nature of the work, there are still several areas that we would like to invite the community to participate in the discussion and establish best of practice guidelines.

- *Extracting Cloud Credentials*. A cloud credential is technically a "secret" , but it is distributed publicly (i.e., together with mobile apps in app stores) and with the intention to be used by the public (i.e., shared by all app users). In our opinion, it is treated as an integral part of the app, similar to other types of app meta-data such as API endpoints. Knowledge of such data can be utilized in testing with good intent, as long as they are recovered in an ethical way and not prevented by the laws. The way we recover the credentials, i.e., static and dynamic app analysis, is similar to manually reverse-engineering mobile apps. According to our research on related laws and discussions [39, 42, 52, 54, 64, 66], this practice is not unethical/illegal in most regions across the globe (e.g., EU, US, AU). First, we gathered all the apps from public sources (i.e., Google Play) lawfully, and reverse engineered them with a benevolent goal of enhancing their security, rather than infringing on their copyright or abusing the cloud services (which are clearly unethical). Second, while some software includes an End User License Agreement (EULA) that prevents reverse engineering, Google Play apps use Google's Term of Use (ToU) [35] as an EULA for app users, which does not place any restrictions on reverse engineering. In fact, reverse engineering has become a popular choice in mobile security research, and generated significant security benefits and received numerous positive feedback from app developers [29, 61, 67, 75]. However, we do want to note that reverse engineering an app may be against the will of its developer implicitly. Specifically, we found that around 14% apps obfuscated the cloud APIs, the intention of which we assume is to prevent others from gathering knowledge of the app. In this work, we analyzed these apps following prior practices [74, 75]. But we look forward to feedback from the community.
- *Probing Cloud Services*. Dynamic probing is a common practice to find vulnerable or malicious services in the wild [28, 32, 36, 50, 58, 72, 75]. While in practice there will always be some form of impact to the service as the result of probing (e.g., service logging), the ethical guideline we follow in this work is that the probing should by design not introduce any materialized state changes to the cloud services, leak any real users' data, or impact cloud service operations. Towards this guideline, in addition to following best practices implemented in prior works, we put in extra efforts to minimize any potential harm to cloud services or app users. First, we create

[5]In the context of our interaction, the app developer refers to the different AWS S3 APIs and capabilities as different "services".

testing accounts on all cloud services involved in this study and launch a thorough field study to ensure that the behavior and effect of cloud APIs are fully understood before they are exercised. Second, the goal of the work is not to fuzz testing cloud services. We intentionally design the probes in a way that is lightweight and compliant with cloud service specs to avoid *costs and damages to the cloud services*. Third, we introduce transaction failures to the probes (Section 3.6), and by design avoid any *modifications to individual users' data*, e.g., `write`- and `delete`-like APIs will always fail without being executed by cloud services. Fourth, we limit the *leakage of individual users' data* by utilizing cloud APIs that require equivalent capabilities. For example, instead of using the `listObjects` API that enumerates all user objects, we evaluate the `headBucket` or `getBucketInfo` API, which requires the same capability but does not reveal data of user objects. On top of these efforts, as a safety net, we execute each probe in a dedicated and isolated environment (i.e., a newly created Android emulator) and destroy any data residue except the status of the probe.

• *Communicating with App Developers*. We would like to obtain explicit (e.g., email approval) or implicit (e.g., bug bounty program) consent from app developers before launching the study. However, bug bounty programs are extremely rare among apps [20] and the response rate from app developers is very low (~2.2% based on our experience). Given the severity of the problem and the easiness to launch data attacks, we decide to analyze the apps to find the vulnerable ones, and make responsible disclosures to the stakeholders, similar to prior work [6, 75].

Limitations and Future Work. *PrivRuler* relies on program slicing and dynamic execution to recover cloud credentials. This approach handles simple data structures (e.g., strings) well. However, it is inherently limited in cases where credentials are stored and then retrieved from local storage, in remote servers that require authentication, and complex data structures (see Section 4.5 for more details). We are evaluating extensions to better cover such missed cases. Also, we assume that any data uploaded by app users is private. This assumption may not hold for all apps. Therefore, our study could benefit from a component that evaluates the nature of the data in the clouds, which also helps to determine severity of the data attacks. In addition, although we used heuristics (Section 4.3) to detect potentially vulnerable apps, confirming a vulnerability is still largely a manual effort. Completely automating this process incurs both technical and ethical concerns, which we leave for future work.

Further, our work only focused on a few high profile cloud services. We believe similar ideas, such as dynamic probing, can be used to study other cloud services. However, as discussed in Section 3.6, this task requires a more automated method to build models and perform safe testing for the cloud APIs. Besides, in addition to the heuristics, we believe there are many cloud capabilities that are benign looking, but when combined with other capabilities, can lead to subtle attacks to app users. We plan to launch a more comprehensive and automated study with more cloud services and a richer set of cloud capabilities to uncover these cases.

7 Related Work

Security of cloud service credentials. Previous studies have investigated the problem of credential leakage in both source code repositories [30, 43, 62] and mobile apps [29, 67, 74]. Instead of the leakage of a credential itself, our work focuses on the capabilities owned by the credential and their implications on app users' security. Our work is inspired by LeakScope [75], which reports cloud credential misuse in mobile apps and proposes techniques for dynamic cloud probing, a method we improve to serve our purpose. However, unlike LeakScope that just determines whether a credential carries root privilege or regular privilege, our study is meant to understand the security implications of exposed credentials even after their capabilities have been significantly limited. As a result, we have to identify the fine-grained capabilities of the credentials, and determine whether they exceed the legitimate needs of apps and how their combinations lead to serious security hazards. This enables us to not only detect over-privileged credentials for normal users, but more importantly uncover new attacks (e.g., spear-phishing) by integrating multiple seemingly benign capabilities.

Detecting least privilege violations. *Principle of least privilege* has been well discussed on different modern systems in the past decades [19, 48, 63, 65]. Particularly, with the rise of mobile platforms, many recent studies have focused on least privilege of mobile apps, e.g., Stowaway [33], PScout [9], WHYPER [56] and AutoCog [57] report unnecessary permissions requested by an app based on its APIs and app descriptions. Our work, compared to above studies, does not study capabilities (i.e., permissions) that allow access to on-device resources. Instead, we focus on capabilities to access cloud resources and report how combinations of such capabilities may cause security and privacy risks.

8 Conclusion

We propose a systematic approach to recover cloud credentials from apps, infer their capabilities, and detect those credentials that grant apps excessive capabilities than the legitimate needs of the apps. We then study the security implications of the exposed credentials, and demonstrate how combinations of cloud capabilities may lead to serious security and privacy risks. We implement a prototype and run it on two cloud services across three mainstream cloud providers. Our large-scale experimental study of more than 1.3 million apps reveals that from the apps where credentials can be extracted, 27.3% have over-privileged credentials, covering hundreds of millions of users. Further analysis shows that a majority of these over-privileged apps are vulnerable to a series of data attacks. We have made responsible disclosure of our findings to both cloud providers and app developers, leading to the fixes of over-privilege issues in hundreds of real-world apps.

References

[1] Alibaba cloud. https://us.alibabacloud.com/en.

[2] Aws. https://aws.amazon.com.

[3] Frida. https://frida.re.

[4] jadx. https://github.com/skylot/jadx.

[5] Supplementary materials. https://sites.google.com/view/privruler/home.

[6] Omar Alrawi, Chaoshun Zuo, Ruian Duan, Ranjita Pai Kasturi, Zhiqiang Lin, and Brendan Saltaformaggio. The betrayal at cloud city: An empirical analysis of {Cloud-Based} mobile backends. In *{USENIX} Security'19*.

[7] AppBrain. Android development tool statistics and market share. https://www.appbrain.com/stats/libraries/development-tools.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[9] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *CCS'12*.

[10] AWS. Amazon s3: Allows amazon cognito users to access objects in their bucket. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_examples_s3_cognito-bucket.html.

[11] AWS. Amazon s3: Allows read and write access to objects in an s3 bucket. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_examples_s3_rw-bucket.html.

[12] AWS. Aws sdk for android: S3 transfer utility tutorial. https://github.com/awslabs/aws-sdk-android-samples/blob/main/S3TransferUtilitySample/S3TransferUtilityTutorial.md.

[13] AWS. Best practices for managing aws access keys. https://docs.aws.amazon.com/general/latest/gr/aws-access-keys-best-practices.html.

[14] AWS. Buckets overview. https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingBucket.html.

[15] AWS. Creating an amazon sns topic. https://docs.aws.amazon.com/sns/latest/dg/sns-create-topic.html.

[16] AWS. Lock away your aws account root user access keys. https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html#lock-away-credentials.

[17] AWS. Policies and permissions in iam. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html.

[18] AWS. Temporary security credentials in iam. https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_temp.html.

[19] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI'08*.

[20] Bugcrowd. Public bug bounty program list. https://www.bugcrowd.com/bug-bounty-list/, Jan 2022.

[21] Yi Chen, Wei You, Yeonjoon Lee, Kai Chen, XiaoFeng Wang, and Wei Zou. Mass discovery of android traffic imprints through instantiated partial execution. In *CCS'17*.

[22] Alibaba Cloud. Function mapping between oss api and s3 api. https://www.alibabacloud.com/forum/read-148.

[23] Alibaba Cloud. Object storage service - quick start. https://www.alibabacloud.com/help/en/object-storage-service/latest/android-quick-start, Dec 2021.

[24] Alibaba Cloud. Set up direct data transfer for mobile apps. https://partners-intl.aliyun.com/help/en/object-storage-service/latest/set-up-direct-data-transfer-for-mobile-apps, Dec 2021.

[25] Alibaba Cloud. Use ram to manage oss permissions. https://www.alibabacloud.com/help/en/resource-access-management/latest/use-ram-to-manage-oss-permissions, Oct 2021.

[26] Alibaba Cloud. Ram policy editor. https://help.aliyun.com/document_detail/32203.html, February 2022.

[27] Terranova Worldwide Corporation. Spear phishing vs. phishing: Everything you need to know. https://terranovasecurity.com/spear-phishing-vs-phishing/, Jan 2021.

[28] Ang Cui and Salvatore J Stolfo. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *ACSAC'10*.

[29] AD Diego. *Automatic extraction of API Keys from Android applications*. PhD thesis, Ph. D. dissertation, UNIVERSITA DEGLI STUDI DI ROMA" TOR VERGATA, 2017.

[30] Zhen Yu Ding, Benjamin Khakshoor, Justin Paglierani, and Mantej Rajpal. Sniffing for codebase secret leaks with known production secrets in industry. *arXiv preprint arXiv:2008.05997*, 2020.

[31] Luc Van Donkersgoed. How aws dumps the mental burden of inconsistent apis on developers. https://www.lastweekinaws.com/blog/how-aws-dumps-the-mental-burden-of-inconsistent-apis-on-developers/.

[32] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. {ZMap}: Fast internet-wide scanning and its security applications. In *{USENIX} Security'13*.

[33] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *CCS'11*.

[34] Google. Android apps on google play. https://play.google.com/store/apps.

[35] Google. Google play terms of service. https://play.google.com/about/play-terms/, Oct 2020.

[36] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *{USENIX} Security'12*.

[37] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. {SUPoR}: Precise and scalable sensitive user input detection for android apps. In *{USENIX} Security'15*.

[38] NortonLifeLock Inc. Spear phishing: A definition plus differences between phishing and spear phishing. https://us.norton.com/internetsecurity-malware-what-spear-phishing.html, Jul 2021.

[39] Legal Information Institute. 17 u.s. code § 1201 - circumvention of copyright protection systems. https://www.law.cornell.edu/uscode/text/17/1201.

[40] Suman Jana, Úlfar Erlingsson, and Iulia Ion. Apples and oranges: Detecting least-privilege violators with peer group analysis. *arXiv preprint arXiv:1510.07308*, 2015.

[41] Dan Kobialka. Aws cloud data leak: Uk consulting firms' sensitive information exposed. https://www.msspalert.

com/cybersecurity-breaches-and-attacks/aws-data-leak-uk-consulting-exposures/, Jan 2020.

[42] Lydian. Court of justice of the european union allows reverse engineering to correct errors. https://www.lexology.com/library/detail.aspx?g=f5b1193c-f423-4f96-bca5-03f5145ecf15, Oct 2021.

[43] Michael Meli, Matthew R McNiece, and Bradley Reaves. How bad can it git? characterizing secret leakage in public github repositories. In *NDSS'19*.

[44] Microsoft. Azure notification hubs - android sdk. https://github.com/Azure/azure-notificationhubs-android.

[45] Microsoft. Introduction to azure blob storage. https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction.

[46] Microsoft. Microsoft azure. https://azure.microsoft.com/en-us/.

[47] Microsoft. What is azure notification hubs? https://docs.microsoft.com/en-us/azure/notification-hubs/notification-hubs-push-notification-overview.

[48] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. Do windows users follow the principle of least privilege? investigating user account control practices. In *SOUPS'10*.

[49] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. Uipicker: User-input privacy identification in mobile applications. In {*USENIX*} *Security'15*.

[50] Antonio Nappa, Zhaoyan Xu, M Zubair Rafique, Juan Caballero, and Guofei Gu. Cyberprobe: Towards internet-scale active detection of malicious servers. In *NDSS'14*.

[51] NoMore201. googleplay-api. https://github.com/NoMore201/googleplay-api.

[52] The Parliament of the Commonwealth of Australia. Copyright amendment (computer programs) bill 1999. https://www.legislation.gov.au/Details/C2004B00467/Explanatory%20Memorandum/Text.

[53] Facundo Olano. google-play-scraper. https://github.com/facundoolano/google-play-scraper.

[54] Charlie Osborne. Us dmca rules updated to give security experts legal backing to research. https://www.zdnet.com/article/us-dmca-rules-updated-to-give-security-experts-legal-backing-to-research/, Oct 2016.

[55] Charlie Osborne. V shred data leak exposes pii, sensitive photos of fitness customers and trainers. https://www.zdnet.com/article/v-shred-data-leak-exposes-pii-sensitive-photos-of-fitness-customers-and-trainers/, Jul 2020.

[56] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. {WHYPER}: Towards automating risk assessment of mobile applications. In {*USENIX*} *Security'13*.

[57] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *CCS'14*.

[58] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS'09*.

[59] Gopal Sathe. 80tb of parler posts, videos, and other data leaked by security researchers. https://gadgets.ndtv.com/social-networking/news/80tb-parler-posts-photos-videos-leaked-by-security-researchers-law-enforcement-can-use-to-identify-january-6-attackers-2350920, Jan 2021.

[60] Alex Scroxton. Leaky aws s3 bucket once again at centre of data breach. https://www.computerweekly.com/news/252491842/Leaky-AWS-S3-bucket-once-again-at-centre-of-data-breach, Nov 2020.

[61] Tanusree Sharma, Hunter A Dyer, Roy H Campbell, and Masooda Bashir. Mapping risk assessment strategy for covid-19 mobile apps' vulnerabilities. In *Intelligent Computing*, pages 1082–1096. Springer, 2022.

[62] Vibha Singhal Sinha, Diptikalyan Saha, Pankaj Dhoolia, Rohan Padhye, and Senthil Mani. Detecting and mitigating secret-key leaks in source code repositories. In *MSR'15*.

[63] Michael M Swift. Least privilege via restricted tokens, October 23 2001. US Patent 6,308,274.

[64] Kirk Teska. Trade secrets 101. https://www.asme.org/topics-resources/content/trade-secrets-101.

[65] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. Webjail: least-privilege integration of third-party components in web mashups. In *ACSAC'11*.

[66] Arne Vidstrom. The legal boundaries of reverse engineering in the eu. https://vidstromlabs.com/blog/the-legal-boundaries-of-reverse-engineering-in-the-eu/.

[67] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *SIGMETRICS'14*.

[68] Sumeet Wadhwani. Shinyhunters leak 2.28m dating site users' personal info online. https://www.toolbox.com/security/data-breaches/news/shinyhunters-leak-2-28m-dating-site-users-personal-info-online/.

[69] Haoyu Wang, Yuanchun Li, Yao Guo, Yuvraj Agarwal, and Jason I Hong. Understanding the purpose of permission use in mobile apps. *TOIS'17*.

[70] Jice Wang, Yue Xiao, Xueqiang Wang, Yuhong Nan, Luyi Xing, Xiaojing Liao, JinWei Dong, Nicolas Serrano, Haoran Lu, XiaoFeng Wang, et al. Understanding malicious cross-library data harvesting on android. In {*USENIX*} *Security'21*.

[71] Haohuang Wen, Juanru Li, Yuanyuan Zhang, and Dawu Gu. An empirical study of sdk credential misuse in ios apps. In *APSEC'18*.

[72] Zhaoyan Xu, Antonio Nappa, Robert Baykov, Guangliang Yang, Juan Caballero, and Guofei Gu. Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis. In *CCS'14*.

[73] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *CCS'14*.

[74] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. Harvesting developer credentials in android apps. In *WiSec'15*.

[75] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *S&P'19*.

Table 5: Different types of credentials and API parameters extracted from apps.

Cloud Service	String Type	String Name	# Strings	# Apps	Sample String
AWS	Credential	AccessKey	907	907	AKIA*
		SecretKey	1,183	1,183	dPS6*
		IdentityPoolId	7,830	7,830	us-west-1:01234567-89ab-cdef-0123-456789abcdef
		AccountId	2,726	2,726	123456789012
		UnauthRoleArn	461	461	arn:aws:iam::123456789012:role/Cognito_TestUnauth_Role
		AuthRoleArn	2,399	2,399	arn:aws:iam::123456789012:role/Cognito_TestAuth_Role
	API Parameter	BucketName	36,696	6,247	test-bucket
		PlatformAppArn	3,269	2,910	arn:aws:sns:us-east-1:123456789012:app/GCM/TestApp
		Region	10,425	10,425	us-east-1
		TopicArn	1,076	262	arn:aws:sns:us-east-1:123456789012:TestTopic
Azure	Credential	BlobConnectionString	118	118	DefaultEndpointsProtocol=https;AccountName=*;AccountKey=*
		NotificationConnectionString	667	664	Endpoint=*;SharedAccessKeyName=*;SharedAccessKey=*
	API Parameter	ContainerName	162	94	user-pictures
		NotificationHubName	209	191	test-hub
Alibaba	Credential	KeyId	77	75	LTAI*
		KeySecret	77	75	EExRxev*
	API Parameter	BucketName	121	112	test-bucket
		Endpoint	235	163	http://oss-cn-beijing.aliyuncs.com

Table 6: Examples of patched mobile apps.

App Description	Cloud Service	Vuln.	Affected User Data	Installs
FM radio app A	AWS SNS	P2	App notifications	179,000
School automation system app	AWS S3	P2, P3	Images and videos from parents and children	100,000
FM radio app B	AWS SNS	P2	App notifications	25,000
Ride-hailing app	AWS S3	P2, P3	User bank details, user vehicle images	10,000
Social networking app for anglers	AWS S3	P2, P3	User profile images, fishing pictures	10,000
Sports radio app A	AWS SNS	P2	App notifications	10,000
Workforce monitoring app	AWS S3	P2, P3	Photos uploaded by employees	10,000
Smart teleprompter app	AWS S3	P2, P3	Text document of a speech or script	10,000
Private social networking app	AWS S3	P2, P3	Photos/videos/text messages among neighbors	10,000
Emergency alert paging app	Azure Blob	P2, P3	Incident information (e.g., site images)	5,000
Construction project mngmt app	AWS S3	P2, P3	Field images, expense images	5,000
Auto dealer mngmt system	AWS S3	P2, P3	Images/Vidoes of vehicle inspection	5,000
Live-action sports app	AWS S3	P2, P3	User profile images	5,000
FM Radio app C	AWS SNS	P2	App notifications	3,000
Ride-sharing app	AWS S3	P2, P3	User profile image, drive/vehicle documents	2,000
Sports radio app B	AWS SNS	P2	App notifications	2,000
Church app	AWS S3	P2, P3	User profile image, church event videos	1,360
Home inspection mngmt app	AWS S3	P2, P3	Home inspection videos, expense images	1,000
Taxi booking app	AWS S3	P2, P3	User profile image, driver signup documents	1,000
List building app	AWS S3	P2, P3	User created lists (e.g., links, media)	1,000

Table 7: Cloud service APIs and capabilities.

Cloud Service	Capability (Cloud API)	Resources	Apps That Require Capability	# Apps That Have Extra (Leak) Capability
AWS S3	ListAllMyBuckets	*	92	1,025
	ListBucket	Bucket	539	1,268
	GetObject	Bucket + Object	2,384	1,022
	PutObject†	Bucket + Object	5,820	933
	DeleteObject	Bucket + Object	639	1,625
AWS SNS	ListPlatformApplications	*	14	413
	ListEndpointsByPlatformApplication	*	7	173
	ListTopics	*	41	389
	ListPhoneNumbersOptedOut	*	1	360
	DeleteTopic	Topic‡	10	352
	ListSubscriptions	*	9	383
	ListSubscriptionsByTopic	Topic	8	364
	CreatePlatformEndpoint	*	1,183	43
	CreatePlatformApplication	*	13	356
	DeleteEndpoint	*	92	168
	Publish†	Topic	75	285
	Subscribe	Topic	765	307
	Unsubscribe	*	491	367
	DeletePlatformApplication	*	12	172
Azure Blob Storage	ListContainers	*	1	46
	ListBlobs	Container	37	34
	Download	Container + Object	36	30
	Upload†	Container + Object	130	7
	Delete	Container + Object	5	39
Azure Notification Hub	Register	*	1,035	7
	SendNotification	*	0	110
Alibaba Cloud OSS	ListBuckets	*	10	27
	ListObjects	Bucket	112	19
	GetObject	Bucket + Object	387	16
	PutObject†	Bucket + Object	75	3
	DeleteObject	Bucket + Object	58	19
Alibaba Cloud Message Service	CreateQueue	*	3	0
	CreateTopic	*	0	0
	PutMessage	Queue	0	0
	PopMessage	Queue	0	0

† `Put`-like APIs are evaluated based on cloud response exceptions, and they will not introduce new objects in cloud services.

‡ `DeleteTopic` permission is specific to a topic, but we are only able to test it for all resources for security consideration.