Assessing the Effectiveness of the Shared Responsibility Model for Cloud Databases: the Case of Google's Firebase

Biniam Fisseha Demissie
Fondazione Bruno Kessler
demissie@fbk.eu

Silvio Ranise
Fondazione Bruno Kessler and University of Trento
ranise@fbk.eu

***Abstract*— Migrating databases to the cloud requires the adoption of the shared responsibility model for protecting data. The database-as-a-service provider secures the database from different kinds of attacks while the developer defines the access control policy to prevent unauthorized access. Recent reports show that developers fail to properly secure their cloud databases leading to sensitive data leaks. In this paper, we investigate the prevalence of the access control misconfigurations in 50K+ top Android apps that use one of the most popular cloud database services, namely Firebase. Overall, we found 763 apps (1 billion downloads) with public databases and 536 apps (630 million downloads) with world-writable databases. Considering the popularity of these apps and the cross-platform nature of Firebase databases, our findings reveal a worrying state in the adoption of the shared responsibility model for the security of cloud databases. To assist developers, we make our prototype tool publicly available as an Android Studio plugin. The plugin performs static analysis to automatically extract Firebase database information from the app under development and checks its configuration status.**

I. Introduction

With currently more than two billion devices running it, Android is the most popular mobile operating system in the world [38]. It is used by devices ranging from smartphones and tablets to wearables, smart TVs, and automobiles. The proliferation of Android devices has created a good business opportunity for novice developers to create and distribute applications (herein apps) easily using the centralized Android market, called Google Play, targeting millions of users. Google Play is now the biggest app store in the world, and as of September 2020, it has more than 3 million apps available for download [39].

To provide valuable functionalities to their users, most apps rely on either local or remote databases. When apps involve users (e.g., social apps) or have a continuously changing content (e.g., news apps), they rely on remote databases. Developers can either set up their own remote database servers or integrate a cloud database service from one of the many providers.

According to Gartner Inc., by 2022, 75% of all databases will be deployed or migrated to a cloud platform [20]. More and more developers are moving to cloud databases for their web and mobile app development. Google is one of the many cloud service providers with different cloud databases.

Google's Firebase cloud service is one of the most popular among app developers with notification, analytics and database services among others. The fact that it comes integrated with Android Studio, the official development environment for Android, and its ease of configuration makes Firebase the obvious choice for Android developers. Though developers can deploy Firebase database instances easily, enforcing the appropriate access control policy is a non-trivial task and developers often make mistakes and expose their cloud database instances [8].

Cloud databases adopt the shared responsibility model where the service provider ensures security of the cloud (e.g., infrastructure, virtualization, communication, etc.) and the developer ensures proper access control policies to avoid unauthorized access to sensitive data [4]. Though the service provider side is secure enough, misconfiguration of the access control policy, however, could lead to data leak.

The first report regarding access control misconfiguration of Firebase databases was seen in mid-2018 where 113GB of data over 2,271 databases from thousands of Android apps containing millions of user's data have leaked [42]. Several months later, the then-new and popular dating app called Donald Daters for President Trump supporters, has leaked its entire database on the first day it was launched containing user sensitive data including chat conversations [40]. A year later, Twitter's popular video app Periscope misconfigured its database that gave write privilege to anonymous users [21]. Moreover, at the time this security misconfiguration was reported to Twitter, their security team did not see the impact of having a publicly writable database (however, they have later acknowledged and fixed the security issue).

Later, Google announced on Firebase Summit that they have improved their Firebase console to warn developers of the possible misconfiguration of their database [17] and send email notifications to the developers. With all these incidents making headlines and the improvements Google is continuously doing, one would assume that the problem is no longer relevant. A recent report [8], however, shows even more apps failing to properly configure their databases.

In this paper, we assess the effectiveness of the shared responsibility model by taking the Firebase cloud database service as a case study. We first investigate Firebase database security and the peculiar access control system features it implements. We then walk through the steps a developer usually takes to develop a simple app to better understand where the security misconfiguration may occur. Different from previous nonscientific reports ([8], [42]), instead of taking a random snapshot of current apps, we investigate Firebase database misconfigurations among *top* Android apps from 2016, 2019

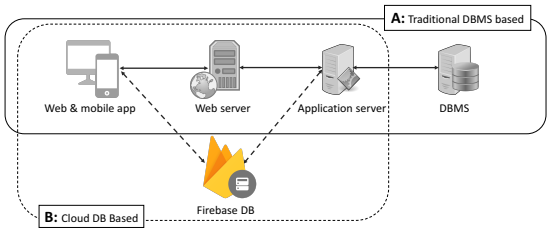

Fig. 1. Applications with (A) traditional DBMS and (B) with Firebase cloud database

and 2020. Top apps are downloaded by hundreds of millions of users and since they are recommended by Google Play, they are believed to be of better quality. Security issues that are common among top apps would require special attention as they affect millions of users. While the previous reports looked at potential data leaks only, we assess database access control misconfiguration including their world-writability. Finally, in order to help developers mitigate the Firebase database security issues, we propose a static analysis tool as an Android Studio plug-in that assists developers in checking the accessibility of their Firebase database from their development environment.

Our investigation shows that 763 apps with close to 1 billion downloads have public databases exposing sensitive user data, while 530 apps with more than 630 million downloads have world-writable databases that could potentially be used to distribute malware.

The contributions of this paper consist in the:

- analysis of the characteristics of Firebase cloud database access control policy language in relation to access controls;
- building of an automated tool to statically analyze compiled Android apps and perform a security check on their Firebase databases.
- the empirical assessment of our tool by analyzing more than 50K top Android apps on Google Play from three different time periods.
- an open-source static analysis tool as an Android Studio plug-in that supports app developers to easily check their database's access during app development;

The paper is organized as follows. After covering the background on web and mobile app architecture from database and security point of view in Section II, security assessment of mobile apps that use the Firebase database and the methodology we used is presented in details in Section III. Then, Section IV presents results of empirical assessment of our approach followed by discussions, recommendation, ethical considerations and tool support. After discussing related work in Section VII, Section VIII concludes the paper.

II. Background

In this section, we present some background notions used in the rest of the paper.

A. Web and app architectures

A traditional multi-tier architecture for the web describes the separation of different component groups in a client-server [19]. Figure 1(A) shows an example of a traditional web architecture organized in distinct tiers. The presentation tier is responsible for displaying information in a structured way while the application logic tier processes user queries and either reads or writes to the persistence tier and responds with the appropriate data. Note that the presentation tier does not have direct access to the persistence tier. In this architecture, the database management system (DBMS) is often secured by the service provider (e.g., the hosting company). The developer has accesses to the database via some kind of console while clients do not have direct access.

Figure 1(B) shows an architecture using the Firebase cloud database in place of a traditional DBMS. As it is evident from the figure, the presentation tier can directly interact with the persistence tier. This means that read, write or update requests can directly be sent from the presentation tier to the persistence tier. In this architecture, since the client has direct access to the database, the developer is responsible to properly configure the database access control in order to avoid abuse from untrusted clients. If the database is not properly configured, user sensitive data could be stolen, modified or deleted by malicious users [42].

In cloud databases, tenants are handled at the level of projects where databases of different applications are hosted in the same logical domain and are tagged with unique identifiers (project IDs in this case). In the multi-tenant model, the responsibilities for maintenance and establishment of secure database environment rely solely on the cloud provider. This could give developers the wrong impression that everything regarding security is handled by the cloud service provider where in reality the responsibilities are shared.

Below (in Section II-B), we introduce Firebase databases that are instances of cloud databases. Later (Section II-C), we present how access control policies can be specified to secure Firebase databases. We then provide a motivating example (in Section II-D) that highlights some of the challenges developers might face when using Firebase databases.

B. Firebase Database

Firebase is a mobile and web application development platform that provides database services such as Realtime Database and Cloud Firestore. Data is stored in JSON format and synchronized in real-time to every connected client. Apps developed either for iOS, Android or web share one Realtime Database instance and automatically receive updates with the newest data.

In general, we can abstract the operations (requests) in Firebase databases into two general operations, namely read and write. Hence, if no granularity is needed, we can simply allow/deny read or write operations. *Read* requests allow apps to either get one specific document or a list of documents. A misconfiguration of access control rules for *read* operation might lead to sensitive data leaks (e.g., if the database contains

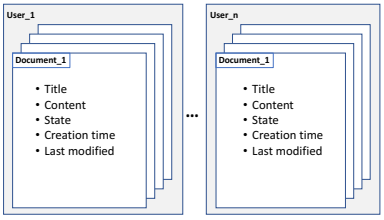

Fig. 2. Non-relational data structure for a note-taking app

personal identifiable information or PII). *Write* requests allow apps to create, update or delete a document (or documents). A misconfiguration of access control rules for *write* operations may have severe consequences as it allows attackers to modify contents (e.g., advertisement IDs, user posts or comments, app configurations), insert contents (e.g., hosting illegal content anonymously), or delete/wipe the entire data.

C. Access Control Features of Security Rules

We now discuss how Security Rules implement some of the desiderata an access control service should have [10].

- **Conditions and support for abstractions**: Firebase Security Rules for Firestore and Cloud Storage support conditions that the developer can specify (e.g., request time, email verified, etc.). Since there is no catalog of abstractions that can be used, it is entirely up to the developer to introduce an appropriate set of attributes that are needed to define the desired abstraction. Since the developer has to manage users (subjects) and data (objects), he/she can include roles, groups or other relevant attributes to users and data collections to achieve abstraction.
- **Positive and negative authorizations**: When database Security Rules are not specified or when multiple Rules are specified for a given path, the access control system should use the appropriate conflict resolution strategies among those available in the Firebase databases.
  - **Closed vs open policy**: All Firebase databases adopt a deny by default strategy and a developer has to write specific `allow` rules to grant access. For example, `allow read;` will allow read access to all documents in Cloud Firestore under a given path.
  - **Incompleteness vs inconsistency**: If no authorization is specified, by default access is denied. On the other hand, if both allow and deny (positive and negative authorization) are specified, the allow will take precedence during conflict resolution. According to the documentation, Security Rules are intended in disjunction and not in conjunction. Consequently, if multiple rules match a path, and any of the matched conditions grants access, Security Rules grant access to the data at that path.

D. Motivating Example

To better understand the security challenges app developers might face when moving from a traditional database to a cloud-based non-relational database such as Cloud Firestore, we consider the development of an app for taking notes. The development is inspired by a real-world scenario [16] and highlights the main issues a developer has to face when defining security policies to secure the app deployment.

Let us start by designing the data structure representing notes and users. Each user can only add, edit or delete his/her own notes and should not have access to other users notes. This implies that users have to be authenticated and authorized to access content. One way that we can design the data structure is as shown in Figure 2. Every user will have zero or more documents (notes). Every document will have basic fields (attributes) such as title, content, creation time and state (e.g., a flag to mark deletion).

In order to have user-based access system that keeps users' data safe, we first need to implement authentication. This could be achieved by using Firebase Authentication with, e.g., OpenID Connect [35]. Assuming this has already been implemented in our app, we then proceed to securing our database.

There are several ways one can secure the database. One obvious way is represented by the Security Rule in Listing 1.

Listing 1. Security Rule for a note taking app

```
1  service cloud.firestore {
2    match /{collection}/{documents=**} {
3      allow read,create,update: if
           request.auth.uid == collection;
4    }
5    match /{documents=**} {
6      allow read, write: if false;
7    }
8  }
9  }
```

The rule allows read, create and update operations (line 3) to any document (note) in a given `collection` (line 2) as long as the collection matches the user ID requesting the operation. For example, for the path `/User_1/Document_1`, only user with ID `User_1` is allowed to perform the read, create and update operations to document `Document_1`. For any other document (line 5), it blocks read/write access (line 6) to everybody. Note that the request coming from the client (web or mobile app) is properly signed and Firebase's authentication service (called Firebase Auth) is responsible for verifying whether the request is coming from the right user. Therefore, one cannot spoof the user ID in order to access someone else's document.

We observe that, in general, the Security Rules highly depend on the data structure that developers define at the beginning of the design. If the data structure changes during development, developers might also need to adapt the access control configuration accordingly. Let us now assume, for example, that we want to add the possibility to share a note with other users. For users to be able to see other's notes, we can first modify the note data structure to have a "public" field (e.g., {`"public" : true`}) that marks whether a note is made public. We then modify the rule on line 3 of Listing 1 to include a check if the specific document has the public field set as shown in Listing 2 below. This code allows access to the specific document if the request comes from the owner or

if the document is public.

Listing 2. Updated Security Rule to support note sharing

```
...
allow read,create,update: if request.auth.uid ==
    collection || resource.data.public == true;
...
```

For ease of development and debugging, we might be tempted to disable the access control altogether. In fact, as acknowledged by Google in the Firebase Summit [17], the common development pattern is making the database initially public, develop a working initial app, then secure the database and publish the app. However, developers often forget the fact that their database is public/open by the time they ship their app. Moreover, depending on the complexity of the app and the underlying data structure, even if a developer attempted to secure the database, it is still possible to make configuration mistakes.

In general, developers may find it difficult to define Security Rules that adequately capture the security goals of their apps because of several reasons. One of the most important reason is related to the paradigm shift from traditional database systems (such as MySQL) to NoSQL [27] cloud databases (such as Cloud Firestore) where access control policies are applied at the level of paths instead of tables. This might become even more complicated when policies involve hierarchical paths.

A malicious user can easily extract misconfigured Firebase database information from web or mobile apps and dump the entire database potentially containing sensitive user data or even wipe the entire database.

Even if a developer is aware of the security issues associated with Firebase databases and attempts to define some Security Rules, app complexity could still lead to data leaks due to misconfigurations.

In this paper, we investigate the trend in Firebase database access control misconfigurations in top Android apps downloaded from Google Play in 2016, 2019 and 2020.

III. Security Assessment of Mobile Apps that use Firebase Databases

In order to investigate if developers are applying the right configuration to their Firebase cloud databases, we designed and deployed a large scale assessment of top Android apps ("top_selling_free" on Google Play) downloaded in three different time periods. Even if Firebase databases can also be used in web apps, considering its popularity among Android app developers, we decided to investigate the misconfigurations on Android apps. To this end we developed a static analysis tool that analyzes an Android app, extracts database information and checks for misconfiguration.

The fact that top apps are suggested by the official store makes us assume that these apps are presumably of certain quality. If security issues are very common in these top apps, we can assume that less popular apps, developed by less professional developers or hobbyists might also have similar security issues.

Since database information (endpoint) can statically be extracted from an Android app, API security testing of the endpoint would tell us if the database is configured to allow *read*, *write* or both. Though there are use cases where a developer might need to give *read* access to any client (e.g., access to app configuration information), *write* access to unauthenticated client is most likely a misconfiguration.

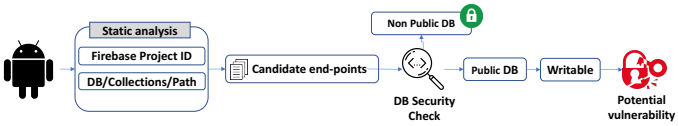

Fig. 3. Static analysis workflow

The lack of awareness and better communication/tool from cloud service providers is likely to give developers the false impression that security is handled by the cloud service provider that provides the database-as-a-service. Though the cloud service provider protects the developer data from different kinds of attacks, the split responsibility model makes the user of the database service (i.e., the developer) responsible for defining the appropriate access control policy (i.e., Security Rules) in order to secure the database deployed in the cloud.

In this section, we introduce the static analysis tool that we have designed and developed to perform a large scale assessment of Firebase database misconfigurations of more than 50K popular Android apps downloaded from Google Play.

A. Static analysis

We now describe the static analysis technique that we have designed and implemented to extract Firebase database details from Android apps and check for access control misconfiguration. We then show how it can be used to perform a large scale analysis on apps downloaded from Google Play.

Figure 3 summarizes the workflow. Given an Android app, the first step is to infer whether the app uses Firebase. If the app uses Firebase, we extract the database location (project ID). We then extract possible paths from the app source code. Paths represent collections or documents. Once we collect database location and paths (candidate end-points), we can query and establish if the database content can be exfiltrated or overwritten labeling the app potentially vulnerable or secure otherwise. In the following, we describe each step in details.

Firebase Project ID Extraction: We implemented a static analysis tool that, given an Android app package (APK), it first uses the Linux `strings` command to check reference of the string `firebaseio.com` on the entire app (i.e., unlike previous approaches such as [8] not only limited to XML resources but also bytecodes). The domain is where Firebase projects reside and hence its reference is a good indicator of usage. This is the fastest way to check whether an app uses Firebase without decompiling it. The output of this step is a URL pointing to the project endpoint. If no reference is found, we decompile the app and check the `res/values/string.xml` resource for existence of the `project_id` field. An example Realtime Database endpoint for a project with ID `my-project` is

`https://my-project.firebaseio.com`

Collection (Path) Extraction: If the app uses Firebase, then we perform further analysis to extract possible *collection* or *document* names. Collection or document names give us the different databases (e.g., `Users` database) under the given project. Document or collection name extraction requires a more advanced static analysis on the bytecodes to extract the string constants used as `path` on actual parameters of database function calls such as `getReference(path)` or `child(path)`. The process starts by identifying call-sites to the interesting database functions (i.e., those used to get references to databases or collections [15]). Starting from each call-site, we perform backward data dependence on the register that holds the actual parameter until we reach the definition. If the definition is a constant assignment, we collect the constant as a potential path. The output of this step is a list of URLs constructed by combining the project URL with the collected paths.

Database Security Check: Once the Firebase database URL and potential databases or collections are extracted, the next step is to perform a query to Firebase endpoints to determine first, if we can dump the entire database (i.e., if the database is public for anonymous users) and second, check if the databases are world-writable. We expect two major outcomes from this query:

- **Database is not public.** In this case Firebase would return `{"error" : "Permission denied"}` (or HTTP error code 403). If the query is to a given path (e.g., /cities), the error response could mean either the path is not public or it does not exist at all. Note that databases can still be writable as the read and write operations are protected by two different Security Rules.
- **Database is public**. A public database that is either read-only or world-writable. We need to consider four cases: a public database
  - does not have an entry. In this case the response to the query would be `null`. If the query is to a given path (e.g., /cities), the response could mean the given database (i.e., citites) is actually empty or it does not exist at all.
  - contains data that is intentionally made public by the developer (e.g., configurations).
  - contains sensitive data (e.g., login credentials, user personally identifiable information) that is made public because of a broken access control configuration. An attacker can simply dump the entire database without any authentication.
  - is *World-Writable*. An unauthenticated user can create/modify content.

In order to classify an app as vulnerable or not, careful manual evaluation of the appropriateness of each of the above cases should be performed. For example, not all public databases are vulnerable. A developer could intentionally make a database public if it contains app configuration information that should be accessible by all clients. A world-writable database, however, requires a careful developer inspection as it could easily be used by malicious users to store and share illegal contents.

Note that in some cases, when the public data is too large, dumping the entire database at once will fail with the error message `"error" : "Payload is too large"`. In such cases, a recursive shallow query might be required in order to dump the entire database. A shallow query limits the depth of the response and returns just the top-level data-structure. For example, if a database contains `users` and `messages` collections, a shallow query would return the following response without the contents in the collections.

```
{
   "users": true,
   "messages":true
}
```

From this response, an attacker can learn that the database contains `users` and `messages` collections and can attempt to dump each one separately. If any of the attempts fail, further shallow queries can be performed on the collection to get the top-level data-structure and repeat the same process until the entire data is dumped.

An example query to check if the entire Realtime database is public for a project with ID ”my-project” is to perform the following simple GET request where the variable `$path` is empty:

```
wget "https://my-project.firebaseio.com/$path.json"
```

On the other hand, if we want to check the path `/users`, we set the variable `$path` to ”users” and perform the GET request. We iterate through the list of potential path strings that we statically collected and check whether the paths have public databases.

In order to check whether a database is world-writable, we carefully crafted a script that inserts and then removes a unique entry without affecting the existing data. We first verify that the database does not contain the unique entry before insertion attempt. Insertion is a PUT request while deletion is a DELETE request to the endpoint.

We script the whole process and perform the analysis on the entire popular app dataset. At the end of the analysis, for each app, the static analysis tool produces output with the following contents: Firebase database endpoint(s), world-writability status, and a dump of shallow query results and a dump of the entire database if public.

IV. Evaluation

In this section, we evaluate our approach and the state of the shared responsibility model for Firebase cloud database security with some experiments. Though we investigated top Android apps, Firebase is a cross-platform service used across multiple platforms such as iOS and web. The goal is to understand whether popular (presumably good quality) apps are protecting their cloud database properly. This would give us the general picture of the Firebase cloud database security of more than 3 million Android apps on Google Play published by developers with different expertise. The empirical evaluation is guided by the following research questions.

- RQ_1: How common is for the top apps using Firebase to have public database?
- RQ_2: How common is for top apps to make their Firebase database world-writable?
- RQ_3: What is the impact of database misconfigurations in top Android apps?

The first research question RQ_1 investigates how many popular apps have their database publicly accessible without any authentication. A public database could be empty or could contain some data such as app configurations or users' sensitive data. Indeed, an app is considered vulnerable if its public database contains sensitive data. The second research question RQ_2 investigates how many of the popular apps have their database world-writable. Though there are few use-cases where a database might require world-writable access control policy for anonymous users (e.g., comments or feedback feature for all users), in most cases, it is an access control policy misconfiguration. With the last research question RQ_3, we investigate the impact of the database access control misconfiguration. Considering that top apps have millions of downloads, a policy misconfiguration in these apps could impact millions of users.

A. Subject Apps and Experimental Settings

For our study, we considered top Android apps from three different time periods. Note that these are all the apps that Google Play offered at the time of the download:

- *DataSet$_1$: Top apps in 2016.* We collected all the available top Android apps in July 2016 from Google Play. This dataset contains around 14K apps spread across 29 categories.
- *DataSet$_2$: Top apps in 2019.* We collected all the available top Android apps in February 2019 from Google Play. This dataset instead contains around 24K apps spread across 57 categories.
- *DataSet$_3$: Top apps in 2020.* We collected all the available top Android apps in February 2020 from Google Play. This dataset contains around 11K apps spread across 58 categories.

Unlike previous reports [8] that perform similar investigation on a snapshot of random apps, we resorted to top apps from three different time periods. This dataset choice is motivated by the fact that (i), top apps are recommended by Google Play and are believed to be well accepted by users, and (ii), the different time period gives us the trend in Firebase usage and misconfigurations. Weaknesses in these apps would potentially affect millions of users.

Note that availability of these apps for download depends on the kind of Android device we used for downloading (e.g., architecture or Android version compatibility) and the location (e.g., some apps not being available in some geographic region).

While extraction of Firebase related details from a given app is quick (less than a minute in the worst case), dumping database content depends on network connectivity speed and size of the database. During this analysis, we did not observe any throttling or blockage from Firebase servers. Requests were made via `wget` without any attempt to spoof the user-agent or add request headers to bypass any limit.

Results are confirmed based on standard error codes (200, 403, 404...etc) and response body (e.g., `null`) when read-/write operations are performed.

The analysis has been conducted on a machine equipped with an Intel Core i7-8700, 3.2 GHz processor with 16 GB of RAM running Ubuntu 18.04 LTS.

RQ_1: How common is for the top apps using Firebase to have public database?

In order to answer RQ_1, we queried the Firebase database endpoint that we statically extracted from the different datasets. We then perform an HTTP GET request to these end-points to get shallow data (i.e., limits the depth of the response and returns just the top-level data-structure). If our query returns HTTP status code 200, we conclude that the apps database is public. Column 4 of Table I shows the result of the analysis. For $Dataset_1$, 8.4% of the apps using Firebase have their database public. Out of the 6789 apps using Firebase in $Dataset_2$, 6.9% of the apps have their database publicly accessible. In 2020, however, even if almost 50% of the apps are using Firebase, the percentage of apps with publicly accessible database is lower than previous years with only 4.7% of the apps having public database. Note that this analysis does not differentiate between public databases with and without data (i.e., `null` data).

> It is common to find public database even in top apps. Since these are popular apps with millions of downloads, even having fewer percent of the total apps having public database could affect millions of users.

RQ_2: How common is for apps to make their Firebase database world-writable?

To see if it is common to find apps with world-writable database, we considered the apps with public database. Similar to RQ_1, we used the statically extracted database endpoints to perform HTTP PUT request with carefully crafted unique input (see Section III-A) and verify whether it was successful. Note that the inserted unique content does not interfere with existing data and app functionality. Once we verify that the operation was successful, we remove the inserted data and conclude that a write operation is permitted by the access control policy. We ran the analysis for all the apps with public database and the results are presented in column 5 of Table I. For $Dataset_1$, all the apps that have public database also have their database world-writable. This could be because these apps were early adopters and were not aware of the security implication of their open database. Another reason could be the lack of resources to learn about proper configuration of the access control policy. In $Dataset_2$, 71% of the apps with public database have also world-writable database. Even if there is a three-year gap between $Dataset_1$ and $Dataset_2$, the security of the databases did not improve much. This can also

Dataset	Total Apps	Apps Using FB (% of total apps)	Public DB (% of apps using FB)	World-Writable DB (% of public DB)
$DataSet_1$	14213	381 (2.68%)	32 (8.4%)	32 (100%)
$DataSet_2$	24432	6789 (27.8%)	471 (6.9%)	335 (71%)
$DataSet_3$	11585	5537 (47.8%)	260 (4.7%)	169 (65%)

TABLE I
SUMMARY OF USAGE OF FIREBASE IN ANDROID APPS ACROSS MULTIPLE YEARS

be seen for the latest dataset, namely $Dataset_3$ in which 65% of the apps having public database also have world-writable database.

A writable database is very dangerous. In fact, a quick grep with regex in the dump shows that there are more than 88 Google Play URLs and in general more than 180K URLs. An attacker that is able to modify these URLs could potentially be able to distribute malware reaching hundreds of million of users.

> It is common to find popular apps having a world-writable database. Apps with public database are also most likely to have their database world-writable.

RQ_3: What is the impact of database misconfigurations in top Android apps?

To answer RQ_3, we focused our analysis only on the recent dataset, namely, $Dataset_3$ because it is the dataset that reflects the current real-world popularity. To this end, we collected download meta-data of the apps with world-readable and writable databases. We consider the number of downloads as a proxy for popularity. We acknowledge that the number of downloads does not necessarily reflect current active users. However, even if a user uninstalls an app or an app is removed from Google Play, the user data collected by the app could still remain in the Firebase cloud database at the discretion of the app developer.

Our analysis on world-writable databases from $Dataset_3$ shows that the number of downloads for these apps ranges from 100 to 100M with average downloads of 2.9M. The total number of downloads for all the apps with world-writable database is, therefore, more than 630M.

On the other hand, the number of downloads for apps with world-readable databases in the same dataset ranges from 1K to 50M with average downloads of 2.4M totalling 275M downloads. Note that only public databases containing data were considered in this analysis as empty world-readable databases do not pose significant security issues.

With this, we can answer RQ_3 stating:

> Apps with world-readable and writable databases are very popular with over 275M and 630M downloads, respectively, potentially impacting millions of users.

B. Discussions

The above results show us our proposed static analysis based approach is effective in identifying Firebase databases misconfigurations. Indeed, we were able to find more than 700 apps with database misconfigurations. We can also observe from the results that the shared responsibility model for securing Firebase cloud databases appears to be ineffective and needs some improvement either by the cloud service provider (in this case Google) or the developers. The service provider can offer a better access control policy enforcement mechanism (e.g., easier way to generate policy rules) and limiting access when a database has a peculiar behavior (e.g., a sudden spike of `list` traffic when `get` was the previously observed traffic). On the other hand, developers can perform security testing of their apps before shipping to improve the privacy and security of end-users.

Developers might be using different email for their cloud database service and therefore, any communication regarding their insecure database could be missed. A developer assistance should, therefore, be closer to the development environment where the developer gets a warnings, for example, during app build time.

We would like to emphasize that the results presented in this paper are not retrospective, meaning that even if our dataset contains top apps from 2016 or 2019, several weeks after our initial analysis, most of the reported vulnerable databases are still active to date and are exploitable. Moreover, our investigation focused only on requests originating from anonymous clients. Database access control misconfiguration for authenticated clients is out of the scope of this work and is considered as future work.

There are several use cases that require a Firebase database to be publicly accessible. Therefore, we cannot claim every public database is vulnerable. One simple example where a developer might need a public database is with advertisement configuration that both anonymous and authenticated users should be able to access so that the app can show ads. However, an attacker could still attempt to exploit a public database in the following ways. The Firebase database service provides a free and payed service. For the free version, developers have few gigabytes of traffic per month for free and have to upgrade the service if the traffic reaches the quota, otherwise the service will be interrupted until the next reset time. Depending on how large the public data is, an attacker can perform multiple requests exhausting the available quota for a free service user or costing money for a paid service user. This could lead to denial-of-service in case of service interruption or costs money for a pay-as-you-go developer.

Depending on the use-case, this situation could be mitigated by denying database `listing` and allowing only `get` operations where a client requests one document (of smaller size) at a time.

There are also uses cases where a database needs to be world-writable. For example, in order to collect feedback from all users, a database should allow write operation to both anonymous and authenticated users. This operation should not allow the users to update/delete existing data. However, in most use cases, writing requires authentication. Therefore, a world-writable database is most likely a configuration mistake (vulnerability).

Whether it contains data or not, a world-writable database is more dangerous than a public (readable) database. In addition to data leaks, attackers could also exploit a world-writable database in many ways such as: making it a medium to share illegal content anonymously, push malware installation by modifying URLs, replace advertisement configurations and redirect revenue to the attacker or in extreme cases, delete the entire data.

In general, if we have a world-writable database, data integrity is no longer guaranteed and depending on the app, an attacker can perform different malicious operations.

The impact of security issues in Firebase databases misconfigurations is high as Firebase is used across multiple platforms. The same database can be shared between iOS, Android or web apps risking exposure of large user sensitive data. Though multi-tenancy is not recommended by Google, we have observed different apps sharing the same database project.

An important point worth mentioning is the fact that there are several apps that no longer exist on Google Play but their Firebase databases are still publicly accessible or world-writable. These databases could be abandoned by the developer or are still being used by other apps that we are not aware of. We were not able to communicate with the developers of these apps as we could not find their address.

In order to understand if the databases contained user sensitive data, we create a list of keywords derived from definition of personal data [7] so that we can grep and count the occurrences of these keywords and patterns without actually looking into the data.

A simple grep for the keywords and patterns in the public data shows that it contains more than 300K user credentials, 5K phone number, 20K IPs, 50K device IDs and 200K URLs.

These apps were downloaded from European Google Play. This means that most of these apps do not comply with GDPR policy and are collecting personal identifiable information (PII) without properly safeguarding the collected data, let alone informing their users.

In addition to statically extracting the Firebase database endpoints from apps, one can also consult the many Github repositories containing list of exposed Firebase databases. Search engine dorks that attackers use for reconnaissance can also be used to find exposed databases. Though Google scrubs search results containing such contents, using the following dork on Bing.com produces some public databases.

```
site:.firebaseio.com
```

Finally, since Firebase databases are effectively API endpoints, some of the top OWASP API security issues might apply [36]. For example, since developers cannot enforce rate limits, an attacker performing multiple requests might exhaust the developer's quota or cost money causing denial of service (OWASP API4).

Limitations: The assessment of database misconfiguration relied on static analysis of compiled code. Hence, the approach suffers from the inherent problems of static analysis techniques, such as not precisely analyzing obfuscated code. For this reason our tool might have failed to extract Firebase database details from heavily obfuscated apps.

V. Ethical Considerations and Disclosure

In this section, we discuss some of the ethical problems arising in the context of conducting our large scale analysis and the process used to disclose our findings to app developers.

The collected potentially sensitive data has been carefully handled and processed by only one person (one of the authors) and has been deleted once the study was completed.

As previously discussed (see Section III-A), not to interfere with app functionality when testing for world-writable databases, we carefully craft a unique content and verify that the content does not exist before attempting to insert it.

In order to investigate if the public databases contained sensitive data, we defined a list of keywords and regex patterns so that we can search the dump for these keywords/patterns and count the occurrences without actually looking into the content. Some of the keywords and patterns include: "password", "device*id", "imei", "email", "message", "img*url", "profile*pic", "profle*image", etc., and regex patterns for URL, phone number, IP and email. The keywords are derived from the definition of personally identifiable information (PII) in the context of GDPR.

Recall that even if a database is empty, the choice of making it world-writable may have security implications as it can be used by attackers to share illegal content anonymously. Therefore, developers interested in using other Firebase services (e.g., analytics) must pay attention how their empty database can be accessed.

Note that not every public database is vulnerable. In fact, majority of the public databases that we observed have small file-size hinting their content could be configurations rather than sensitive user-data. World-writable databases are, however, highly likely misconfigurations. We, therefore, considered the 218 top apps with non-empty world-writable databases for the responsible disclosure. For each app, we attempted to reach out to the developers via their developer mail that is listed on Google Play. All mails (except one) were successfully delivered to the respective developers. In fact, for most of the apps, we received automatic acknowledgments that the mail has been received and the responsible person will get back to us. For each developer, we automatically composed the mail including the package name, the security problem (world-readable/-writable), the implication if it contains user sensitive data (e.g., since the apps were downloaded from European Google Play, the apps should be compliant to GDPR) and timeline. Even if we do not disclose the list of apps with vulnerable Firebase databases, since our analysis is on top apps, it is easy for a reader to find and exploit these apps. To avoid this situation, we have provided the developers more than 30 days to investigate and fix the security problems.

Within two weeks, we received acknowledgments from 17 developers (7.8%) promising to fix their database security

as soon as possible. Most of the developers acknowledged that they forgot about the security of their database. Some developers asked us how to fix their database Security Rules.

Considering the liability concern (e.g., for GDPR compliance), most developers that have exposed sensitive user data might not respond to our report. In order to understand whether it is because the developers are not reading their emails; they do not maintain the app anymore or if they have silently fixed the problem, we rerun our experiment more than two months later. The results are as follows:

- 60 of the reported apps (28%) fixed the problem by either modifying the security rules (54 out of 60) or disabling their databases (the remaining 6);
- Out of the 54 apps that updated their database Security Rules, 46 apps made their entire database private (denied read/write access to anonymous clients) while 8 apps disabled only write access,
- While all the developers that acknowledged the problem (17) have fixed their databases, 43 apps fixed the security problem without acknowledging the report.

By observing how the developers fixed their databases, we can see that granting a write access to anonymous users is not a common practice. Unfortunately, 158 apps still have world-writable databases containing some data. We have also contacted Google's Firebase team regarding this issue. The response we received was that they have tried to contact the developers multiple times to no avail.

VI. Recommendations and Tool Support

One of the basic tenets of web and mobile application security is to always distrust clients. This also holds in the context of securing apps using the Firebase cloud infrastructure, especially in the light of Figure 1(B); from which it is immediate to realize the importance of configuring appropriate security rules to mitigate malicious and unfiltered clients inputs to the database.

A developer that implemented the right authorization on the client app might assume that all the access requests are coming from an authorized client. As we have seen previously, an attacker can easily extract the database information and perform a direct read/write request to the database server. If the same access control policy is not enforced on the server side, then an attacker might be able to easily perform the malicious operations we presented above.

Authenticated requests, however, can be trusted as the request signature can be verified for consistency by Firebase service.

The commonly observed trend, as also acknowledged by the Firebase team, is making the database public during development time and shipping the app without securing the database either because the developer forgot about it or the app became too complex and security configuration is postponed for later. Developers should start from a secure root database and work with less restrictive paths. This way, attackers that are performing a large scale analysis by querying the root path (e.g., `https://my-project.firebaseio.com/.json`) would be blocked.

Lastly, though the most recommended thing to do is to secure the database with appropriate access rules, in order to make it more difficult for attackers to perform large scale automatic extraction of Firebase database details from Android apps, a developer could apply some basic obfuscations. An option is to dynamically construct the Firebase project ID and URL at runtime and use the Firebase APIs to instantiate the database and get reference instead of using the `google-config.json` configuration file. In this way, an attacker performing only lightweight static analysis would have more difficulty to extract these details. Moreover, since the Firebase library itself is obfuscated, parameters of calls to database instantiation functions will be harder to extract statically.

A. Tool Support

In the previous section, we have discussed that developers usually create the database structure by setting the access control policy open with the intent to secure it later. By doing this, the developers risk forgetting the security of the database as some of the developers we contacted acknowledged. Even if Google shows warnings in the Firebase console and sends out warning emails when databases are configured to be public, developers of even popular apps continue to leak their databases. The database console and Android Studio (the official IDE) are two separate environments. We argue that if a solution that warns developers about open database is not integrated into the IDE, the developers will likely forget about the open database when publishing their app. The multiple email warnings that are sent by Google could potentially be missed if the the developer has different Google accounts for Firebase and Google Play.

To this end, we have developed a prototype tool as a plugin to Android Studio to assist developers in avoiding the main security issues highlighted by our large scale analysis. Below, we describe the tool in more details.

Firebase Checker: We propose a prototype Android Studio plug-in that automatically checks the security misconfiguration of the Firebase databases being used in the app under development [1]. This tool uses the same static analysis approach presented in Section III-A and evaluated in Section IV except in this case the analysis is performed on the app source code rather than the bytecode. Since the analysis is performed on the app source code, we minimize the limitation caused by obfuscation.

When a developer starts the analysis, the tool first checks the configuration file (`google-services.json`) to extract Firebase project ID. It then performs static analysis on the program source code to detect further reference to Firebase databases and paths. Once the Firebase database project ID and the different possible paths are extracted, the tool performs read and write tests to the different possible paths. The tool then generates a report with the status of the different end

points such as "world-readable but not writable." The developer can then take action in securing the database based on this report. The prototype is configured to start the analysis on a menu item click event. However, one can simply integrate the analysis tool in the build process so that automatic check can be performed when building the app. The prototype supports apps developed both in Java or Kotlin programming languages.

VII. Related Work

Cloud Database Security Security in the context of cloud database as a service is well studied from the point of view of secure data outsourcing [2], [3], [22], [25], [28], [37], [43]. The approaches propose privacy-preserving queries where data at rest and in transit is always encrypted and operations or queries are applied in the encrypted domain. If an attacker gains access to the database or interaction with the user, he/she would not be able to extract clear data. Further research is done to improve the efficiency of encrypted data queries by proposing a secure database indexing. While the research on secure data outsourcing focused on relational databases such as MySQL, Firebase cloud databases are non-relational (aka NoSQL) [27]. Since security was not the main feature of non-relational databases [44], they trade consistency and security for performance and scalability [34]. Therefore, previously proposed privacy-preserving security mechanisms (e.g., encrypted data) would degrade the performance [31] and hence would go against the design goal. For this reason, a data breach on Firebase cloud database has a high risk of compromising the security and privacy of end-users.

Recent nonscientific studies [8], [42] reported public Firebase cloud databases. While these studies analyze a snapshot of Google Play apps at a certain time, our large scale analysis is performed on top apps from three different time periods providing us the trend in misconfiguration. Moreover, while these studies just check whether a database is public, our study also reveals world-writable databases.

The most closely related work is by Continella et al. [9] where the authors perform a similar study on Amazon S3. In this study, the authors attempt to extract bucket (database) names from Alexa top 1 million websites. Moreover, the authors rely on two additional methods to get bucket names, (i) by generating candidate names using acronyms and dictionary words, and (ii) by using reverse DNS lookup queries to S3 IPs. In our study, we extract Firebase database information by performing static code analysis on top Android apps from three different years. Amazon S3 can also be used on mobile apps in place of Firebase databases. Therefore, this work and our work are complementary.

Static Analysis of Android Apps Static analysis examines program source code (or binary) without executing the program. Several previous works used static analysis to uncover potential security issues in Android apps [11]–[14], [29], [30], [32], [41]. These approaches mostly relied either on call-graph analysis or data flow analysis in order to see if sensitive data can be propagated from sensitive source (e.g., APIs that require permission) to sensitive sinks (e.g., network). WARDroid [33] uses static analysis to extract web APIs used in apps to perform inconsistency and vulnerability analysis on the endpoints. Similarly, we rely on static analysis to extract Firebase project ID/endpoints and documents/collection/paths to perform access control policy enforcement checks.

Access Control Policy Misconfiguration There is a wealth of research in access control policy verification [18], [23], [24], [26]. In ZELKOVA [5], the authors represent policy semantics as SMT and use a solver to verify properties in order to identify possible policy misconfiguration for AWS cloud database. Bauer et al. [6] use association rule mining to extract rules from historical accesses. They then use these mined rules to analyze future policies for potential misconfiguration. In our case, instead of looking at the policy to look for property violations, our Android Studio plug-in performs tests on all possible database paths extracted from the app under development with combinations of read and write operations simulating an anonymous user.

VIII. Conclusions

Mobile and web apps are migrating their database to the cloud. Google's Firebase cloud database service is popular among app developers. The fact that it comes integrated with Android Studio makes it the obvious choice for developers. With the shared responsibility model, cloud database users are responsible for securing their database with the appropriate access control policy. However, Firebase database access control misconfigurations are becoming more common and are attracting malicious users.

We investigated the trend of Firebase database access control misconfiguration among 50K+ top (presumably high quality) Android apps across several years. While our study shows that the number of apps with a misconfigured database is decreasing year by year, there are still several top apps with misconfigured databases exposing sensitive user data. Since these apps are very popular, more than 760 million users are affected. If top apps pose these kinds of security risks, it is not hard to imagine how apps developed by inexperienced developers would be handling their databases. These results show us that there is a need for more work to improve the security of cloud databases when the shared responsibility model depends on the inexperienced developer to write the appropriate access control policy.

The fact that the same database can be shared among different platforms amplifies the security issues. To assist developers to improve the security of their Firebase database, we developed and released an open-source static analysis tool as an Android Studio plug-in that checks the accessibility of the Firebase databases used in an app under development.

References

[1] Firebase-Checker, https://github.com/biniamf/firebase-checker.

[2] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. Two can keep a secret: A distributed architecture for secure database services. *CIDR 2005*, 2005.

[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574, 2004.

[4] Amazon. Amazon web services: Overview of security processes, https://d0.awsstatic.com/whitepapers/aws-security-whitepaper.pdf, last accessed October 12^{th} 2020.

[5] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming. Semantic-based automated reasoning for aws access policies using smt. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9, 2018.

[6] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. *ACM Trans. Inf. Syst. Secur.*, 14(1), June 2011.

[7] European Commission. What is personal data?, https://ec.europa.eu/info/law/law-topic/data-protection/reform/what-personal-data_en, last accessed October 12^{th} 2020.

[8] comparitech. Report: Estimated 24,000 android apps expose user data through firebase blunders, last accessed July 2020.

[9] Andrea Continella, Mario Polino, Marcello Pogliani, and Stefano Zanero. There's a hole in that bucket! a large-scale analysis of misconfigured s3 buckets. ACSAC '18, page 702–711, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Access control: principles and solutions. *Software: Practice and Experience*, 33(5):397–421, 2003.

[11] Biniam Fisseha Demissie and Mariano Ceccato. Security testing of second order permission re-delegation vulnerabilities in android apps. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pages 1–11, 2020.

[12] Biniam Fisseha Demissie, Mariano Ceccato, and Lwin Khin Shar. Anflo: Detecting anomalous sensitive information flows in android apps. In *Proceedings of the 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. ACM, 2018.

[13] Biniam Fisseha Demissie, Mariano Ceccato, and Lwin Khin Shar. Security analysis of permission re-delegation vulnerabilities in android apps. *Empirical Software Engineering*, 25(6):5084–5136, 2020.

[14] Biniam Fisseha Demissie, Davide Ghio, Mariano Ceccato, and Andrea Avancini. Identifying android inter app communication vulnerabilities using static and dynamic analysis. In *Proceedings of the IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pages 255–266. ACM, 2016.

[15] Firebase. FirebaseDatabase, https://firebase.google.com/docs/reference/android/com/google/firebase/database/FirebaseDatabase, last accessed April 20^{th} 2021.

[16] Firebase. Build a note-taking app with flutter + firebase, https://medium.com/flutter-community/build-a-note-taking-app-with-flutter-firebase-part-i-53816e7a3788, last accessed February 2020.

[17] Firebase. Five tips to secure your app, https://youtu.be/pvLkkLjHdkw?t=1331, last accessed February 2020.

[18] Kathi Fisler, Shriram Krishnamurthi, Leo A Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering*, pages 196–205, 2005.

[19] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[20] Gartner. Gartner says the future of the database market is the cloud, https://www.gartner.com/en/newsroom/press-releases/2019-07-01-gartner-says-the-future-of-the-database-market-is-the, last access in February 2020.

[21] Hackerone. Periscope-all firebase database takeover , https://hackerone.com/reports/684099, last accessed February 2020.

[22] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 720–731, 2004.

[23] Graham Hughes and Tevfik Bultan. Automated verification of access control policies. 2004.

[24] Graham Hughes and Tevfik Bultan. Automated verification of access control policies using a sat solver. *International journal on software tools for technology transfer*, 10(6):503–520, 2008.

[25] Murat Kantarcıoglu and Chris Clifton. Security issues in querying encrypted data. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 325–337. Springer, 2005.

[26] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing web access control policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 677–686, 2007.

[27] N. Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[28] Jun Li and Edward R Omiecinski. Efficiency and security trade-off in supporting range queries on encrypted databases. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 69–83. Springer, 2005.

[29] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, pages 280–291, 2015.

[30] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[31] R. Macedo, J. Paulo, R. Pontes, B. Portela, T. Oliveira, M. Matos, and R. Oliveira. A practical framework for privacy-preserving nosql databases. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 11–20, 2017.

[32] Christopher Mann and Artem Starostin. A framework for static detection of privacy leaks in Android applications. In *27th Symposium on Applied Computing (SAC): Computer Security Track*, pages 1457–1462, 2012.

[33] A. Mendoza and G. Gu. Mobile application web api reconnaissance: Web-to-mobile inconsistencies vulnerabilities. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 756–769, 2018.

[34] Lior Okman, Nurit Gal-Oz, Yaron Gonen, Ehud Gudes, and Jenny Abramov. Security issues in nosql databases. In *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 541–547. IEEE, 2011.

[35] OpenID. Open*id*, https://openid.net/, last accessed October 12^{th} 2020.

[36] OWASP. Owasp api security project, last accessed August 2020.

[37] Erez Shmueli, Ronen Waisenberg, Yuval Elovici, and Ehud Gudes. Designing secure indexes for encrypted databases. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 54–68. Springer, 2005.

[38] Statcounter. Mobile operating system market share worldwide , https://gs.statcounter.com/os-market-share/mobile/worldwidef, last accessed November 20^{th} 2020.

[39] Statista. Google play: number of available apps 2009-2020, https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, last accessed October 2020.

[40] Techcrunch. Donald daters, a dating app for trump supporters, leaked its users' data, https://techcrunch.com/2018/10/15/donald-daters-a-dating-app-for-trump-supporters-leaked-its-users-data/, last accessed 2020.

[41] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. AmAndroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1329–1341, New York, NY, USA, 2014. ACM.

[42] Xda-developers. Millions of users' data leaked through misconfigured firebase backends, https://www.xda-developers.com/user-data-leak-misconfigured-firebase-backends/, last accessed February 2020.

[43] Zhiqiang Yang, Sheng Zhong, and Rebecca N Wright. Privacy-preserving queries on encrypted data. In *European Symposium on Research in Computer Security*, pages 479–495. Springer, 2006.

[44] A. Zahid, R. Masood, and M. A. Shibli. Security of sharded nosql databases: A comparative analysis. In *2014 Conference on Information Assurance and Cyber Security (CIACS)*, pages 1–8, 2014.