A Modern-Day Side-Channel Attack: Utilises Several Vulnerable Channels on Android to Detect your Identity

Amin Mansour

Abstract

Side-channels attacks have always existed. Starting with Android Marshmallow (Android 6), access to these side channels have been constrained to protect against possible information leaks between apps. In this paper, we propose a simple technique that can infer certain activities from the way an app uses external storage and use it to build an attack. Such occurrences arise from vulnerabilities in the app's logic. It also comes from Android's inability to isolate and safeguard adequately. In this paper, we utilise this vulnerability to attack the Instagram app and find the user's identity. We do so by associating local activity (detectable through changes in the external storage amongst other things) with events that occur remotely on Instagram. These connections can be formed from a mixture of strategies, each achieving varying levels of success. This paper details a complete analysis of the different strategies considered. We put forward a fully-operational and scalable approach that successfully identifies the user's profile.

Project source : https://gitlab.doc.ic.ac.uk/am318/side-channel-attack/

1. Introduction

In this paper, we introduce a single attack which takes advantage of several side-channels available on Android. Android has become the most active platform, and the Play Store commands the largest repository of any market. The Android operating system welcomes variation amongst apps while trying to protect itself from vulnerabilities. This has proved to be very challenging in practice with the number of attacks increasing each year [1]. Our focus in this project is towards the external storage and in particular how apps utilise it to carry out functionality. Android provides apps with individual private folders that are isolated from all possible entry points. The containment prevents any third-party apps (or users) having access to sensitive information belonging to individual apps. However, sometimes it is necessary to store data publicly through the external storage, i.e. data may need to persist independently of the app. For this, there is no well-defined standard, and as a result, many apps fail to handle their data securely. Many apps use external storage to store non-sensitive data that when combined with various techniques can be used to obtain

private information. A malicious app can exploit these leaks to detect information relating to the user without requiring any critical permissions. As we will see later, our attack builds on this idea. We will show how the Instagram app is susceptible to the same problem and how we can use it to derive the identity of the user.

Apps typically generate a significant amount of data. How securely these apps store information depends on the app's design and how well the Android OS acts to secure vulnerable channels that might leak information across. Many of the attacks focus on exploiting vulnerabilities in implementation. Android rarely depreciates code. In development, there are usually many ways to implement a feature, and the developer may not be aware of the subtle differences between each. This confusion can cause vulnerabilities in the code that may become apparent later. Our attack principally targets apps that leak data to public storage. We use instances of this to infer the activity that is occurring on the foreground and utilise it to obtain the identity of the person. Instagram will be the app that will be used to demonstrate this attack. A web crawler will crawl known endpoints on the Instagram platform. We also provide flexible strategies in protecting against this attack. One of them includes partitioning the external storage into isolated regions and enforcing apps that use external storage to specify access to those partitions. It would implement coarse-grained access control but might be problematic for apps which utilise other directories in their functionality.

In this paper, we will start by giving a brief overview of the main areas of the Android operating system. It will provide enough overview to appreciate this attack fully. We will then go in detail on how to implement this side-channel attack. After that, we extend the attack by showing how we can utilise it to identify a user on Instagram. This will include a comprehensive study of the different techniques and show how the attack was constructed. For our evaluation section, we test the attack on a target device and examine how successful it is at identifying the identity of a user.

2. Background

In this section we give a brief overview of the Android platform. It will provide enough of an overview to fully understand the approach taken in this paper.

2.1. Android Ecosystem

The Android adoption rate has increased with it now commanding an 85% share in the mobile market [8]. An amalgamation of its open source licensing, its expansive and open app market, and its widely customizable interface make it extremely popular amongst consumers and hardware vendors.

Android is an open-source mobile operating system built on top of other known open source systems - most notably the Linux OS. Though Android is built on Linux and relies heavily

on much of its infrastructure - most notably the kernel - Android has become a fully-edged operating system that diverges from Linux. Android presents a vast array of frameworks that are constructed on top of the kernel and operate on the user space level. They collectively work to limit and constrain specific Linux capabilities, whilst enhancing more obscure Linux features. Most of the later versions of Android primarily focus on modifications to these frameworks/libraries, with only a few affecting the underlying system kernal.

Android's extensive set of frameworks is what allows Android to break free from it being just another embedded Linux distribution system like MontaVista - the primary mobile system that preceded Android. Frameworks facilitate and simplify the creation procedure of applications, permitting developers to use the higher-level Java language, instead of C++ or C. Java along with the Android run time environment (ART) (Dalvik in versions prior to Marshmallow) allows apps to be ported across devices with relative ease. This process involves taking the Java code associated with an app and converting it into DEX bytecode. The DEX bytecode is independent of any underlying device. The DEX bytecode is then compiled to native machine code and packaged, ready to be executed on the device. ART introduces an ahead-of-time compiler which carries out this translation once at installation time. All subsequent runs of an app requires the retrieval of the compiled code from memory. For later versions of Android, ART comes with a just-in-time compiler which allows for optimisations to be made during the app's lifecycle on a device.

2.2. Application Sandbox

Frameworks also allows developers to utilise APIs, which grant both resource and hardware access to third-party apps. Android ecosystem uses Linux user-based security capabilities to isolate app resources and protect both the apps and the underlying system from malicious apps. It does this, by assigning an exclusive user ID (UID) to each process. This UID is used by Android to enforce constraints and provide isolation. By default, only the (third-party) app that initiates a process can access the resources related to that process. Libraries, frameworks, runtime environment, and all applications, run within this Application Sandbox.

Further enhancements have been introduced over time with later versions of Android. They have significantly strengthened the original UID-based discretionary access control sandbox introduced in legacy versions of Android. In Android 9 all third-party apps with targetSdkVersion >= 28 must run in their own SELinux sandboxes, providing mandatory access control on a per-app basis.

2.3. Permission Model

As was discussed in the previous section, Android apps are self-contained and by default only have access to their own files and a minimal set of low-risk system services. For a third-party app to obtain additional capabilities, it can request for additional permissions at run-time. Pre-historically these types of permissions were granted at installation time. This approach was later replaced on Marshmallow for a more dynamic strategy, as to prevent malicious apps taking advantage from the lack of information available to the user at installation time. The implications of the install-time approach have been covered in detail in previous works [9].

The Android SDK comes with a set of predefined permissions that define access to underlying system resources. These permissions are broad and group many similar capabilities under one assignment. The user is not expected to accept all permissions individually. This process would be too intrusive. For an app requesting a permission instance, the user is prompted with the permission at run-time along with the other permissions that share the same group and are required by the app. If the permission instance is granted, then the Android system allows the desired functionality to take place. When a permission instance from the group is later requested it is automatically granted. With later versions of Android, newer permissions are added to the predefined set of permissions. They can either arise with the emergence of newer technologies (i.e. USE_BIOMETRIC) or protect pre-existing vulnerable channels that were once believed to be safe (i.e. READ_CLIPBOARD_IN_BACKGROUND). Also, more general existing permissions can be broken down into smaller permissions. READ_EXTERNAL_STORAGE on Android Q (9) is broken down into READ_MEDIA_VIDEO, READ_MEDIA_IMAGES and READ_MEDIA_AUDIO.

Applications must specify the permissions they require in the AndroidManifest.xml located within the source files. Permissions are classified into protection levels. Permission protection levels characterise the potential risk implied in the permission and how the Android system should deal with them. "Normal" protection level is the default protection level. Permissions classified as such are automatically granted at installation time without the need of prompting the user. They are generally considered to be low risk. ACCESS_NETWORK_STATE is an example of a normal-level permission. The "Dangerous" protection level, in contrast, labels permissions of high risk. The user at run-time must explicitly grant this type of permission.

In this project, our malicious SDA app requires the READ_EXTERNAL_STORAGE permission. Before Android KitKat (4), this permission was defined as a normal-level permission but in later versions was reclassified as dangerous. Today it requires of the user at run-time to grant it. In Android version Q (9), the permission is broken up into smaller permissions. The attack proposed is still possible in this context. We must request for the READ_MEDIA_IMAGES permission instead.

The opposite is true with the INTERNET permission, which pre-historically allowed apps to access the internet. It was defined in previous versions of Android as a dangerous permission requiring approval at installation time. It was deprecated as a permission in later versions. Third-party apps now have automatic access to the internet. The reasons given by Google for this decision was that the user's data is protected on the Android system by default, and as a result, no additional risk is added if third-party apps can access the internet [10]. The problem with this assumption is that in practice there is no way of validating the security of the Android system. As we will later see, we describe a way of obtaining sensitive information from side-channels on the Android operating system. Another more obvious reason for Google rescinding the internet permission is because it encourages apps to be more advertising-friendly. Many apps do not require internet but construct their business around showing ads. If the internet permission was reverted to being dangerous, then most users would decline to grant it, specifically in cases where the reasons for requiring it were not apparent to the user.

2.4. External Storage

Apps can additionally make use of external storage and store files on the device. The data generated by the apps are independent from the apps themselves and persist after uninstallation. There is no security imposed on files that are saved to external storage. This makes it a frequent target of attacks. All apps have the same view of the external storage. For an app to access external storage it must first acquire permissions that allow reading and writing to the external storage. In Android Oreo (8), the permissions are READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE respectively.

In the Android system, there are two types of external storage recognised. The first is primary external storage: storage that comes with the device and can be accessed by apps at any time. The other is secondary external storage, which is external to the device and can be removed (i.e. SD cards). Access is subject to whether the external storage is available at the time of access. In this project, we will focus on primary external and more specifically the DCIM directory located in the external storage.

2.5. Side-Channel Attack

A side-channel attack on Android usually requires trying to obtain information through exploiting the underlying system. Instead of targeting the implementation of an app like more conventional attacks. Android OS is a fully-operational system which provides support for multiple apps on a single device. This comes with many challenges. Most of Android's Kernel is Linux-comprised, which means in a mobile context a lot of it would be considered dangerous or unnecessary. Android through updates has slowly patched out vulnerabilities that arise from these insecure channels. As was discussed previously, it was done by integrating libraries/frameworks on top of the kernel to limit access to specific resources. In Android Marshmallow (6), third-party apps were blocked from accessing real-time power,

memory and network usage information. These were primary targets for adversaries prehistorically [11]. Some shared channels remain open today and are necessary to allow apps to function. The external storage being one of them. The operating system does not impose any limitations on the external storage and apps are free to access it. However, this open approach comes at a risk. We will leverage this to carry out our attack.

We propose a side-channel attack in this paper which leverage's on several side-channels found on the Android operating system, one of those being the external storage, and use each in combination to predict whether or not the Instagram app is running on the foreground and also predict when the user makes a post from their Instagram client. This paper was heavily inspired by a previous work that proposed and implemented a side-channel attack that works a similar way [12]. It involved time-stamping and geo-tagging a tweet sent on a target device and then using the Twitter API to locate the exact tweet on Twitter by using its location and time. It would be repeated several times until the user could be identified. The malicious app was able to detect when the user tweeted from their device by accessing the /proc/uid_stat/ <uid>/tcp_rcv public directory. The malicious app would detect when Twitter was running in the foreground and listen for a packet with a certain signature matching that of a tweet packet. In Android Marshmallow third-party apps were blocked from having access to the /proc/uid_stat directory. As a result, a lot of legitimate apps which require network usage statistics went out of business. This attack was proposed before Android Marshmellow and no longer works. We try to propose a similar attack which is structured the same way but works on all versions of Android.

The general structure of that attack is very similar to the attack we propose in this paper. The attack can be broken into 3 main phases:

1. **Phase ONE** : Identify when a post is made on the Instagram client (by using several side-channels).
2. **Phase TWO** : Detect how to link the local post event with an event that occurs on the Instagram platform (utilising web crawlers).
3. **Phase THREE** : Obtain the identity of the user along with their posts and store it on a remote server.

We will refer to our malicious app that will exploit these side-channels with the acronym SCA. It will stand for SIDE-CHANNEL ATTACK. The app will run in the background as a service.

3. Phase One - Detecting an Instagram Post

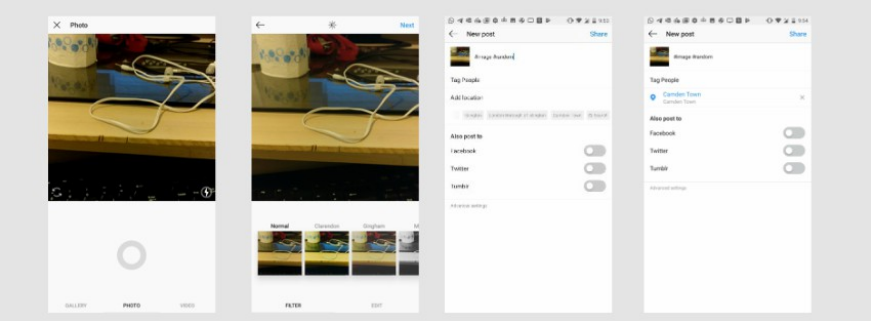


Figure 1: The stages involved in making a post on the Instagram app

Making a post on the Instagram client is relatively straight-forward. It involves a user following a sequence of steps. From fig 1, The user first chooses the media instance to post. This may either be a video or an image and can either come from the gallery or the camera. After selecting the media item, Instagram will open its editor and enable post-modifications to be made to the media object. The next section involves defining additional tags to accompany the media item. That includes the description that makes the post on Instagram easier to find. The user can also apply an extra location tag to the post as well. Unlike Twitter, this is left up to the user to define themselves. Then after the user is satisfied with their post they can finalise the process by pressing the share button on the top right.

During this process of creating a post on the Instagram app, various data leaks occur. This will allow our SCA app to not only detect when Instagram is running on the foreground but also detect when the user is making a post and what step in this process the user is on during this procedure. The SCA app does this by combining the leaks from several side-channels and making certain inferences from each.

3.1. The Channels Exploited

"Instagram is a free photo and video sharing app available on Apple iOS, Android and Windows Phone" [13]. We will only focus on posts that are made through the camera and not on those selected through the image gallery. This attack can also be generalized for videos but we will only focus on photos to demonstrate the fundamentals of this attack.

Our SCA app is running in the background and listening for certain activities. For our SCA app, there are two states (each represented as fields in the application) :

PASSIVE_STATE (initially equal to 1 for true) : This represents the state when the SCA app is waiting for an Instagram post procedure to begin. This is the default state of the SCA app.

ACTIVE_STATE (initially equal to 0 for false) : This represents the state when the user is believed to be currently posting on Instagram. Several side channels can be used in combination to infer this event.

3.1.1. Overview

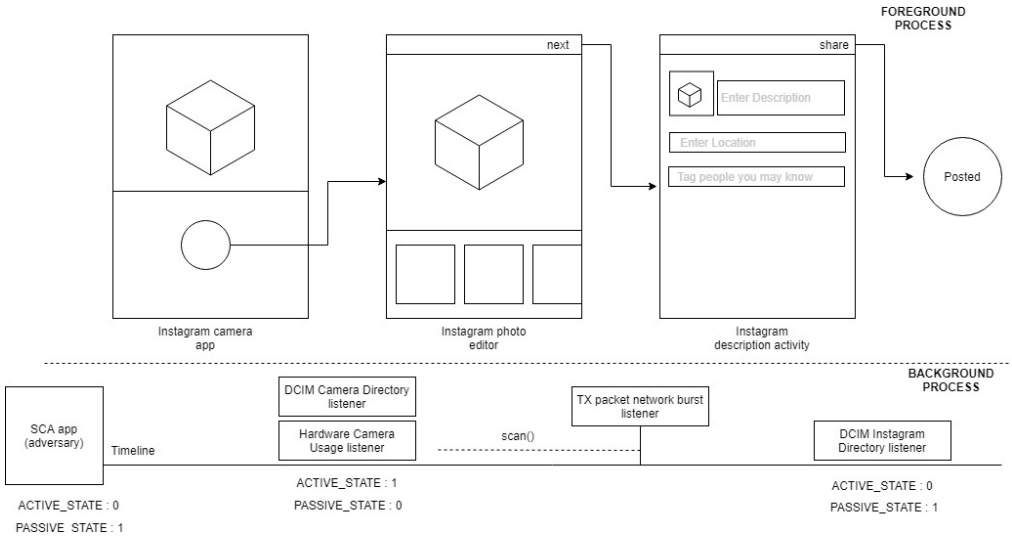


Figure 2: The Instagram post procedure represented as a transition diagram alongside the SCA activity timeline. The listeners above the timeline represent what is triggered at certain parts of the process. What is below the timeline represents the state changes that occur after each function is executed.

In the next section we will define the job of each cross-channel listener featured in our SCA app.

3.1.2. Camera Directory Listener

So as was mentioned above, for a user to make an Instagram post, they must utilise the device's camera. The Instagram app implements its camera feature instead of utilising the default camera app that can be accessed programmatically via intents. It uses Android's camera2 package which "provides an interface to individual camera devices connected to an Android device" [14]. On taking an image with the Instagram camera, the app automatically stores a copy of the image within the DCIM camera directory located in public storage, irrespective of if the post is later made or not. From the original Android documentation

of camera2 on how to a "display a camera preview" and "take pictures", the code necessary to build a camera is compiled into a sample app [14]. When this sample app is ran, and a photo is taken the same thing occurs. The image is automatically stored within the DCIM camera directory. It is not obvious locating the part in the code to disable this feature. The developers designing this app might not have been aware of the leakage that occurs when a user takes a image.

Our SCA app takes advantage of this leak by listening for changes to the DCIM camera directory. It does this by setting a FileObserver on that directory as seen below in fig 3. The state of the SCA app changes from a PASSIVE_STATE to an ACTIVE_STATE when a new file is added.

From this point, the SCA app does not know with certainty if Instagram's posting process (as defined in fig 1) has begun. It still requires other side-channels to be employed to achieve greater certainty. The file added to the camera directory may just be a photo taken with the native camera app. In this case the SCA app must still detect it but must recognise it as being unrelated to the Instagram app.

```
final File cameraDirectory = Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DCIM);
final File dir = new File(cameraDirectory.getAbsolutePath() + "/Camera");
dcimObserver = new FileObserver(dir.getPath()) {


    @Override
    public void onEvent(int event, String file) {
        if (PASSIVE_STATE == 1) {
            if (event == FileObserver.CREATE && !file.equals(".probe")) {
                imagePreModified = new File(dir.getPath() + "/" + file);
                PASSIVE_STATE = 0;
                ACTIVE_STATE = 1;
            }
        }
    }
};
dcimObserver.startWatching();
```

Figure 3: Android code that implements a directory listener.

Additionally, the image leaked to the DCIM camera directory contains fine-grained geotagging metadata. It is even present when the device has the geotag feature disabled on their native camera app. This is another example of a side-channel attack. The malicious app (SCA) can obtain fine-grained GPS data without requiring the permissions needed to get it directly. We will later use this to generate possible location tags for which we can search posts for.

3.1.3. Camera Activity Listener

Our malicious app (SCA) can detect when the camera of the device is in use and we can do it without requiring the camera permission. We can do this by setting up two individual callback functions that will be fired when the camera becomes available and unavailable. We can obtain the CameraManager by getting the system's camera service, and register an AvailabilityCallback object by calling the registerAvailabilityCallback() method on that CameraManager instance. The AvailabilityCallback object defines two methods :

onCameraUnavailable(String cameraId) : This is triggered when the lock for the camera(s) is obtained by a new process. More informally, it is when the camera goes from being available to unavailable. For fig 1, the camera must be opened to make a post. When this occurs, the camera lock is obtained by the Instagram app. We are more interested in the case where the camera goes from being unavailable to available in this attack.

onCameraAvailable(String cameraId) : This is triggered when the device lock for the camera(s) is released by a process. More informally, it is when the camera goes from being unavailable to available. This is triggered anytime the user takes an image with their camera. This is used in conjunction with the storage listener outlined above to detect the first stage of a post procedure on Instagram (fig 2). The DCIM camera directory listener will always be fired before this CameraManager callback function. This callback is additionally called every time the SCA app starts (since the camera is available at this point) so we must check that the SCA app is in an ACTIVE_STATE before proceeding with the required next step of the attack to account for this occurrence. We do not want anything to happen if the app is in a PASSIVE_STATE and has not detected a new file in the DCIM directory.

It might not be apparent at this stage why we need the onCameraAvailable callback function when it is enough to have the DCIM camera directory listener to detect when a photo is taken. Google provides a feature which allows pictures of the user to be moved from their DCIM camera directory to their Google Photo storage and back. We wanted a way to infer with certainty that the files being added to the DCIM camera directory were ones that were recently produced with the device's camera. We needed both to be able to infer that.

3.1.4. Network Packet Burst Detector

We have so far covered how our SCA app detects when the post procedure begins. On detection, this triggers the SCA scan process to start. This process can only begin when the app is in an ACTIVE_STATE and will stop when the app goes back to a PASSIVE_STATE (i.e. when the camera is reopened, or a timeout of inactivity occurs). This process involves recording the amount of data being transmitted over the network per unit of time and inferring when the image (of the post) is sent. As was mentioned above, apps were given access to fine-grain network usage stats before Android Marshmallow (6). In later versions this was blocked, and as an alternative Android introduced the TrafficStats class which provided general network information. We utilise the static getTotalTxBytes() method which provides us with the total bytes transferred from the device since the mobile was last booted (resets to 0 after every reboot). Unlike the prior approach which allowed us to identify transmitted packets from individual apps, we only have access to the total amount of bytes transmitted by the device. On a scan, we continually invoke this method within a loop and subtract each value with the previous to obtain the total byte change per second. This byte change represents a packet that is transmitted over the network. If this packet follows a particular size signature which is similar to that of the image, then we can extrapolate that the packet must be related to the Instagram post procedure. When this occurs, the SCA app can infer that the user is at the definition step (the 3rd stage in fig 1) of the post procedure.

We can not be sure that the packet detected is from the Instagram app yet. In cases where there is a background process that is transmitting large amounts of data across the network, this approach does not work well, but as we will see, it will have no impact on the reliability of this attack.

We know from analysing the network activity that the image itself is sent between the post-image modification step and the description step in the process. When the user is happy with the modifications to the image and presses next, it is processed and sent over the network. We analysed the packet signatures that are transmitted during a post procedure. To conduct this experiment, we cleaned our test device from any background processes (apart from the SCA app). We repeated the experiment 15 times using the same sized photo each time.

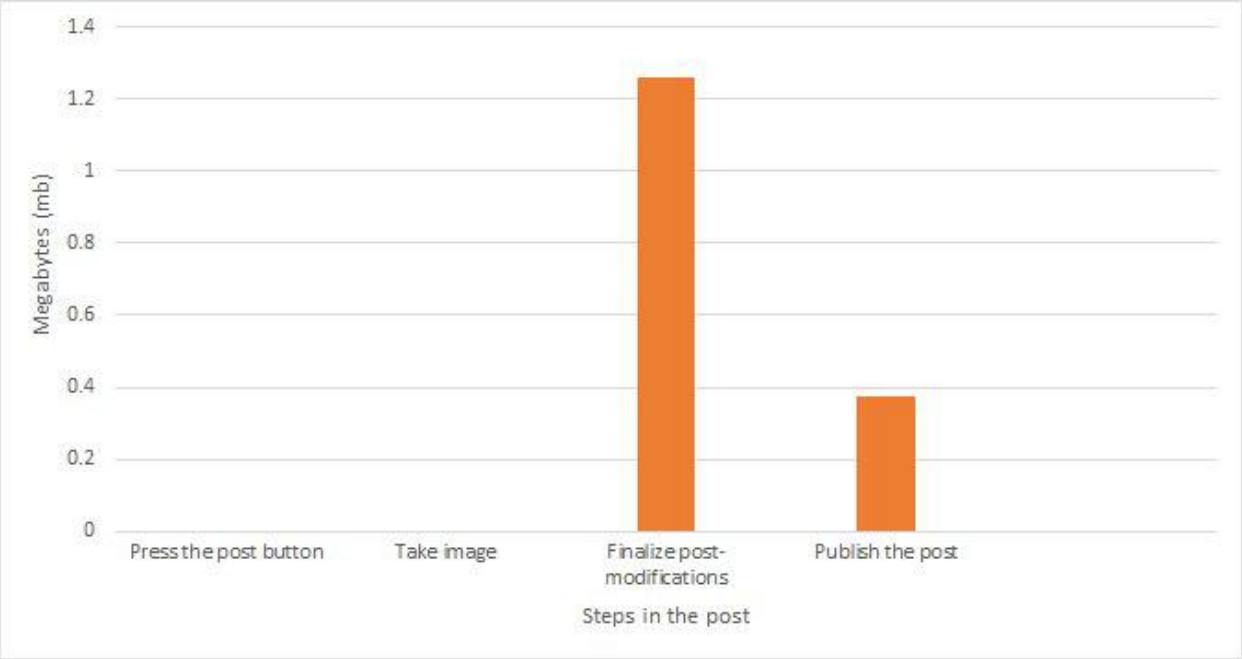


Figure 4: A bar chart which outlines the megabytes of data transmitted over the network at each stage of the post procedure.

We can see a TX burst occurring twice - first when the user finalises the modification of an image and the second when the user completes their post. From a closer look, the second burst is large enough (0.28 mb) to contain the metadata of the post but not large enough to hold the image itself (1.03 mb). So it was conjectured that the first burst must represent the event of the image being sent over the network.

3.1.5. DCIM Instagram directory listener

At this point, our SCA app can detect both when a post procedure starts and when the description part of that process is reached. As was discussed previously, from the description page the user can apply several tags on a post. The user then completes the process by clicking the share button which publishes it on the platform. When this occurs, the post's image is added to the DCIM Instagram directory automatically. Our SCA app detects this by listening for new files within the DCIM Instagram directory. It sets a FileObserver to point to the DCIM Instagram directory. The DCIM is a public directory and can be accessed by any app with the EXTERNAL_STORAGE_READ permission.

This side-channel leak indicates to the SCA that the Instagram app is running on the foreground and validates all side-channel inferences made prior. It can be possible to carry out this single side-channel exploit alone and still be able to detect when a post is made on Instagram. The problem with this approach would be that all files added to the DCIM Instagram directory are not necessarily from regular posts made on the platform. Instagram

offers a story feature which allows users to share their stories amongst their followers for 24 hours. With this feature, the user has an option to save the images they post to their public storage. So the app must distinguish between the two events. As we will later see, we must be able to link a local event with a remote one, and stories are private and not accessible through crawlers. Our SCA app can identify with certainty when the user is posting a regular post.

4. Phase Two - Linking Local Activity with a Remote Event on Instagram

Next we discuss how we can best utilize what we have implemented in our SCA app to identify the user.

4.1. Instagram Crawler

For our next part we will need a way of retrieving posts from Instagram. The Instagram API, which traditionally allowed developers to search for posts, was restricted from the beginning of 2018. It came after the Cambridge Analytica Scandel broke [15]. The changes that came about as a result meant that access to the offical API was severely constrained. The Instagram Graph API is now only available for businesses that provide formal proof of their credentials.

We instead used an unofficial open source crawler for retrieving real-time Instagram data, which works by exploiting known endpoints on Instagram. We went with a service provider Phantombuster which hosts the crawler on their server and allows us to make crawl-requests programmatically. We can define which hashtags or locations we are interested in, and it would collect the data straight off Instagram and place it straight into the JSON file ready for processing. The JSON key fields for each post crawled is represented below:

profileUrl (Profile URL of post author)

profileName (Username of post author (only if available))

ownerId (Instagram unique ID of post author)

postUrl (Instagram post URL)

description (Post description)

pubDate (Post publication date)

likeCount (Number of likes the post received)

commentCount (Number of comments the post received)

views (Number of views, if the post contains a video)

location (Where the photo was taken (only if available))

query (Hashtag (or location) that lead to the post)

postVideo (Link to raw video file, if available, can contains more than one link if the data is available)

videoThumbnail (Link to raw video thumbnail, can contains more than one link if the data is available)

postImage (Link to raw post image, can contains more than one link if the data is available)

Figure 5: The JSON keys for an Instagram post.

4.2. Data Study of The Various Approaches

It has been shown that the Instagram app leaks a lot of information. At this stage, we have implemented the means of detecting when a post is made. We also have access to the image that is associated with the post stored within the DCIM Instagram public directory. We now require to utilise both to identify the user.

There are several approaches we can take to link a local event (the user making a post on their device) with an event that occurs on Instagram. The process involves detecting when a post is made on a device and then finding the equivalent post on Instagram. We take a perceptual hash of the image contained within each post retrieved and compare it with the perceptual hash of the image stored within the DCIM Instagram directory. If the two are equivalent, then we have identified the post for which the user posted. We can then determine the user by simply extracting their profielName and profileUrl out from the post. If there are no matches, then the post has not been identified, and the crawling attempt was unsuccessful. In this project, we must determine suitable queries that are optimal in finding a particular post.

The response the crawler returns for a specific query depends on how Instagram decides to serve the request. The crawler only works through known endpoints implemented by Instagram. So the crawler cannot retrieve recent posts with no hashtag or location specified. Instagram uses powerful techniques of obfuscation to prevent effective data crawling. It stripes the location field, the profileName field and other important fields from a post one hour after it is posted and attributes it with a postID. We need the profileUrl and profileName to link a post with a user. Since our attack will be triggered immediately after the user makes a post, it does not affect our attack. A more destructive finding is that Instagram only allows 22 posts to be returned where fields for location, profileName and profileUrl exist. Even when the crawler is specified to return 500 posts for a particular hashtag, it would only return 22 posts where location, ProfileName and ProfileUrl values were available to access. We can counter this by repeating the same query to retrieve 22 new posts. However, as we will see, there is no guarantee that Instagram will provide us with 22 different posts. We analysed the effect of this below.

We conducted an experiment which examined how easy it would be for our crawler to find posts made on Instagram. We posted ten individual posts using ten different hashtags of varying popularity. We additionally published four posts with four different locations. The locations used are of varying granularity. All the crawls were made within 2 minutes of making the original post.

Post tag (either hashtag or location)	After 1 crawl: Success rate	After 2 crawls: Success rate	After 5 crawls: Success rate	After 10 crawls: Success rate
#screen (1.6m posts)	100% (found)	100% (2/2 found)	100% (5/5 found)	100% (10/10 found)
#technology (10.2m posts)	100% (found)	50% (1/2 found)	80% (4/5 found)	90% (9/10 found)
#travel (398m posts)	0% (not found)	0%	0%	0%
#bike (15.9m posts)	100% (found)	100% (2/2 found)	100% (5/5 found)	100% (10/10 found)
#car (57.5m posts)	100% (found)	100% (2/2 found)	100% (5/5 found)	100% (10/10 found)
#burger (4.3m posts)	100% (found)	100% (2/2 found)	100% (5/5 found)	100% (10/10 found)
#monalisa (1.3m posts)	100% (found)	100% (2/2 found)	100% (5/5 found)	100% (10/10 found)
#keyboard (2.1m posts)	100% (found)	100% (2/2 found)	100% (5/5 found)	100% (10/10 found)
#guitar (31m posts)	100% (found)	100% (2/2 found)	100% (5/5 found)	100% (10/10 found)
#pig (4.3m posts)	100% (found)	100% (2/2 found)	100% (5/5 found)	100% (10/10 found)
London, United Kingdom	100% (found)	50% (1/2 found)	20% (1/5 found)	10% (1/10 found)
Islington	100% (found)	100% (2/2 found)	100% (5/5 found)	100% (10/10 found)
Camden Market	100% (found)	100% (2/2 found)	100% (5/5 found)	100% (10/10 found)
United Kingdom	0%	0%	0%	0%

Figure 6: An experiment which involved creating a post with a particular tag (either a hashtag or location) and trying to find the equivalent post on Instagram by crawling for that tag. Success rate is the percentage of times the post was correctly identified. The queries for a particular tag were made consecutively instead of in parallel.

There are two interesting observations. Whenever the crawler successfully identifies and returns a post for a particular tag on its first attempt, the crawler is successful for the next ten as well. This would indicate that the Instagram algorithm is predictable in how it responds to queries. The second interesting observation is where the crawler did not work. The crawler was not able to find the post with the #travel tag. #travel is a popular hashtag on Instagram. London, United Kingdom tag was found once and not found again after and United Kingdom was not found at all. It would appear that Instagram puts a high emphasis on its more recent posts. It happens that channels that have small amounts of activity within them compared to more active ones produce a higher chance for our crawler to find posts within them. Out of 140 total crawls 110 correctly identified the post. So it would appear that the attack would still be successful as long as the post had at least one uncommon hashtag or location attributed to it.

In this section, we carry out a data study on the three main approaches proposed and examine how successful each is at locating the user's post. Each approach categorises the type of request that needs to be made by the crawler in order to make that link.

LocationalBasedSearch : We can try to identify the user's post based on its location tag. This approach was inspired by the Twitter side-channel attack discussed earlier. Instagram, unlike Twitter, does not allow for the automatic geotagging of posts. The user in the description page can attribute a location of their choice. This choice may not be correct.

We analysed the utility of this approach by examining the number of posts made on Instagram that included their location. If the user does not include a location, then we can not use this approach to identify their post on Instagram. This study involved us crawling the posts of 100 random active profiles.

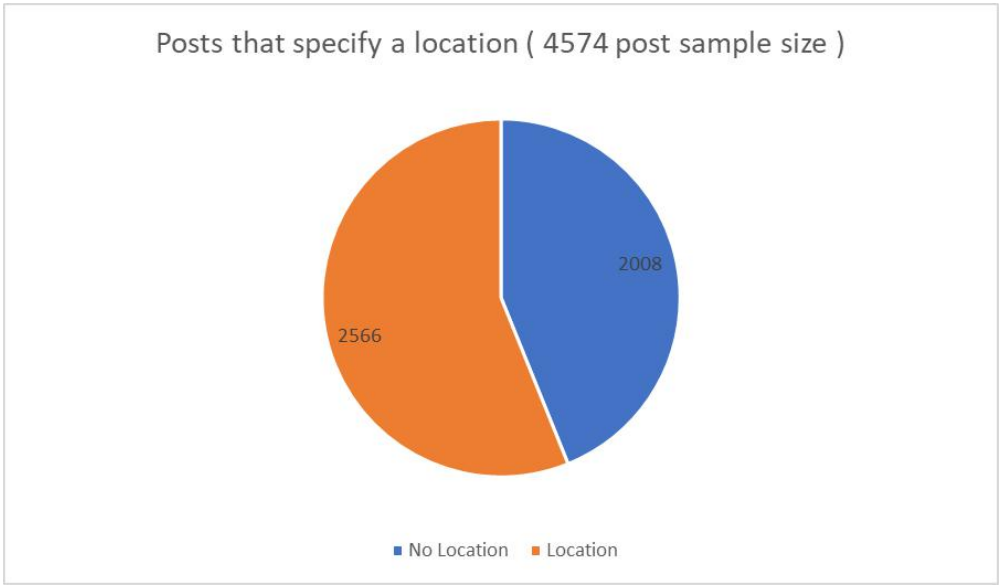


Figure 7: An experiment which involved crawling the profiles of 100 random active users and recording the number of posts which contained a location. There were a total of 4574 posts in this sample.

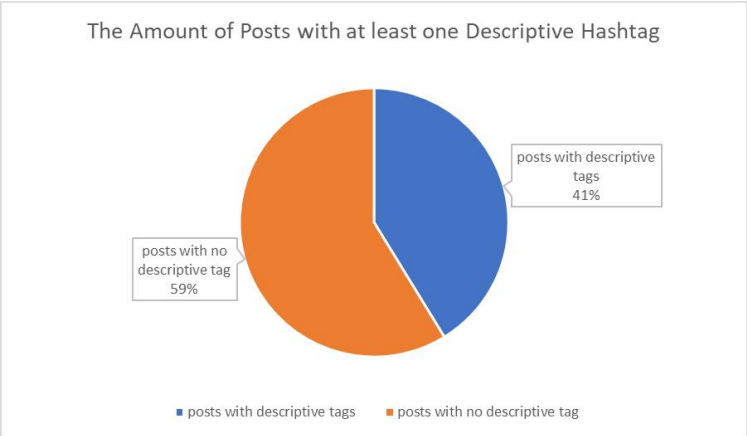
As can be demonstrated a total of 4574 posts have been examined. 56% of those include a location. This coupled with our previous study would indicate that utilising the location tag is an excellent approach for obtaining the identity of the user.

HashtagBasedSearch: This approach is based on predicting the possible hashtags associated with a post by analysing its image. We must assume that the user has left a description consisting of one or more tags and those descriptively represent the image. Instagram allows a user to attribute a serious of tags to a post (i.e. #car #black). These tags are represented as individual channels on the Instagram platform. We can crawl these individual channels and try to identify our post from amongst them.

We will use the Google Vision API which will allow us to annotate images. To evaluate the utility of this method, we must examine how well the computer vision algorithm works at tagging images on Instagram. This can be measured by how well the labels produced match up against the user-specified hashtags. In theory, our attack only requires one label to match for our crawler to successfully identify a post. The more labels that match, the higher the chance a post has of being found after several different queries.

We retrieved posts from 100 active users (spam accounts filtered). There were a total of 4391 (image) posts. We had to implement our own algorithm which could analyse data quickly. For each post retrieved, the image and description was extracted. We feed the image through the Google Vision API, producing a collection of automatically generated labels. We took the description of the post and extracted the hashtags from it. Then we calculated the difference between the two and recorded the results. We did the same for each post in the group. The data below represent the results of this experiment.

Figure 8: An experiment which involved crawling the profiles of 100 random active users and recording the similarity between the user-specified hashtags and the computer-generated labels of the image. The difference of each post in the group is recorded. There were a total of 4391 posts in this sample.



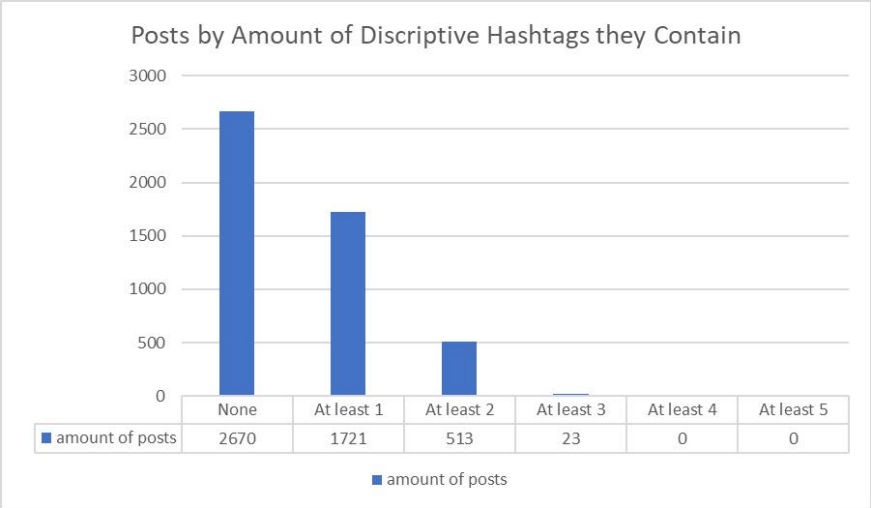


Figure 9: An experiment which involved crawling the profiles of 100 random active users and recording the similarity between the user-specified hashtags and the computer-generated labels of the image. The difference of each post in the group is recorded. There were a total of 4391 posts in this sample.

41% of the posts have a description with at least one descriptive label associated with it. It would indicate that this approach would be successful on average 4/10 times if executed.

PopularHashtagSearch : Instagram has comprised a list of their most common hashtags on their platform as seen below. This crude approach involves crawling posts based on these hashtags. A lot of the popular hashtags (i.e. #instagood) are general and can apply to a large number of post types. In theory, we only require a single matching term to be attributed to a user's post for our crawler to find it.

1	#love	1.221B
2	#instagood	704.0M
3	#photooftheday	478.6M
4	#fashion	456.5M
5	#beautiful	445.0M
6	#happy	413.8M
7	#cute	404.3M
8	#tbt	401.4M
9	#like4like	393.9M
10	#followme	374.3M

Figure 10: A list of the most common hashtags.

To examine the effectiveness of this approach we analysed the post of 100 random active users and analysed how many of them contained a popular hashtag.

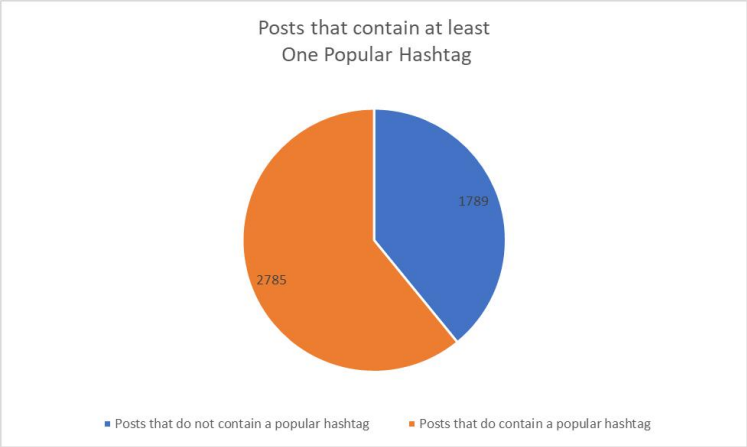


Figure 11: An experiment which involved crawling the profiles of 100 random active users and recording the number of posts from amongst them which contained a popular hashtag. There were a total of 4574 posts in this sample.

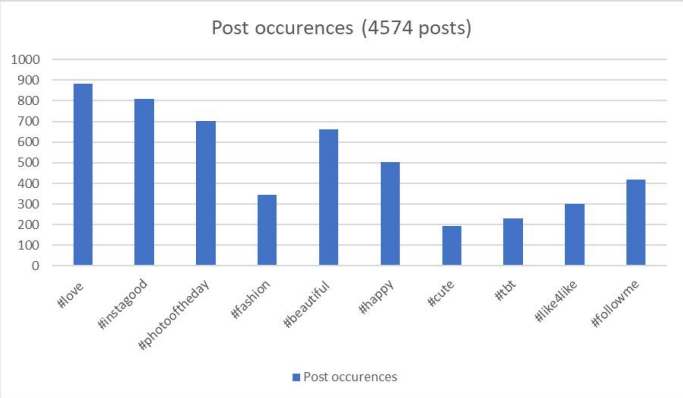


Figure 12: An experiment which involved crawling the profiles of 100 random active users and recording for each popular hashtag(in the top 10) the number of posts which reference it. There were a total of 4574 posts in this sample.

Fig 11 demonstrates that a significant number of posts contain at least one popular hashtag. Fig 12 examines this idea further by establishing a commonality between how popular a hashtag is and how frequent it comes up across random posts. But the problem identified in fig 6 still stands. The popularity is what makes a post difficult to find. When the crawler queries a popular tag like #instagood, the original post can no longer be retrieved due to the sheer amount of activity occurring on that channel. #travel is an example of a popular tag. The crawler was not able to find it after ten separate crawl instances, and #travel is only 1/10th active as #instagood. This approach may be ideal for a social media platform which does not constrain the way data is crawled.

4.3. Implementation

In this section we examine the rest of the attack and outline how it works in obtaining the user's identity.

4.3.1. Server

The server-side part of this attack will carry out a lot of the heavy computation requirements needed. The main components and their functions in this attack are defined below:

Google Functions - We have implemented two functions in typescript. Both are hosted on the Google platform. A server-side function is triggered by (HTTP) request. This allows our SCA app to make requests to our server and receive responses.

Google Vision - This allows us to generate labels for images.

Firebase Storage - This is used to store the images relating to a post for processing. It will allow our server functions to access these images remotely.

Firestore - This is used to store information on different users. Once we identify a user, we store their information on Firestore.

Phantombuster - Hosts the crawler that allows us to make crawl-requests programmatically. We can define which hashtags or locations we are interested in, and it would collect the data straight off Instagram and place it straight into the JSON file ready for processing.

We have described in full how the SCA app can detect when a user makes a post. The next part of the attack is locating that specific post on Instagram. We have implemented both LocationBasedSearching and HashtagBasedSearching approaches in our attack. We execute both in parallel to increase the chances of finding a post.

For LocationBasedSearching we must infer a set of location tags that the user might use. We must do it without requiring any additional permissions. For the HashtagBasedApproach we try to estimate what hashtags have been used by obtaining the generated labels associated with the image. We then compile the labels and the locations into a batch. We make individual crawl requests for each term in the batch and we do it in parallel. Only one term from the batch needs to result in the post being identified. The SCA app must prepare the batch with terms before the crawling can begin. Below are the steps involved in preparing the batch of terms.

4.3.2. Preparation of the Search Terms

1. The SCA app on detecting a post, first examines if the user making the post has not been identified by the app previously. It makes a request to Firestore to examine if there is a remote directory with a device id matching the current device. If no matches are found then the attack continues. The SCA app is able to generate a device id from the device's hardware primitives and associate it with a particular user. It does not require any permissions to obtain and is immutable in nature.

```
id = "" +
            Build.BOARD.length() % 10 + Build.BRAND.length() % 10 +
            Build.CPU_ABI.length() % 10 + Build.DEVICE.length() % 10 +
            Build.DISPLAY.length() % 10 + Build.HOST.length() % 10 +
            Build.ID.length() % 10 + Build.MANUFACTURER.length() % 10 +
            Build.MODEL.length() % 10 + Build.PRODUCT.length() % 10 +
            Build.TAGS.length() % 10 + Build.TYPE.length() % 10 +
            Build.USER.length() % 10; //13 digits
```

Figure 13: How the SCA app generates a device ID for a particular user.

2 (HashtagBasedSearching). The next part involves generating labels for a given image. As we explained in phase one, the pre-modified image of the post is stored in the DCIM camera directory. The SCA app sends this image to Firestore. Our Google Function method filter() is called from within the app with the url of the image being passed as a parameter. Within the filter() method we feed the url into the Google Vision API. It produces a collection of object, landmark, logo and web similarity-based labels for a given image and returns that data back to the Android SCA app.

2 (LocationBasedSearching). This part involves generating possible locations. This app uses two approaches. The first involves analysing the metadata of the photo stored within the DCIM camera directory. In phase one, this was discussed as occurring automatically when the user takes a photo with their Instagram client. This photo contains the location where the image was taken. It is defined in the form of longitude and latitude coordinates which can be retrieved through Android's ExifInterface. We utilise Geocode to reverse the coordinates to a particular location. The method call getFromLocation(lang, long, resultsSize) returns an array of addresses. For each address returned, we extract both its country and its feature name and add both to the batch (if not present).

The second approach used involves analysing the image itself. We can use machine learning techniques to determine if the content of the image is a landmark or location. If a famous landmark or area is identified, then we can construct an address.

3. Once we have the labels and locations, we send the post-modified image located in the DCIM Instagram directory to Firebase storage and obtain the resulting url. We are ready to make requests to our crawler.

4.3.3. Finding the Post

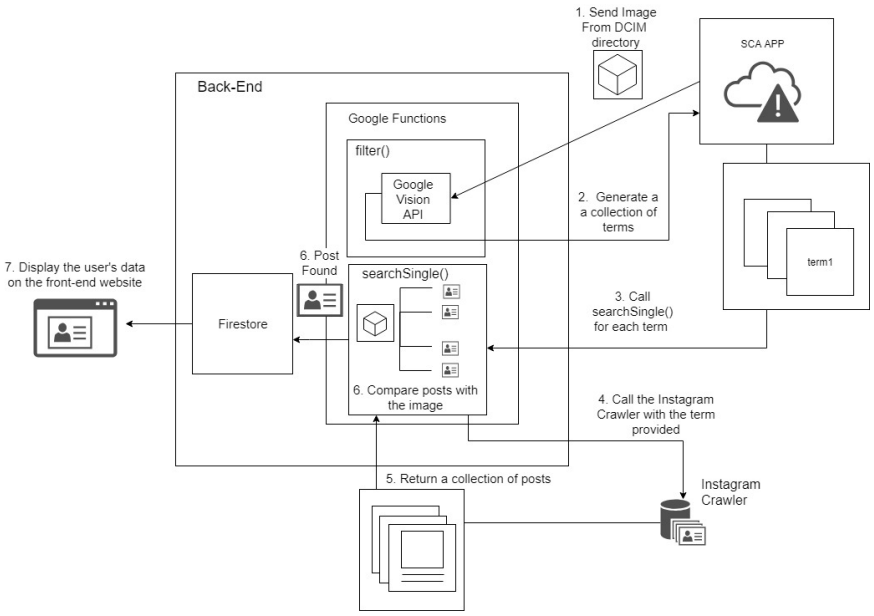


Figure 14: An overview of the components involved.

searchSingle() is a callable HTTP method. This method is responsible for making requests to the crawler. The parameters:

- tag - the tag is the parameter term that will define the request that needs to be made to the crawler. It can either be a hashtag(label) or a location.
- id - the device ID. On a positive match, it will be used to add a user entry into Firestore.
- url - the URL of the image that will be searched for from amongst the crawled posts.
- isLocation - if marked as true the tag is left how it is, but if false the tag is turned into a hashtag, i.e. Black is converted to #black.
- searchOccurences - Marks the number of unsuccessful attempts so far (additional optimisation)

The SCA app makes a searchSingle() request for every term identified, whether location or label. It goes through each item in the batch, and it makes a searchSingle() call. All calls are done in parallel, and it only requires one seachSingle() call to be successful at identifying the post for the attack to work.

How searchSingle() works:

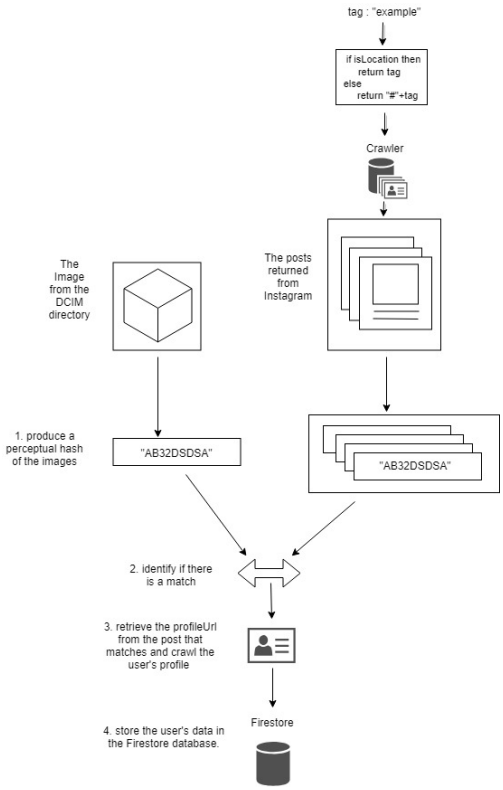


Figure 15: An overview of how the searchSingle() method works.

1. It first identifies whether the tag provided is a location from the isLocation parameter. If the tag is not a location (a label), then we convert it to hashtag form.

2. It generates the perceptual hash of the image that was passed as a parameter (url). This represents the image we are looking for.

3. It makes an HTTP request to the Instagram crawler with the tag contained within the request. If the search was successful it would return a collection of posts in JSON format.

4. It iterates through each post, generating the perceptual hash for each by using the postUrl key to download the image locally.

5. It compares the perceptual hashes of each post with the image hash defined in step 2. If there is a match, then the two images are equivalent, and the post has been found. That post moves on to the next stage.

6. The profileUrl is extracted out from the post.

7. All of the data is crawled from the user's profile and sent to Firestore. The algorithm then terminates.

8. If there are no matches, then the algorithm tries again with the same term. The searchOccurences parameter measures the number of runs that have been attempted. The algorithm currently only limits two tries before terminating. The term must also be a single word for a re-run to occur.

4.4. Displaying the Data

We have implemented a way of displaying data. The website displays the users data in readable format. The user ID is passed as a URL parameter to identify the user's data on Firestore.

5. Evaluation

In this paper we have a produced a way of obtaining the user's identity through several side-channels and various technologies and approaches. We can measure the reliability of this attack by examining how many posts on average must occur before the user can be identified. Due to the constraints imposed on fetching data from Instagram, there is no guarantee that the crawler will locate a post on Instagram. It might require several attempts before this is achieved. There is also no guarantee that the user has allocated any hashtags or locations to their image. It would be impossible in these cases for our crawler to find these posts. But these type of posts only represent a tiny number of posts on Instagram. Out of the 4574 posts that were examined, only 0.01% of them did not contain a location nor a hashtag.

The experiment involved simulating five different users on Instagram. We examined the number of attempts (the number of posts needed to be published) before the identity of the user was determined. The experiment involved re-posting the user's posts back to back. The SCA app only detects posts for which a photo was taken for, and this involves utilising the camera. To trigger the SCA procedure we took photos of the images used. We attributed the same hashtags and provided the same location. The five users that were chosen were randomly selected from amongst a sample of 100 users.

User	Amount of posts detected on device	Amount of posts with no match (fail)	The number of matched terms that fail	The matching term on success	Amount of posts till a match (user is identified)
1st	100%	1	0	#buckinghampalace	2
2nd	100%	6	1 (#flower)	#kiss	7
3rd	100%	0	0	#vehicle	1
4th	100%	0%	N/A	N/A	N/A
5th	100%	3	1 (#style)	#party	4

Figure 16: An experiment which simulated 5 different users found on the Instagram platform on a device with the SCA app running in the background.

From the data, the crawler can locate the post made whenever a match occurs. The number of matched terms that fail were reasonably low. The problem arises when the user-defined description is not descriptive enough. Only a minority of posts have a tag that can be identified by its media content. For the fourth user, no matches were made before the tenth post. The user was attributing context-specific labels to their posts, and a large section of their profile consisted of self-portraits.

On average the user was identified after 3.5 posts. This can be improved if a more informed and robust approach was taken to attributing labels. More extensive analysis of how users make posts on Instagram would need to occur to form a greater understanding of how to assign possible terms. The attack currently assumes that the user only attributes tags based on how accurately it describes the image content. We do not consider the context surrounding a post.

A case might arise where the user might be using a private account. When their account is private, only people they approve are able to view their posts. These posts can never be retrieved by the crawler. Instagram has not released any official statistics.

6. Protecting From This Attack

In this section, we discuss some of the ways Android can thwart this type of attack. This will involve blocking and restricting certain channels on the Android operating system.

6.1. Protecting From Instagram Public Directory Leakage

To stop this attack, It would be enough to block the Instagram public directory from being accessed by other apps. The SCA app would not have access to the image that was posted on Instagram, and could not establish a link with a remote post on the Instagram platform. This would involve Android introducing isolated containers within the external storage. In Android Q (9), a similar approach is taken. Apps can specify access to their the external storage partitions. For Instagram this approach would involve storing images in its sandbox. This would be in contrast to storing it in the DCIM public directory. However, these isolated sandboxes do not exist independently from the app. When the app is uninstalled from the device, its partition is automatically cleaned.

If this is adopted, it might restrict apps that require photo directories to function. Photo editing apps command a large market on the Play Store, and all of them require the DCIM directory. Any updates made to the Android OS must strike the right balance between the functionality it offers and the security it provides the overall system with.

An alternative approach is to attribute ownership to an image and define how apps can access that image. This will introduce a more fine-grained approach to external storage which takes into account the context of the file. When the app writes a file to memory, it can specify the permission context of that file. The permission context can be categorised as

a user-defined set of permissions on the Android system. A third-party app trying to access that file must have obtained all permissions within that set. Instagram can utilise this feature by associating a permission context {READ_EXTERNAL_STORAGE, CAMERA} with the images that are added to the DCIM Instagram directory. It will allow access from photo editing apps that might require those files whilst preventing access from the SCA app. The SCA app does not have the CAMERA permission required to access the file.

Another approach would be for Android to close public storage listeners that are trying to run in the background. The malicious app might find alternative ways of detecting when a change has been made to the external storage that work just as effectively. The Android system must be able to detect those occurrencess and close them.

6.2. Protecting From Camera Activity Leakage

The camera availability listener can be blocked by putting the method registerAvailability-Callback() of the CameraManager class behind a permission wall. No third-party app should be able to detect when a camera is available if they do not have the permissions necessary to use the hardware camera.

6.3. Protecting From Camera Public Directory Leakage

There are some use cases to why a third party app might need the DCIM camera directory to store photos. However, even in those cases, it should not automatically store photos without the user's consent. The user should explicitly define whether a photo should be saved to their public directory. In the case of Instagram a photo should only be stored to the DCIM camera directory if the user later shares the post for which the picture was taken for.

6.4. Protecting From GPS Leakage From Image Metadata

No third-party app should have the ability of geotagging images that are stored within the external storage. The geotagging information should be removed from all images deposited to the public directory. This information should not be available to any apps that have access to that directory.

7. Future Work

7.1. Extending this Attack

The attack in its current state links a user to a specific device. This grants access to all the data contained within that user's profile along with the posts they have posted. The posts themselves contain useful information that can allow us to derive reliable inferences. We can utilise various NLP approaches to carry out sentiment analysis on that data. Sentiment

analysis is the automated process of associating content with a particular mood. We can analyse a range of posts belonging to a user and identify the characteristics it conveys.

7.2. Generalising this Attack for other Media-sharing Apps

We can generalise the side-channel attack proposed to work for other media-sharing apps, like Tumblr or Pinterest. These platforms utilise hashtags in the same way Instagram does. We can detect when a post has been made by exploiting the same combination of side-channels. We also have access to the official API which will allow us to crawl posts effectively. We can compile several of these attacks across different platforms into one fully functioning app.

8. Conclusion

Over time Android has constricted specific channels on their operating system. It was primarily done to limit the number of leaks that occur. Some channels must remain open, and play a significant role in a multi-app operating system like Android. We have shown how we can exploit these channels to carry out a attack that detects when a user makes a post. We also built on it by developing an attack which can obtain the user's identity by linking the local event (user making a post on a device) with a remote event (identifying the equivalent post) on Instagram. Even with the constrictions imposed on the crawler by Instagram, the attack still remains fairly reliable. There are cases where the attack does not work well but those cases can be reduced with more dynamic approaches. We can generalise this attack for other media-sharing apps that have more open APIs. It would be easier in these cases to obtain the identity of the user. We also have discussed some possible ways of thwarting against this attack.

References

[1] "android m will never ask users for permission to use the internet, and that's probably okay", https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise, accessed: 30-04-2019.

[2] J. Levin, Android Internals: A Confectioner's Cookbook Volume I: The Power User's View, 1st Edition, Technologeeks.com, 2015.

[3] K. Yaghmour, Embedded Android, 1st Edition, O'Reilly, 2013.

[4] N. Elenkov, Android Security Internals : An In-Depth Guide to Android's Security Architecture, 1st Edition, Cox publishing, 2015.

[5] A. et al., "sok: Lessons learned from android security research for appified software platforms", Tech. rep., IEEE Symposium on Security and Privacy (2016).

[6] "application sandbox", https://source.android.com/security/app-sandbox, accessed: 28-04-2019.

[7] "permissions overview", https://developer.android.com/guide/topics/permissions/overview, accessed: 23-04-2019.

[8] "global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 2nd quarter 2018 ", https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/, accessed: 30-04-2019.

[9] A. H. S. E. D. W. Primal Wijesekera, Arjun Baokar, K. Beznosov, "android permissions remystified: A field study on contextual integrity ", Tech. rep. (2015).
[10] "cyber attacks on android devices on the rise", https://www.androidpolice.com/2015/06/06/android-m-will-never-ask-users-for-permission-to-use-the-internet-and-thats-probably-okay/, accessed: 30-04-2019.
[11] e. a. Michalevsky, Yan, "powerspy: Location tracking using mobile device power analysis.", Tech. rep., USENIX Security Symposium (2015).
[12] e. a. Zhou, Xiaoyong, "identity, location, disease and more: Inferring your secrets from android public resources.", Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM.
[13] "what is instagram?", https://help.instagram.com/424737657584573, accessed: 30-04-2019.
[14] "android camera2basic sample", https://github.com/googlesamples/android-Camera2Basic, accessed: 30-04-2019.
[15] "instagram is limiting how much data some developers can collect from its api and cutting off others altogether", https://www.vox.com/2018/4/2/17189512/instagram-api-facebook-cambridge-analytica, accessed: 19-04-2019.