Lucas Palmer, 4A BMath (AMATH, CS, CM)
Undergraduate Research Assistant, Winter 2017
Supervisor: Urs Hengartner

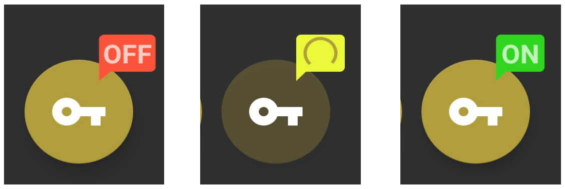
Privacy Guard is an Android application that analyses the network traffic of all applications on a smartphone in order to detect leakage incidents. For the purpose of this report, a leakage incident (or "leak") refers to when an application sends sensitive user information across the network (such as contact, location, or device information). The first goal of this URA was to polish Privacy Guard's user interface using modern trends in Android programming. The second and more interesting goal was to give the user the power to visualize, summarize, and analyze leakage incidents.

1.) First of all, the main activity was cluttered with several features that various contributors to the project have added along the way. These features were moved into a settings activity that is accessible through the Android Options Menu.

The main activity has one main function: turn on and off the VPN that monitors network traffic. This functionality was moved into a Floating Action Button (common Android widget in many applications such as Gmail, Facebook, YouTube).

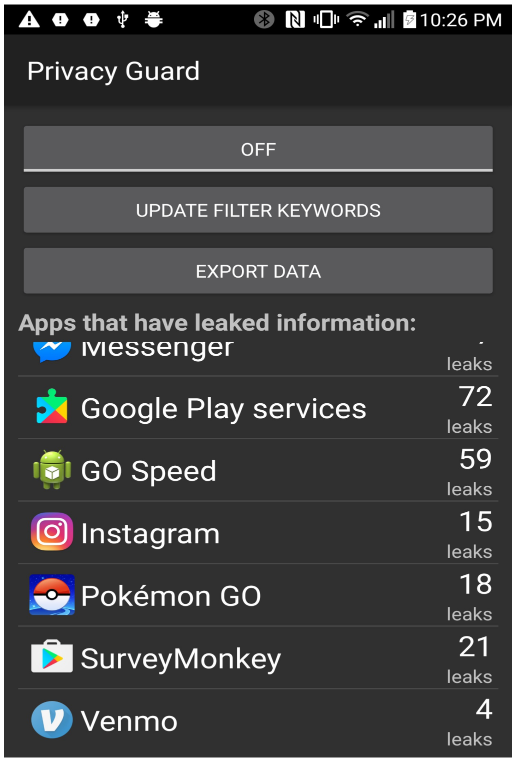
This Floating Action Button displays an "OFF" message when the VPN is off and an "ON" message when the VPN is on. When the VPN is starting up (for about 5 seconds after it is turned on), a loading animation is displayed and the button is disabled. This sequence of states was added to prevent the user from trying to turn off the VPN while it was starting up (an action that caused ambiguous behaviour in the application).

Figure 1: VPN Custom Widget



Implementation note: when the VPN is fully running, it broadcasts a custom intent. A broadcast receiver is registered in the main activity to listen for this broadcast and change the VPN status from the loading animation to "ON" and re-enable the button.

Before:



10:26 PM

Privacy Guard

OFF

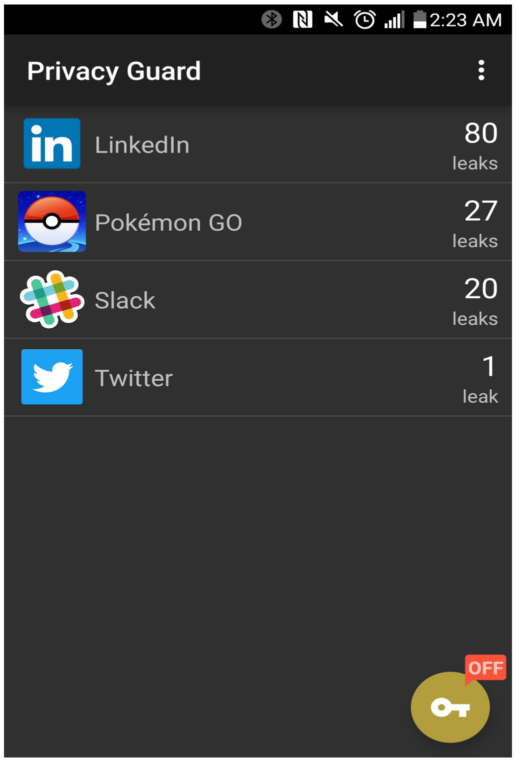
UPDATE FILTER KEYWORDS

EXPORT DATA

Apps that have leaked information:

Messenger	leaks
Google Play services	72 leaks
GO Speed	59 leaks
Instagram	15 leaks
Pokémon GO	18 leaks
SurveyMonkey	21 leaks
Venmo	4 leaks

After:



2:23 AM

Privacy Guard

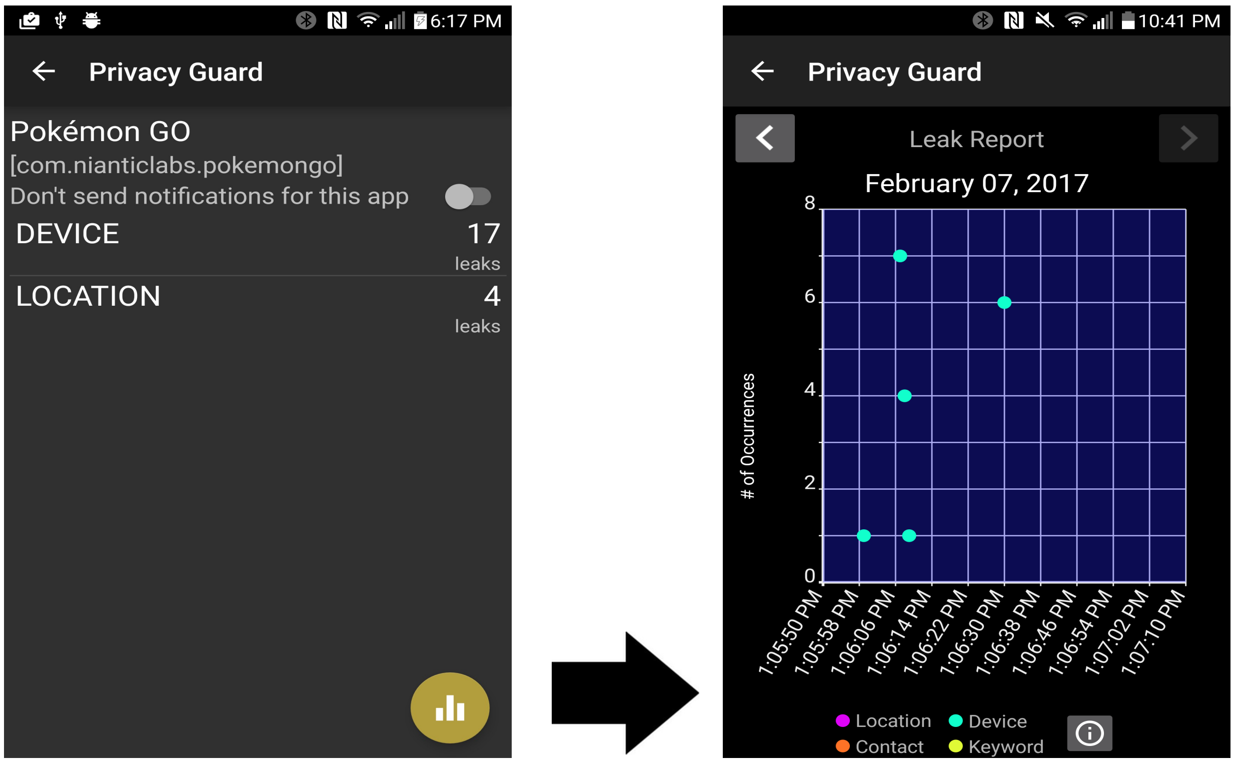
LinkedIn	80 leaks
Pokémon GO	27 leaks
Slack	20 leaks
Twitter	1 leak

OFF

2.) Privacy Guard records data leaks. However, the user really has no way of viewing or analyzing these leaks other than just viewing them in a list. The first idea was to plot leaks on a graph with time on the x-axis and the number of leaks on the y-axis. The library Android Plot was used.

> i.) Problem: Leaks are generally sparse. A user typically only uses an application several times in one day, and most applications don't leak a lot of data. So plotting all the leaks for a single application on one graph compresses the data to the point where it is not very insightful.
>
> ii.) Solution: Give the graph a small domain size and make the most recent leak in the center of the graph. Then allow the user to cycle through which leak is in the center of the graph using arrow buttons. Allow the user to navigate to the graph from an individual application summary activity. Lastly, allow the user to change the domain size from the settings screen.

Figure 2: Initial Leak Report Prototype



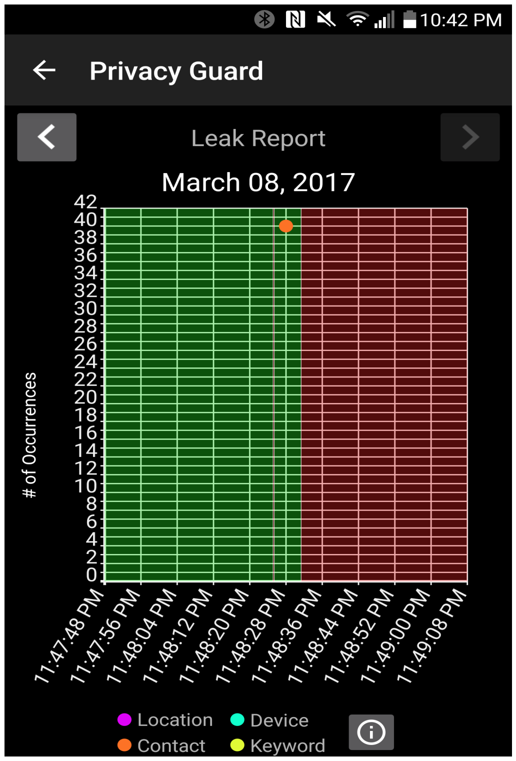
Pokémon GO
[com.nianticlabs.pokemongo]
Don't send notifications for this app

DEVICE	17 leaks
LOCATION	4 leaks

3.) Next, the data that Privacy Guard records was combined with other useful data: the status of an application (ie. foreground or background) when it is leaking. As it turns out, Android records status events for all applications. Using this Android API, timestamps for when applications move into the foreground and into the background can be obtained. Then, these timestamps can be plotted on the same graph as the leaks so that the user can see when their application is leaking data compared to when they are using it.

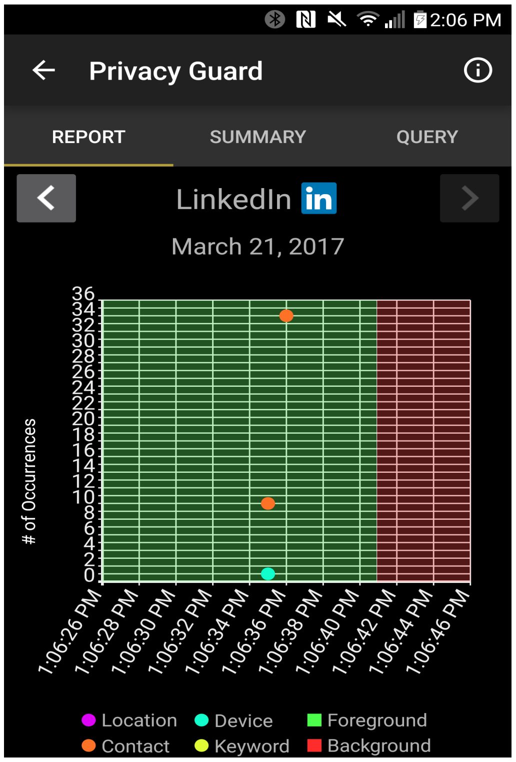
Figure 3: Updated Leak Report

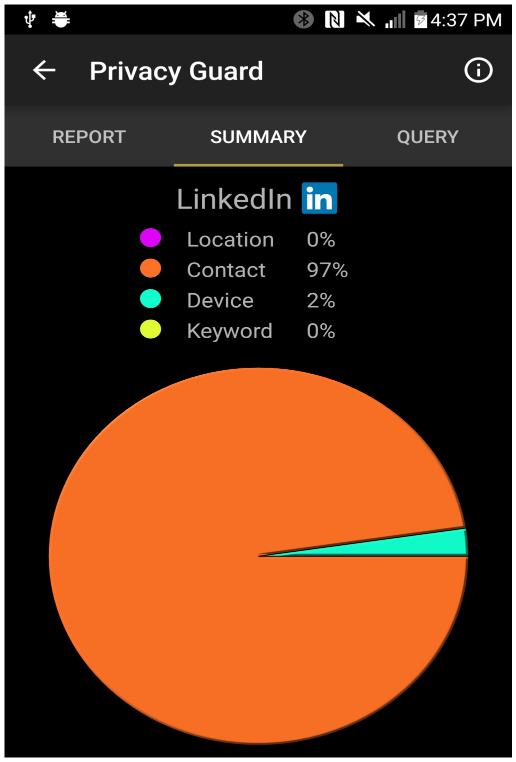
i.) Green: Application is in the foreground.
ii.) Red: Application is in the background.



4.) The leak report graph can be useful for visualizing leaks. It would also be useful to summarize the data for the user and give them the ability to define custom queries on the database. Hence, a tab layout with a Report Tab (the graph seen previously), a Summary Tab, and a Query Tab was created.

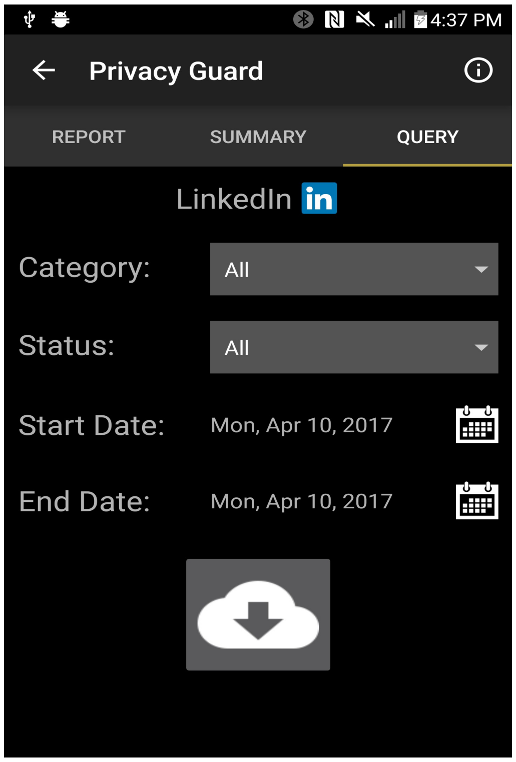
Figure 4: More User Functionality





●	Location	0%
●	Contact	97%
●	Device	2%
●	Keyword	0%

Figure 4 continued:



Category: All

Status: All

Start Date: Mon, Apr 10, 2017

End Date: Mon, Apr 10, 2017

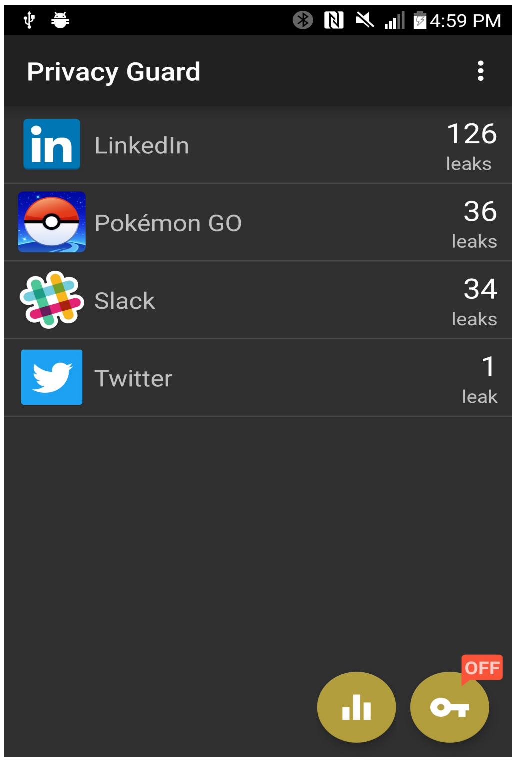
a.) Report Tab: Plot leaks and background/ foreground status events.

b.) Summary Tab: Show proportion of leaks in each category.

c.) Query Tab: Query database for leaks with various filtering parameters such as time frame, category, and (addressed later in the report) application status (ie. foreground or background).

5.) This functionality could be useful for all data leaks rather than just one specific application. Thus, allow the user to navigate to a similar data analysis view from the main activity.

Figure 5: User Functionality for All Applications



LinkedIn	126 leaks
Pokémon GO	36 leaks
Slack	34 leaks
Twitter	1 leak

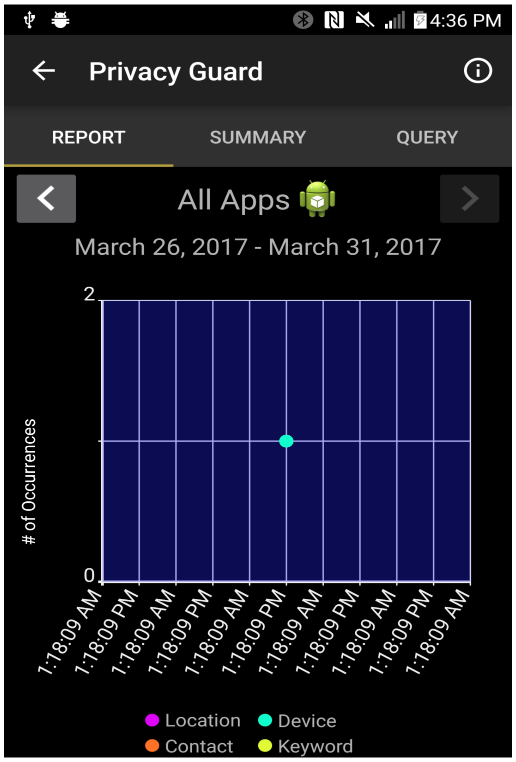
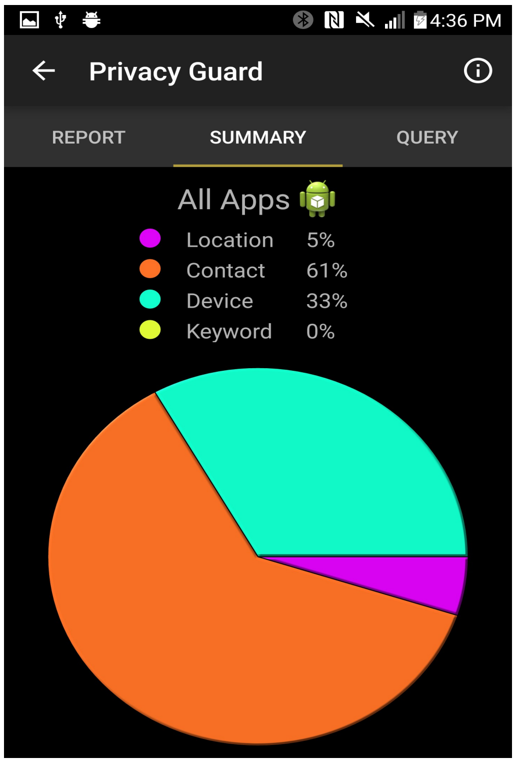
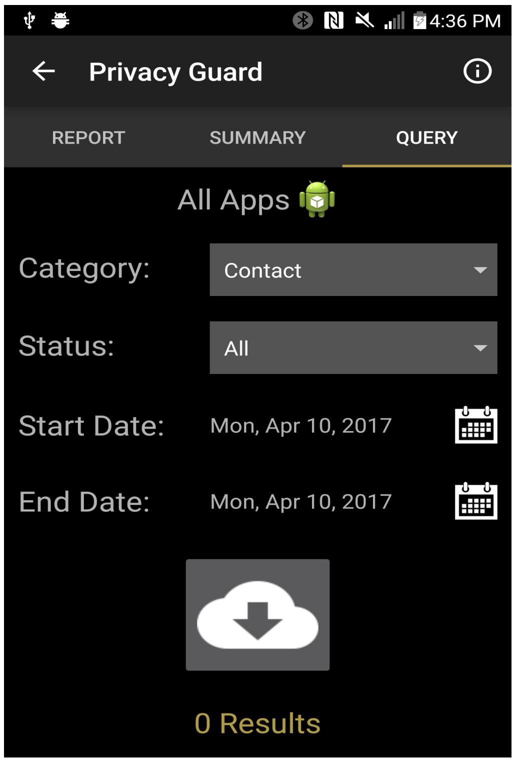


Figure 5 continued:



●	Location	5%
●	Contact	61%
●	Device	33%
●	Keyword	0%



Category: Contact

Status: All

Start Date: Mon, Apr 10, 2017

End Date: Mon, Apr 10, 2017

0 Results

6.) Right now, the only information contained in the database for each leak is: application name, package name, timestamp, category, type, and content. Next, the application status (ie. foreground or background) will also be recorded for each leak. This will allow leaks to be summarized by application status in addition to category. As well, the application status can be added as a filter parameter in the Query Tab.

Using the background/foreground status events recorded by Android, each leak can be classified as having occurred in the foreground or background. The timing of this calculation is important.

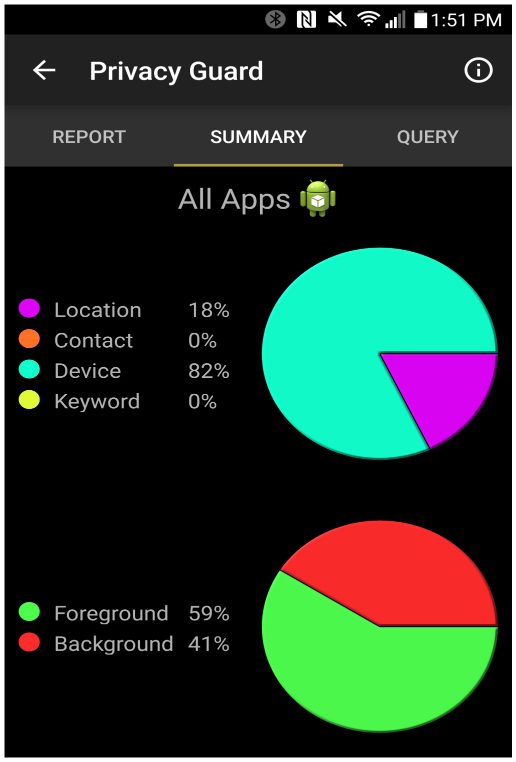
One possibility is to do the calculation every time that the application status of a leak is needed. But then a potentially long computation would be completed every time that this information was needed to summarize data or perform a database query based on application status.

Instead, the leak should be classified as foreground or background every time that a leak is recorded in the database. This means just adding an additional column to the leak table in the database. However, this calculation should not be done on the same thread that the VPN code in running on because the VPN is already handling a lot of traffic.

Hence, after a leak is written to the database, the leak database id is passed to an Asynchronous Task that will perform the computation and update the database on a background thread.

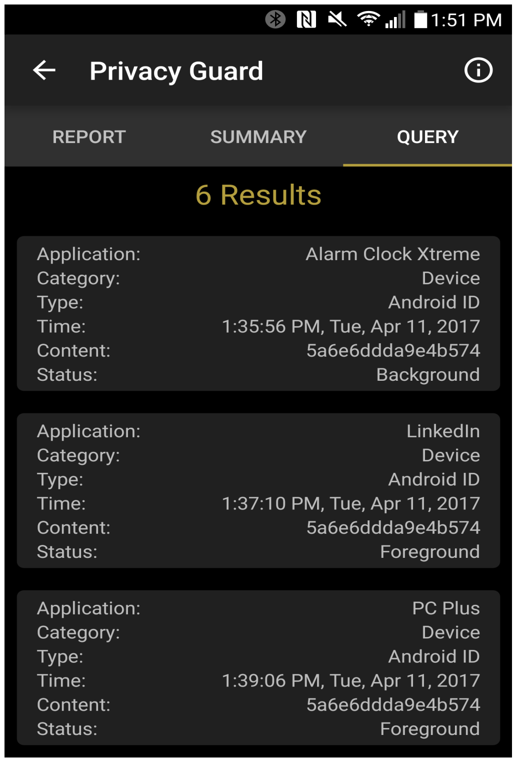
With this new data available, the application status of a leak can be used to summarize leak data and to filter queries.

Figure 6: Classifying Leaks by Application Status



●	Location	18%
●	Contact	0%
●	Device	82%
●	Keyword	0%

●	Foreground	59%
●	Background	41%



6 Results

Application:	Alarm Clock Xtreme
Category:	Device
Type:	Android ID
Time:	1:35:56 PM, Tue, Apr 11, 2017
Content:	5a6e6ddda9e4b574
Status:	Background

Application:	LinkedIn
Category:	Device
Type:	Android ID
Time:	1:37:10 PM, Tue, Apr 11, 2017
Content:	5a6e6ddda9e4b574
Status:	Foreground

Application:	PC Plus
Category:	Device
Type:	Android ID
Time:	1:39:06 PM, Tue, Apr 11, 2017
Content:	5a6e6ddda9e4b574
Status:	Foreground

7.) Up until this point, the foreground/background status information that Android records was used to display the application status on the leak report graph and to classify leaks as having occurred in the foreground or background. Android only records this information for seven days in the past.

In terms of classifying leaks, this isn't an issue. The Asynchronous Task that uses this data to update a leak in the database runs rather quickly after a leak is recorded. So seven days in the past is more than enough.

However, this in not the case for graphing the application status events on the leak report graph. Leaks can be recorded an arbitrary amount of time in the past. So in order to always show the application status events on the graph, seven days is not sufficient. The application status events must be recorded in Privacy Guard's database in order to ensure that they aren't lost when Android gets rid of them.

My first thought was to set up an alarm that will run approximately every 7 days and record all of the relevant application status events in the database. The requirements are as follows:

> i.) The database update needs to be run at least once every 7 days.
> ii.) This scheduled update needs to be persisted across device reboots.
> iii.) There isn't a specific time that this database update needs to occur. Perhaps this flexibility could allow us to pick a time that is better than others (ie. according to phone charging status, how active the phone is).

These requirements led to the Google Cloud Messaging API. This API allows you to schedule periodic tasks as follows:

Figure 7: Scheduling a Periodic Task with GCM

```
GcmNetworkManager manager = GcmNetworkManager.getInstance(this);
PeriodicTask task = new PeriodicTask.Builder()
        .setService(RecordAppStatusService.class)              *1
        .setTag(RECORD_APP_STATUS_TAG)
        .setPersisted(true)                                     *2
        .setPeriod(TimeUnit.DAYS.toSeconds(5))                  *3
        .setFlex(TimeUnit.DAYS.toSeconds(1))                    *4
        .build();
manager.schedule(task);
```

*1: The service that will update Privacy Guard's database.
*2: Persist this scheduled task across device reboots.
*3: Perform this task at least once every 5 days.
*4: Set a flexibility of 1 day and allow the API to decide the best time to run the task.

I actually did not end up using this API because it currently requires the Google Play Store application to be installed on the phone. Most Android devices would have the Google Play Store application installed. But I didn't want to add an additional dependency if it wasn't necessary. Perhaps in the future this API will be more independent and can be used. Moreover, it captures the essence of what needs to be done.

Now for the solution that was actually used: while examining various Android alarm APIs that are dependent on Android version and harder to manage because of issues like persistence, I decided to just use an Android Broadcast Receiver (added in API level 1).

Android Broadcast Receivers can be configured to run when various events occur (such as when the user turns the phone on or off). The triggering event chosen for the database update is: ACTION_USER_PRESENT. This means that every time the user turns on the phone and unlocks the home screen, my receiver will be run. Updating the database every time the user unlocks their home screen is overkill. As a result, the last time the database was updated is stored in Privacy Guard's shared preferences. When the receiver is run, it first checks the last time that the database was updated. If it was updated less than 5 days ago, no work is done. If it's been more than 5 days, the database is updated.

So essentially Privacy Guard relies on the user to unlock their phone at least once every couple of days in order to record this data. And if the user doesn't unlock their phone, then there isn't any relevant application status data to record anyway.

8.) Previously, only the UI thread accessed the database. Now, an Asynchronous Task and a Broadcast Receiver access the database on different threads. This could cause synchronization issues. Privacy Guard uses a SQLite database, which luckily handles synchronization as long as only one database access handle is used.

Solution: Use the singleton pattern as described here:
http://www.androiddesignpatterns.com/2012/05/correctly-managing-your-sqlite-database.html

Figure 8: The Singleton Pattern in Privacy Guard

```
//The singleton pattern is used here because multiple threads could potentially be writing to
the database.  Because of this pattern, only one DataBaseHandler is created for the
application lifecycle.  As a result, do not call .close() on a DatabaseHandler instance.
public static synchronized DatabaseHandler getInstance(Context context) {
      // Use the application context, which will ensure that you
      // don't accidentally leak an Activity's context.
      // See this article for more information: http://bit.ly/6LRzfx
      if (sInstance == null) {
            sInstance = new DatabaseHandler(context.getApplicationContext());
      }
      return sInstance;
}
```

9.) Privacy Guard needs many permissions to function. You can locate all of these permissions in the AndroidManifest.xml file. Android divides these permissions into 2 categories: normal permissions (that do not directly risk a user's privacy) and dangerous permissions (that give the application access to a user's confidential data).

When the application is installed, normal permissions will be granted to the application automatically. However, Beginning in Android 6.0 (API level 23), users must grant dangerous permissions to applications while the application is running. Privacy Guard has 6 dangerous permissions. This means that when Privacy Guard is initially installed, none of its dangerous permissions are granted and parts of the application won't function.

There are 2 general ways of handling these permissions:

i.) The first is to write the application to partially function when it is lacking permissions. Whenever the user tries to evoke functionality that uses a permission, use a system dialog to prompt the user to grant the permission. If they grant access, then enable the feature. If they deny access, keep the application running with that particular feature disabled.

ii.) Prevent the application from functioning unless the user grants all the required permissions. When the user opens the application, check for required permissions and prompt the user to grant them (again using a system dialog that Android provides).

I chose to go with option ii.) because all of the dangerous permissions in Privacy Guard were not added by me. I'm not exactly sure how instrumental they are for the application to function. I believe it to be a lot safer to require the user to grant all of the required permissions rather than to account for all the edge cases needed to fail gracefully in the case of permission denial.

The solution works as follows:

When the user opens Privacy Guard for the first time, the application will prompt the user to grant each permission, one at a time. If the user denies a prompt, the same prompt will just be shown again. To get past a prompt, you need to grant the permission. This looks like the following:

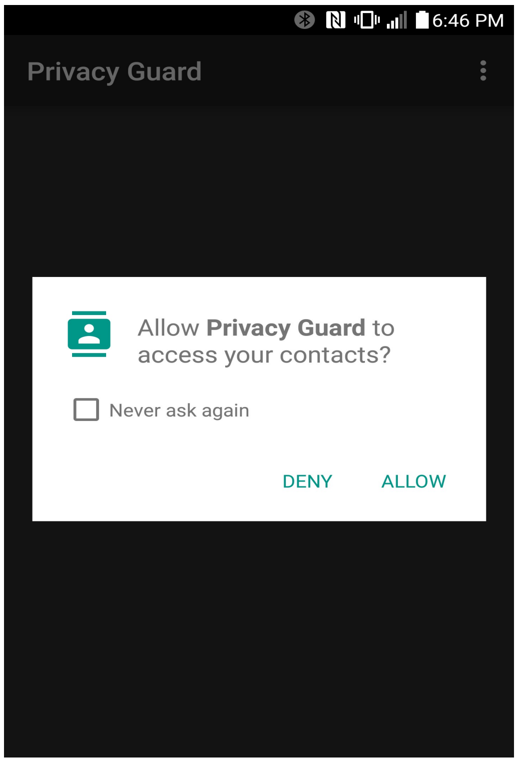


Figure 9: Permission Prompt

Notice how the application is inaccessible to the user while permissions are being requested.

Here's where things get a little more interesting: note the “Never ask again” option on the permission system dialog in Figure 9. If the user selects that option, then the system will not allow Privacy Guard to prompt the user for that particular permission ever again. The only way for this permission to now be granted is for the user to navigate to the application settings and turn on the permission manually. Hence, if the user ever selects “Never ask again”, the permission dialogs stop and the user is presented with this:

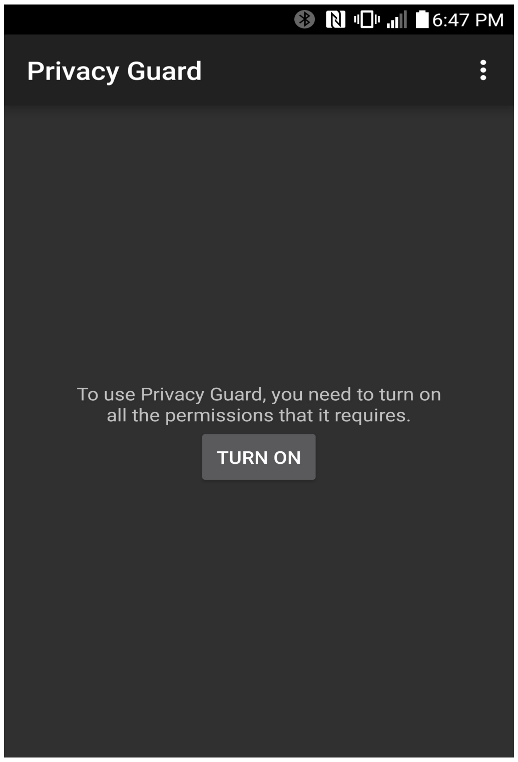
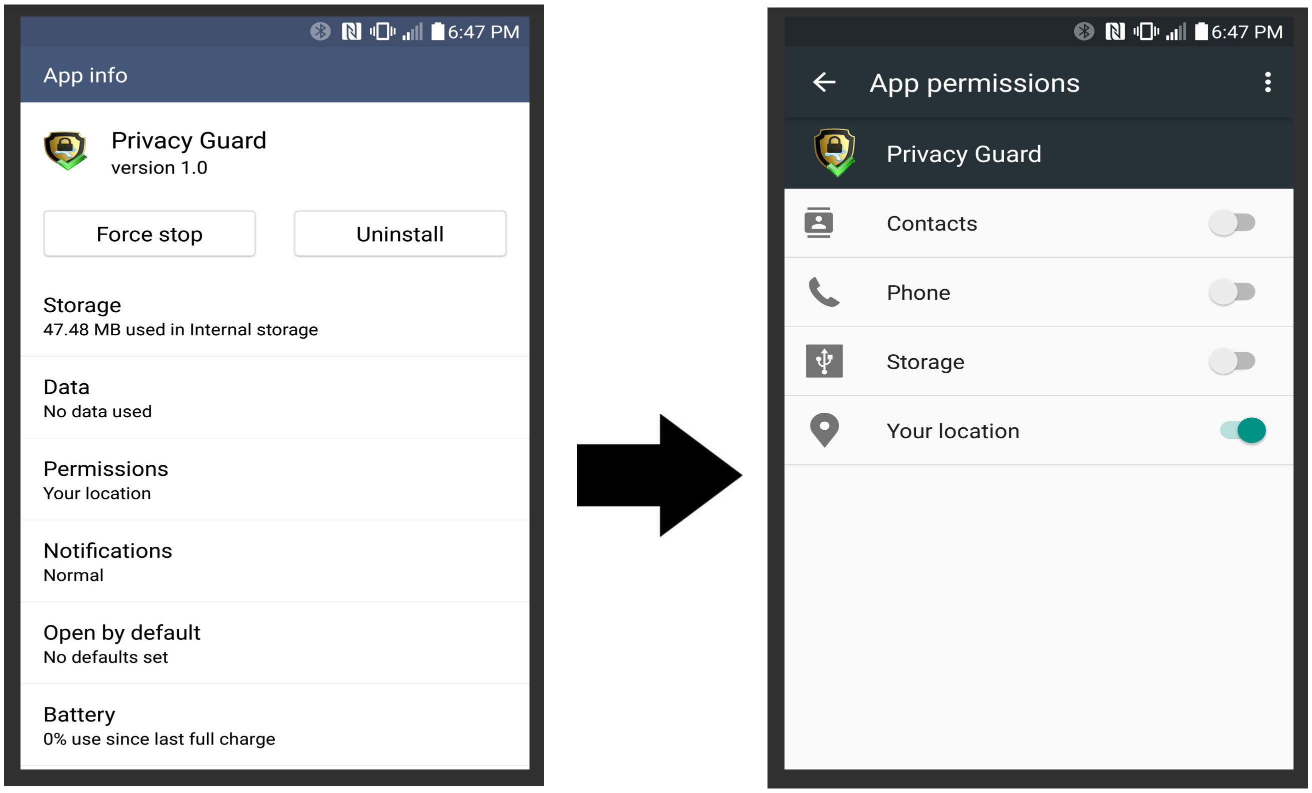

6:47 PM
Privacy Guard
To use Privacy Guard, you need to turn on all the permissions that it requires.
TURN ON

Figure 10: User Selects “Never ask again”

Once again, the application is inaccessible to the user because required permissions are not granted.

If the user selects the “TURN ON” button, they will be directly sent to the application settings where they can turn on the permissions manually.

Figure 11: Application Settings


6:47 PM

App info

Privacy Guard
version 1.0

Force stop | Uninstall

Storage
47.48 MB used in Internal storage

Data
No data used

Permissions
Your location

Notifications
Normal

Open by default
No defaults set

Battery
0% use since last full charge

6:47 PM

App permissions

Privacy Guard

Contacts

Phone

Storage

Your location

The majority of the time, a user will just accept the permission prompts when they install Privacy Guard and never worry about it again. If, however, the required permissions aren't granted, unpredictable behaviour will be avoided by preventing the user from accessing the application. In the future, if you add a dangerous permission to Privacy Guard, make sure you modify the code in MainActivity.java in order to include your permission in the system prompts that I have described above. For specifics on implementation, visit the following link:

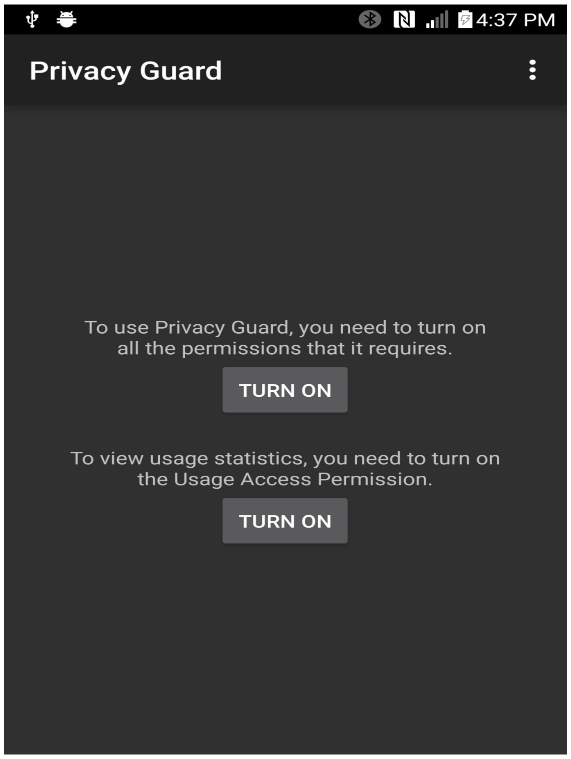
https://developer.android.com/training/permissions/requesting.html

10.) On a related note to the application permissions described above, Privacy Guard also requires a quite different permission. In order for the application to access the application status events that are plotted on the leak report graph and used to classify leaks as having occurred in the background or foreground, they must enable the “Usage access for apps” permission for Privacy Guard. Although this permission is also declared in the AndroidManifest.xml, it is not located in the application settings. Rather, it is located under “Settings”->“Security”->“Usage access for apps”. Furthermore, there are no Android system dialogs (such as the one in Figure 9) in order to prompt the user to grant the permission. The user must be directed to the security settings and grant the permission manually.

Similarly to application permissions, I require the user to grant the usage access permission in order to gain access to the application. This prevents the user from easily turning on the VPN before the usage access permission is granted, which could cause leaks to become “unclassified” (in terms of foreground or background) in the database.

I require the usage access permission to be set from the main activity in the same workflow as the application permissions. In order to gain access to the rest of the application, the user must not only turn on the application permissions, but also the usage access permission.

Figure 12: Requiring Usage Access Permission



To use Privacy Guard, you need to turn on all the permissions that it requires.

To view usage statistics, you need to turn on the Usage Access Permission.

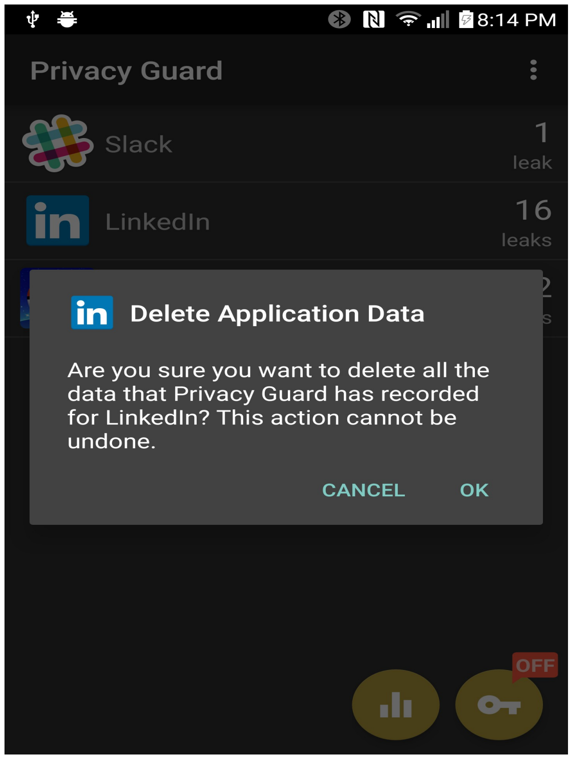
Note: If the application permissions are turned on and the usage access permission is turned off, then only the second message and button will show. Similarly, if the usage access permission is turned on and the application permissions are turned off, then only the first message and button will show.

Of course, the user could always turn on the VPN, navigate to the security settings by themselves, and then turn off the usage access permission. In this (rare) case, a leak's status will be “unclassified” as foreground or background. I can avoid this misclassification from persisting by checking for any unclassified leaks and correcting them every time that the user navigates to the leak report/summary/query screen. (Note: the user can only access the leak report/summary/query screen if the permission is turned on, so correcting the database at this time is valid.)

11.) Finally, it was necessary to give the user the ability to delete data that Privacy Guard has recorded. The simplest form of this, and quite useful indeed, is to allow the user to delete all the leaks that have been recorded for a single application.

Now, when the user long clicks (a prolonged click) on an application summary item in the main activity list of all applications, a dialog will appear giving them the option to delete all the data that Privacy Guard has recorded for that particular application.

Figure 12: Delete Application Data Dialog



Delete Application Data

Are you sure you want to delete all the data that Privacy Guard has recorded for LinkedIn? This action cannot be undone.

Should the user select “OK” in this dialog, all the data that has been recorded in the database pertaining to this application will be deleted. This means all data leaks, all application status events that have been recorded (as described previously in this report), etc. will be deleted for this particular application. Note: should more database tables be added in the future, make sure to modify this delete functionality accordingly. We don't want useless data sitting in the database for an application whose leaks have all been deleted.