Leak Finder: A tool for Dynamic Analysis of Android Applications

Nikolaos Kofinas
University of Maryland, College Park
nikofinas@gmail.com

ABSTRACT

In this project we implemented Leak Finder, a tool for examining how Android applications use their granted permissions. Leak Finder instruments Android applications with custom log code, then it collects the logs from actual user application runs and uses these logs to create an execution graph. Auditors can use the executing graph to examine when an applications tends to use its permissions. We used Leak Finder to audit eight different Android applications. From our results we verified that Leak Finder can give insighting the way that the application handle their permissions. We made an interesting observation that many permissions are used in a non-user interactive way.

1. INTRODUCTION

Android applications which use permissions can have access sensitive user data such as the user's location, contacts, text messages, etc. The current Android permission system asks the user only once to give permission to an application to access their data, and then the application can do anything with that data. This raises the question: how do we know exactly what an application does with the permissions it has acquired?

This question is very difficult to answer by reading the description page of an application. As an example, we can look at a Battery Saving application,[1] whose purpose is to enhance the battery life of the mobile device. Surprisingly this application requires almost all Android permissions. While it seems reasonable for it to acquire the `Device & app history` permission (which allows the application to gather information about the running applications), there is no obvious reason it needs permission to access the contacts, photos and phone data of the user.

Another example is a wedding planner application,[2] which asks for the `Contacts` permission. By simply reading the description of the application, it is not obvious why this permission is needed. By navigating through the different app screens after installation, though, we find a screen that allows the user to create a list of their wedding guests. In that screen, the user is presented with all their contacts and is able to select who to add to their guest list. While this seems to be a "correct" use of the `Contacts` permission, we do not know if the application used the `Contact` permission only on that screen and not, for example, to send the contacts to a server at some other time.

From these two examples we see that from the description alone we cannot determine if the applications use the permissions in a "correct" way. Even when we start using an application, we cannot know if it uses the permissions in the places that we understand as the "correct" places to use them. This is a problem with the current state of the Android permission system because the users don't know how an application is handling their private data.

As a solution to this problem, in Section 2 we propose a dynamic analysis of Android applications which shows when and why each of its permission is used. Our goal is to find out if there is a relation between permission usage and user interaction with the GUI elements of the applications.

For our analysis we implemented a tool named Leak Finder. Leak Finder works by injecting logging code inside an Android application that we want to audit. The logs generated from this application can then be used to visualize its execution graphs. This graph contains nodes that represent GUI calls, permission-protected calls and activities/threads generation calls. The edges of the graph represent sequence of events. As an example, an edge from a GUI node to a permission Node represents that the user interacted with the application and then a permission protected method was called.Thus, auditors can use these graphs to determine when each permission is used inside the application.

We conducted an internal survey between two people to find out if by using Leak Finder we could understand where and why each permission is used. The method that we followed was to create a survey with questions about the usage of permissions. Then we answered this survey twice, once after looking at the description of the application and using it, and once by looking at the graphs generated by Leak Finder.

From this survey we validated first that Leak Finder can help auditors to decide if a permission is necessary for an application to function. Also, we observed that programmers tend to use permission-protected calls immediately when the application start, even if this is not necessary, without a specific user interaction first. In Section 3 we present the methodology that we followed and our results.

2. IMPLEMENTATION

Our implementation of Leak Finder consists of various parts that handle the application instrumentation with log calls, log collection and log visualization. Figure 1 depicts the architecture.

[1] https://play.google.com/store/apps/details?id=com.dianxinos.dxbs

[2] https://play.google.com/store/apps/details?id=com.wepala.weddingplan

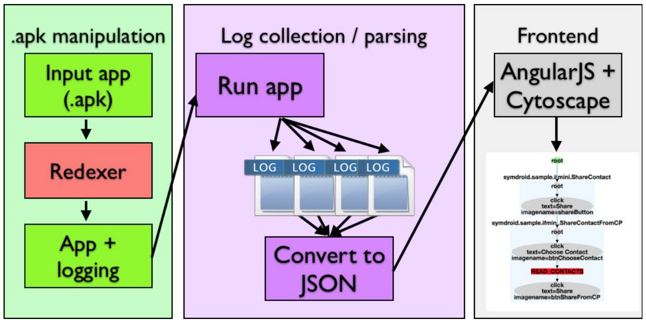


Figure 1: The architecture of Leak Finder

2.1 APK Manipulation

The first component of our implementation is the instrumentation of existing Android apps shown in the left box in Figure 1. We use Redexer [4] to inject logging code which surrounds every method call that we want to capture. Redexer is a toolkit capable of transforming Dalvik, the byte-code which is the compiled code found in an Android Application APK file.

We implemented a static logging library in Java, and we use Redexer to inject log calls around relevant Android API calls and at the start/end of relevant user-written methods. The logging library implements two general log methods, one to log the entry and one to log the exit from the corresponding method/API, with void return type. The signature of the logging method for the entry is the following (it is the same for the exit):

```
static void logEntry(String cname,
                     String mname,
                     Object...  args)
```

These log methods take as input the callers object class name, the name of the method, a reference to the caller object, and all the arguments of that call, or in the case of log exit method, the return value of the corresponding method.

For each argument, if it is not a primitive type we log specific attributes corresponding to the type of that argument. If it is primitive we log its actual value. Some examples of such attributes for non primitive objects are the class name, the object id, the resource id if the object is of type `android.ui.view`, the text if the object is an `android.ui.view` item with a text field, a captured screen-shot of the current activity if any of the arguments is a GUI element (such as activity or view), etc. Finally, we log the thread id in which the method was called. This allows Leak Finder to disambiguate between calls coming from the main UI thread and and those that come from a running background thread.

Our first approach was to surround every API and method call with logging code, but we found this adds too much overhead to the execution of the applications. The reason for this overhead is that the logging class investigates each argument, extracts information from it, and finally prints it. The overhead of doing this for each API call made some applications unusable. Thus, we use a configuration file to define regular expressions of methods/API calls that we want to log. Currently for user methods, we only care about GUI event handlers (such as onClick, onTouch, etc.), activity life-cycle methods (such as onStart, onResume), and thread starting methods (such as async.onPreExecute, async.doInBackground, etc). In Section 2.3.1 we explain this list in more detail.

We also need to log all API calls which are protected with a permission by the Android system. We used the PScout [1] permission mapping list, which contains a large subset of the Android calls that are protected under the Android permission system. This mapping also includes method calls that require a permission only when a specific argument is provided (e.g., a specific `URI`). An example of such a call is:

`android.content.ContentResolver.query(...)`

which when it is called with a specific `URI` equal to:

`content://com.android.contacts/contacts`

requires the `READ_CONTACTS` permission.

Additionally, we need to log all the methods that are involved in the life-cycle of an activity but may not be implemented by the programmer. While these methods may not be necessary for the actual application, we need them to determine the boundaries of each activity. We observed a similar problem with the life-cycle of ASync Tasks. In that case, the programmers only implemented the `doInBackground` method and not the `onPreExecute` and `onPostExecute` methods which define the start and the end of an ASync Task. For this reason, before the logging step, Redexer injects an empty implementation of such missing methods (the only code inside these methods is a call to the `super` method of the parent class).

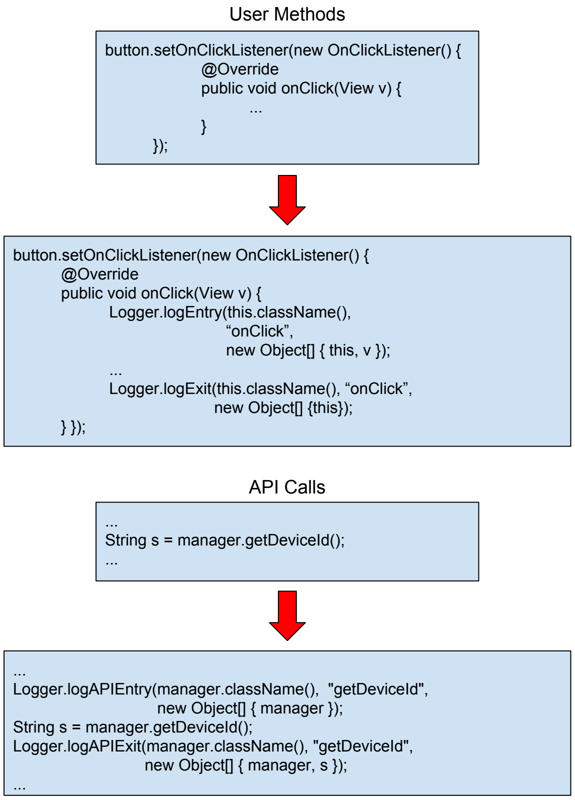
Having these configuration files, we can use Redexer to instrument the application. Redexer takes the following steps to instrument an application are the following:

1. Unpack the application APK
2. Load the byte-code from the `classes.dex` file
3. Traverse all the classes
4. If class is of type `Activity` or `ASyncTask`, add missing life-cycle methods
5. Use the configuration file to identify the methods of interest
6. Surround them with our logging code

Figure 2 presents two examples of what the code looks like inside the new APK. In the first example, there is a user method that implements the `onClick` listener on a specific button. Redexer added code at the start and at the end of the method body. As we mentioned above, the logging method takes as input the callers class name, the method name, and an object array which contains the caller object and the input argument of the method. Similar, the logging method at the end of the method body takes the same first two inputs and an array containing only the caller object reference since there is no return value for this method. In the second example Redexer surrounded an API call with our logging code. The main difference compare to the previous example is that the object array in the `logExit` methods contains the object that stored the return value of the method, in this case the `String s`.

2.2 Back-end

After we create the final re-instrumented apk file, we distribute the application to users. At any time, we can collect



```
button.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                ...
            }
      });
```

```
button.setOnClickListener(new OnClickListener() {
      @Override
      public void onClick(View v) {
            Logger.logEntry(this.className(),
                           “onClick”,
                           new Object[] { this, v });
            ...
            Logger.logExit(this.className(), “onClick”,
                          new Object[] {this});
      } });
```

```
...
String s = manager.getDeviceId();
...
```

```
...
Logger.logAPIEntry(manager.className(),  "getDeviceId",
                  new Object[] { manager });
String s = manager.getDeviceId();
Logger.logAPIExit(manager.className(), "getDeviceId",
                 new Object[] { manager, s });
...
```

Figure 2: Example of logging code

the generated logs from their device by using the `adb logcat` command that it is provided with the Android SDK and redirect the output to a file. This procedure is illustrated in the middle box of Figure 1.

We have implemented a back-end server to load these log files, clean all the irrelevant log lines (e.g., system logs), and pull the screen-shots that our code captured during the app execution. The screenshot pulling can be done by using the `adb pull <filename>` command from the Android SDK. It is a separate step from the log collection because the screenshots are saved as `.jpeg` files in the user storage of the device. The back-end also provides the necessary data to the front-end, which is designed as a web-app, to construct the execution graphs.

The back-end is implemented in Ruby and uses the Sinatra REST API to handle HTTP requests from the front-end. After it receives a request for a specific application log, it first transforms the corresponding logs from our own format into JSON format. During this transformation, it annotates each method call as a “GUI”, “permission” or “other” call.

2.3 Front-end

Leak-finder comes with a front-end designed as a website which a user can use to examine logs, explore the execution graph and see the permissions that each application uses. The implementation of the front-end is based on the angular-js framework[3] and we also use the CytoscapeJS library[4] to visualize our execution graphs.

2.3.1 *Graph Generator*

Part of the front-end is responsible for transforming the logs to the execution call graph. Figure 3 presents a simple graph created from logs that we collected from a test application that we handcrafted. This graph presents the following sequence of events:

1. User opens the application
2. `ShareContact` activity is instantiated
3. User clicks a button which code name is `shareButton` with text `Share`
4. `ShareContactFromCP` activity is instantiated
5. User clicks a button which code name `btnChooseContact` with text `Choose Contact`
6. An API call protected with the `Read_Contacts` permission is executed
7. User clicks a button which code name `btnShareFromCP` with text `Share`

From this graph we can see that the application accessed the users contacts only after the user interacted with the application.

More generally, the nodes of the graph represent events which are logged method calls. The edges represent ordering of events, e.g., an edge from $node_1$ to $node_2$ means that the event encapsulated by $node_1$ occurred just before the event encapsulated by $node_2$.

Graph Nodes. From Figure 3 we see that our execution graphs contain three distinct kind of nodes:

1. GUI nodes which represent GUI method calls (onClick, onTouch, etc). They are drawn with circular nodes.
2. Permission nodes which represent method calls that require a permission. They are drawn with red rectangular nodes.
3. Outer Nodes, which represent an activity or a thread and are drawn with light blue rectangular nodes and contain sub-nodes. These sub-nodes are GUI and/or permission nodes. The outer nodes encapsulate calls that originate from an activity or thread.

Activities. To create an activity node we need to find out when an activity came to the foreground and when it left. We know from the life-cycle of an activity that `onResume` is called before the activity comes to the foreground, and the first method that is called when the activity leaves the foreground is `onPause`. When we create the graph, we look for these calls, and whenever we find an `onResume` call we create a new activity node. Then we insert into it all the future nodes that we will add to the graph until we observe an `onPause` call. The `Entry` node inside each activity node represents the entry point inside this sub-graph.

[3] https://angularjs.org/

[4] http://js.cytoscape.org/

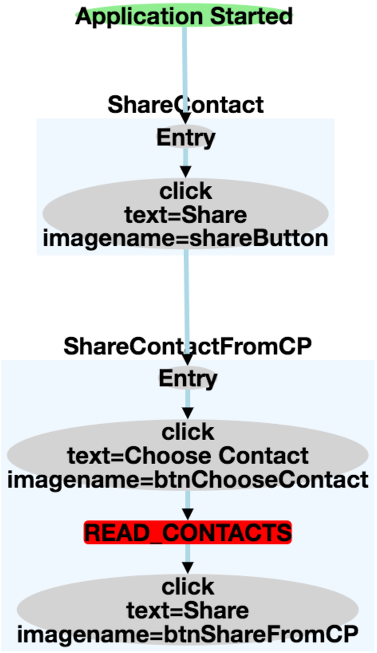


Figure 3: An example of an application execution graph

Threads. Android executes all the GUI-specific calls in the main thread of the application. Programmers need to start new threads whenever they want to execute a method which may block the main thread and "freeze" the application. For this reason, some calls protected under permissions must be called in a thread because they will block the main thread (e.g., loading an image from the users photo album, which is protected under the `READ_EXTERNAL_STORAGE` permission). To handle this kind of behavior, we need a different kind of node to represent threads and the calls they contain. We must also know where exactly a thread was created (not where it started) to generate the correct graph. This is necessary if we want to determine if there was a GUI interaction before a thread with a permission call inside started.

While extracting the thread in which a method was called is trivial since we log the thread id of each call, finding the place where the thread was created is more complicated. In the Android system users can create two type of threads, normal Java threads and also Async tasks.

Async tasks have a lifecycle similar to activities. First the `onPreExecute` method is called in the main thread, and then the `doInBackground` method is called on a thread. The `onPreExecute` method may or may not be implemented by the program, and thus we again use Redexer to insert an empty `onPreExecute` method if the programmer chose not to insert one. When we find an `onPreExecute` call inside our logs we create a new thread node and then we add to it all the subsequent calls that happened in that thread.

Java threads do not have a similar life-cycle, but the programmer first needs to call the `start` method. Then when the thread is ready the `run` method starts automatically. By capturing these two calls, we can create the corresponding thread nodes.

Node Merging. As application is used, the user may navigate multiple times to the same activity and click the same button more than once. Creating a new node for each of these calls would create a large graph that is difficult to understand. Thus, we merge nodes which represent the same activity, thread, GUI, or permission call.

Activity merging is trivial since we can coalesce activities with the same name (note the class name is unique for each activity). This is not true for GUI objects, which may be inside multiple activities (the same object) or share resource ids with other GUI objects that appear in other activities. Merging two GUI nodes should happen only if they are in the same activity, have the same resource id, and finally have the same properties (e.g., same text, same image, etc.). Having the same properties is important since it is possible the programmer changed the appearance of a GUI element after an event and thus, from the user's perspective, it is a different GUI element.

The logic behind merging permission nodes is different. We observed that, with high probability, after a method which is protected by a permission is called, there will be more method calls protected under the same permission. This leads to graphs with many permission nodes that do not provide any useful information. Thus, we merge sequences of permission nodes if they are for the same permission.

Thread Merging. In our execution graphs we merge threads that contain the same code because it is easier for the end-user to understand which threads are instantiated often. Thread merging is similar to activity merging because we know the two methods that are executed when a thread starts, e.g., `doInBackground` and `run`. We use the name of the class that contains these methods to find and merge threads together.

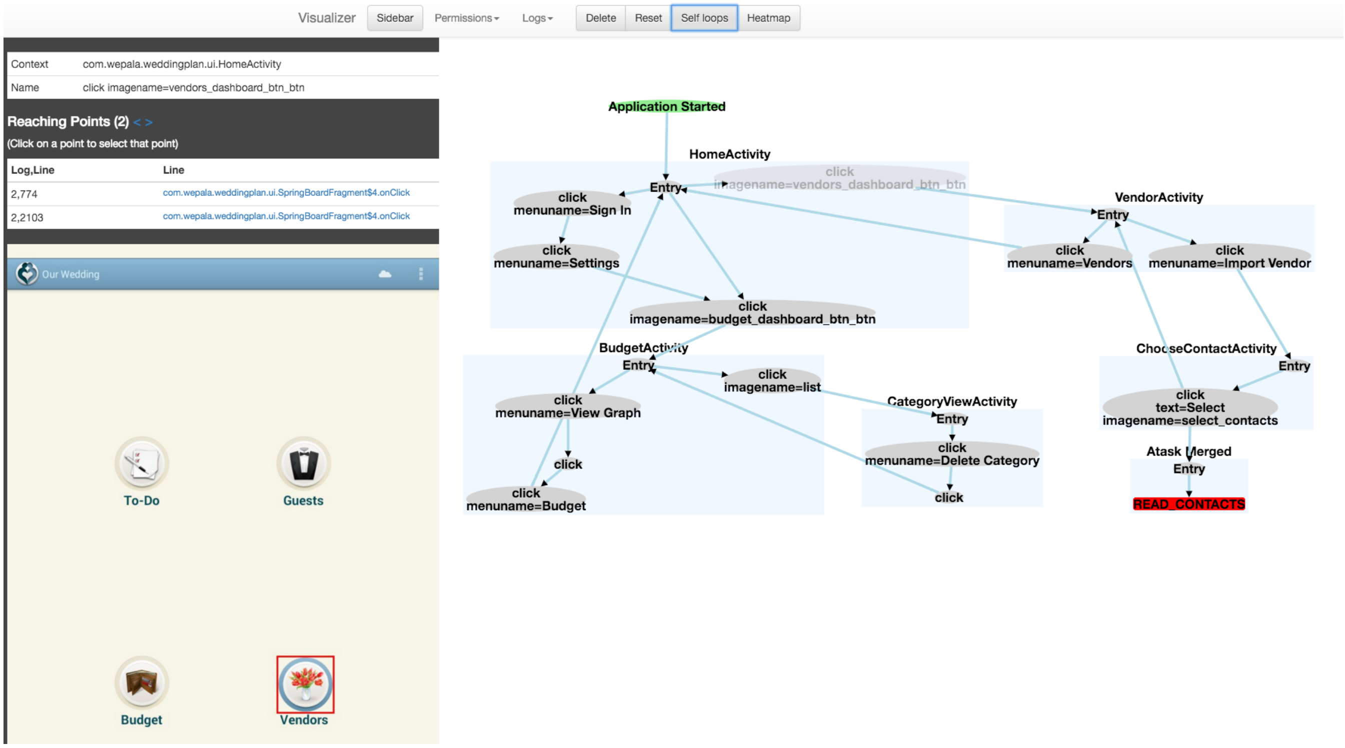
2.4 Graph Interactivity

Users can observe more information by clicking each node. For each node, we present to the user the method calls that are represented by that node. If it is a GUI node we present the screenshot we captured during the execution of the application, annotated with the location of the GUI element and the context of that node (e.g., the thread/activity in which this event occurred). Figure 4 presents such an example in which the user clicked on the GUI node inside the "ShareContact" activity.

3. RESULTS

To test our approach, we conducted a study involving two persons that worked on this project. For this study we selected eight random applications from the Google Play store, and for each one we filled out a survey. The questions inside the survey were about the timing and the reason each permission was used by each application. Some key questions inside the survey are the following:

- Is there an obvious reason that the application needs the permission `Contacts`, intrinsic to the app's core functionality?



Context	com.wepala.weddingplan.ui.HomeActivity
Name	click imagename=vendors_dashboard_btn_btn

Reaching Points (2) < >

(Click on a point to select that point)

Log,Line	Line
2,774	com.wepala.weddingplan.ui.SpringBoardFragment$4.onClick
2,2103	com.wepala.weddingplan.ui.SpringBoardFragment$4.onClick

Figure 4: Front-end of our implementation

- What do you believe is the use of the permission `Contacts` in the application?
- Why do you think the permission `Contacts` is/isn't used in an interactive way?

These questions answered not only for `Contacts` but for all the permissions that the corresponding application uses. The complete survey can be found in the following link: https://umdsurvey.umd.edu/SE/?SID=SV_es12ExCennv9JOt.

We answered the questions of this survey twice, once by only looking at the description and then using the application and once after reviewing the output of Leak Finder.

3.1 Survey Results

Figure 5 presents the results for the question "Do you think the app uses the permission in an interactive way?". We observe that when we first answered this question by only reading the description and using the application, we believed that most permissions were used only when we interacted with GUI element of the application. Then, when we explored the graphs generated by Leak Finder we were surprised to observe that most of the permissions were accessed immediately when the application started and not in the place that we thought the application used them. Also, we observed that all access except these initial ones where non user-interactive.

This was common for internet relative permissions since most of the applications had advertisements but it is strange that the same happened for the location permission. Our belief was that the location will only get accessed when we navigated into an activity which contained a map or location base information but in reality the applications started reading our location immediately. We don't know exactly the reason that the application behaved in that way since we didn't observe any indication that the location was used for advertisement.

The second question that gave us interesting results is the following: "Do you think the app uses this permission needs this permission?". Figure 6 visualizes the answers for this question. Our initial answers and the answers that we gave after we observed the output of Leak Finder were the same. The execution graphs increased the confidence of our answer because we observed were and for what each permission was used.

As an example, the "Gas Buddy" application uses the `Location` permission whenever we search for a gas station near us. While it still reads the location at other irrelevant points during the execution, this functionality is important for the application to achieve what it advertises. Another interesting example is the permission `Bluetooth` when it is used from the Battery application. In that case we cannot be absolute confident why it needs the permission. Our assumption is that it needs that permission to calculate the estimated battery life, but the execution graph did provide us with enough evidence to support that our claim is right or wrong.

Finally, we categorized each permission usage into five categories:

1. Used for location adds
2. Used in an interactive way
3. Used in a non-interactive way to update a GUI element
4. Used in an non-interactive way in the background but the effect was observable by the user

5. Used but not observable by the user

Figure 7 presents our visualized results for this question. We observed that by using Leak Finder we increased our confidence to our answers and also we ruled out some categories. An interesting insight is that as users we expect the internet permission to be used for UI updating. What we observed with Leak Finder though was that the applications access the internet everywhere and the user does not have an indication that this is happening.

4. RELATED WORK

Pegasus. Chen et al. [2] defined the Permission Event Graphs (PEGs) which present what permission may be used after each event. They define events in the same way that we do. The created a tool named Pegasus which they use to build this graphs, and then they use an auditor which creates linear-time temporal logic (LTL) policies [7] to define when a permission is used inside the application. The main difference of this tool compare to Leak Finder is that it does a data-flow analysis on the target application and for that reason it cannot be used for examining large applications.

Pidgin. Johnson et al. [5] designed a tool to explore the information flow of various Android applications in an interactive way. The tool, Pidgin, allows an auditor to use a query language to find possible ways that sensitive information leaks from the application. It also allows the auditor to ensure that inputs of the applications are declassified before they released. The main difference of Pidgin with Leak Finder is that our tool is based on the user's perspective of events sequence while Pidgin is a information flow representation on a source level.

TaintDroid. Enck et al [3] implemented a system named TaintDroid which tracks the information-flow during the execution of an Android application. It can detect when sensitive information is sent over insecure channels (e.g., internet connections). Leak Finder doesn't track the information flow of variables during the execution containing sensitive information but instead it tracks sensitive method calls.

Interaction-Based Declassification Policies. Micinski et al [6] introduced a formal way to define policies that allow release of sensitive information base on user interaction. In their work, a user can define using LTL formulas specifying when a secret input can be released. Leak Finder can be used along side their work to examine the execution graph of applications and then, base on the observations, a user can design such LTL formulas.

5. CONCLUSION

In this project, we implemented Leak Finder, a tool that contacts dynamic analysis on Android applications. Tested Leak Finder on eight applications downloaded from the Google Play store. we found that programmers prefer to call permission protected method in a non interaction-based way, which contradicts the way that we believe users want their private data to be accessed.

Future Work. We need to find out if the statement that we made when we defined our problem holds, are the user expect the permission usage to be interacted-based? To answer this question we will design a survey which we will present to auditors. The survey will contain a slide-show of screen-shots from actual application runs. After some specific screen-shots we will asks the user to answer questions about permission usage. These questions will state some permissions, and user picks which of them should be used by the application at the current state. To avoid the confusion of the users, we will not use the actual permission names but instead a more user-friendly description of each permission.

We will upload this survey to MTurk,[5] and we will ask from approximately 200 users to answer it. We will investigate the users answers to find out how users believe an application should handle their permissions.

6. REFERENCES

[1] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[2] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual policy enforcement in android applications with permission event graphs. In *NDSS*, 2013.

[3] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

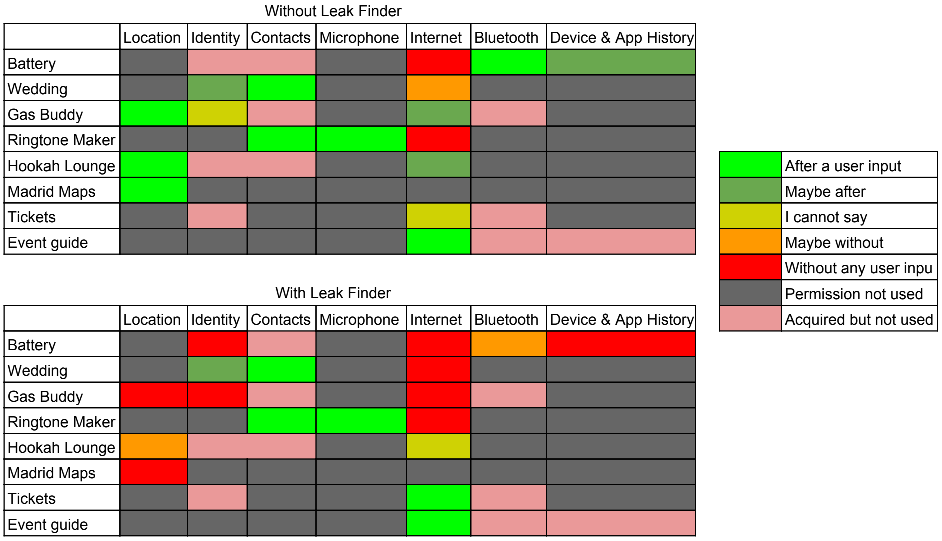
[4] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

[5] A. Johnson, L. Waye, S. Moore, and S. Chong. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–302. ACM, 2015.

[6] K. Micinski, J. Fetter-Degges, J. Jeon, J. S. Foster, and M. R. Clarkson. Checking interaction-based declassification policies for android using symbolic execution. In *Computer Security–ESORICS 2015*, pages 520–538. Springer, 2015.

[7] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[5] https://www.mturk.com/mturk/welcome



Without Leak Finder

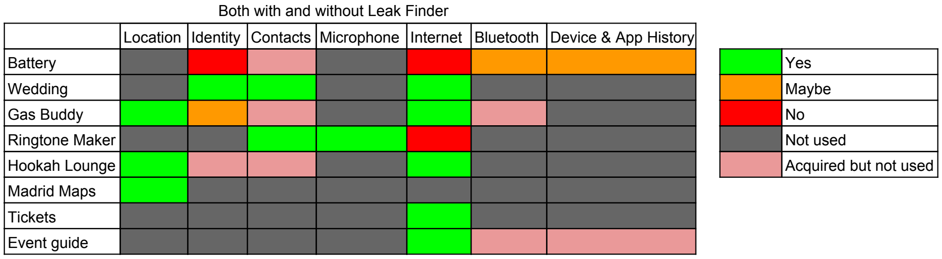
	Location	Identity	Contacts	Microphone	Internet	Bluetooth	Device & App History
Battery	⬛	🟪	🟪	⬛	🟥	🟩	🟢
Wedding	⬛	🟢	🟩	⬛	🟧	⬛	⬛
Gas Buddy	🟩	🟨	🟪	⬛	🟢	🟪	⬛
Ringtone Maker	⬛	⬛	🟩	🟩	🟥	⬛	⬛
Hookah Lounge	🟩	🟪	🟪	⬛	🟢	⬛	⬛
Madrid Maps	🟩	⬛	⬛	⬛	⬛	⬛	⬛
Tickets	⬛	🟪	⬛	⬛	🟨	🟪	⬛
Event guide	⬛	⬛	⬛	⬛	🟩	🟪	🟪

With Leak Finder

	Location	Identity	Contacts	Microphone	Internet	Bluetooth	Device & App History
Battery	⬛	🟥	🟪	⬛	🟥	🟧	🟥
Wedding	⬛	🟢	🟩	⬛	🟥	⬛	⬛
Gas Buddy	🟥	🟥	🟪	⬛	🟥	🟪	⬛
Ringtone Maker	⬛	⬛	🟩	🟩	🟥	⬛	⬛
Hookah Lounge	🟧	🟪	🟪	⬛	🟨	⬛	⬛
Madrid Maps	🟥	⬛	⬛	⬛	⬛	⬛	⬛
Tickets	⬛	🟪	⬛	⬛	🟩	🟪	⬛
Event guide	⬛	⬛	⬛	⬛	🟩	🟪	🟪

🟩	After a user input
🟢	Maybe after
🟨	I cannot say
🟧	Maybe without
🟥	Without any user inpu
⬛	Permission not used
🟪	Acquired but not used

Figure 5: Answers for the question: Do you think the app uses the permission in an interactive way?

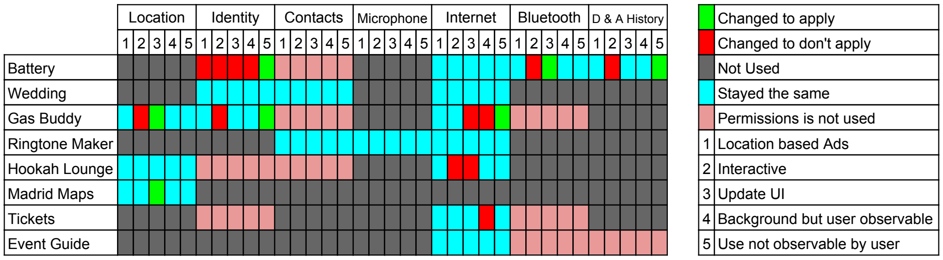


Both with and without Leak Finder

	Location	Identity	Contacts	Microphone	Internet	Bluetooth	Device & App History
Battery	⬛	🟥	🟪	⬛	🟥	🟧	🟧
Wedding	⬛	🟩	🟩	⬛	🟩	⬛	⬛
Gas Buddy	🟩	🟧	🟪	⬛	🟩	🟪	⬛
Ringtone Maker	⬛	⬛	🟩	🟩	🟥	⬛	⬛
Hookah Lounge	🟩	🟪	🟪	⬛	🟩	⬛	⬛
Madrid Maps	🟩	⬛	⬛	⬛	⬛	⬛	⬛
Tickets	⬛	⬛	⬛	⬛	🟩	⬛	⬛
Event guide	⬛	⬛	⬛	⬛	🟩	🟪	🟪

🟩	Yes
🟧	Maybe
🟥	No
⬛	Not used
🟪	Acquired but not used

Figure 6: Answers for the question: Is this permission required for a core functionality?



	Location					Identity					Contacts					Microphone					Internet					Bluetooth					D & A History				
	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Battery	⬛	⬛	⬛	⬛	⬛	🟥	🟥	🟥	🟥	🟩	🟪	🟪	🟪	🟪	🟪	⬛	⬛	⬛	⬛	⬛	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟩	🟦	🟦	🟦	🟥	🟦	🟦	🟩
Wedding	⬛	⬛	⬛	⬛	⬛	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	⬛	⬛	⬛	⬛	⬛	🟦	🟦	🟦	🟦	🟦	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛
Gas Buddy	🟦	🟥	🟩	🟦	🟦	🟦	🟥	🟦	🟦	🟩	🟪	🟪	🟪	🟪	🟪	⬛	⬛	⬛	⬛	⬛	🟦	🟦	🟥	🟥	🟩	🟪	🟪	🟪	🟪	🟪	⬛	⬛	⬛	⬛	⬛
Ringtone Maker	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛
Hookah Lounge	🟦	🟦	🟦	🟦	🟦	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	⬛	⬛	⬛	⬛	⬛	🟦	🟥	🟥	🟦	🟦	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛
Madrid Maps	🟦	🟦	🟩	🟦	🟦	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛
Tickets	⬛	⬛	⬛	⬛	⬛	🟪	🟪	🟪	🟪	🟪	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	🟦	🟦	🟦	🟥	🟦	🟪	🟪	🟪	🟪	🟪	⬛	⬛	⬛	⬛	⬛
Event Guide	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	🟦	🟦	🟦	🟦	🟦	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪

🟩	Changed to apply
🟥	Changed to don't apply
⬛	Not Used
🟦	Stayed the same
🟪	Permissions is not used
1	Location based Ads
2	Interactive
3	Update UI
4	Background but user observable
5	Use not observable by user

Figure 7: Categorization of each permission usage