A Usage-Pattern Perspective for Privacy Ranking of Android Apps

Xiaolei Li[1], Xinshu Dong[2], Zhenkai Liang[1]

[1]National University of Singapore, Singapore
{xiaolei, liangzk}@comp.nus.edu.sg
[2]Advanced Digital Sciences Center, Singapore
xinshu.dong@adsc.com.sg

Abstract. Android applies a permission-based model to regulate applications (apps). When users grant apps permissions to access their sensitive data, they cannot control how the apps utilize the data. Existing taint-based techniques only detect the presence of exfiltration flow for the sensitive data, but cannot detect how much sensitive data are leaked. Users need more intuitive measures to inform them which apps are going to leak more of their private information. In this paper, we take an alternative approach for identifying apps' internal logic about how they utilize the sensitive data. We define such logic as a sequence of operations on the sensitive data, named as the *data usage pattern*. We build a static analysis tool to automatically extract data usage patterns from Android apps. Our evaluation shows that our approach effectively and efficiently identifies the key operations and thus ranks Android apps according to different usage patterns.

Keywords: Android, Privacy, Static analysis, Information flow analysis

1 Introduction

The Android system relies on a permission-based model to protect sensitive resources on mobile devices. However, the existing permission-based model relies heavily on users' perception of the permissions. A recent study shows that the Android permissions are insufficient for users to make correct security decisions [6]. Users have little idea about how an application (app) would use the granted permissions. For example, to use the advertised features of an app, users may simply grant the dangerous permission to access their locations. In fact, the app may directly leak the location information to an external third-party domain, or carelessly open new interfaces for other apps to escalate their privileges to access it [3]. Although several existing mechanisms have been proposed to analyze the permission usage in Android apps by detecting what and where permissions are used, they do not provide comprehensive information for users to understand how one app utilizes sensitive data after being granted permission to access. Instead, we need a solution which is both technically comprehensive and sufficiently intuitive to end users. Such a solution should help users to make wise choices to protect their privacy when they are installing new apps.

One well-explored direction in understanding the permission usage is to apply data flow analysis on Android apps [2, 4, 7, 8, 13, 16, 17]. However, most of them only determine whether a flow to leak sensitive resources exists or not, but lack precise description

regarding the internal data processing logic, i.e., whether the data usage leaks a lot of information or only a little. Thus, they are unable to inform users of the difference between an app that sends the raw user location to third parties, and another app that only provides a yes/no answer to whether the user is presently at a certain museum or not. Therefore, a desirable approach should deliver more insight to users regarding how their sensitive data are processed and to what extent they are leaked to other parties.

Quantitative information flow (QIF) is an emerging technique for quantifying the information leakage. Various information-theoretic metrics have been proposed, such as through one particular execution path [14] or publicly observable states [9]. Ideally, QIF could be a suitable tool to evaluate how apps use sensitive resources and how much of such information is leaked. Unfortunately, the performance and scalability of existing QIF algorithms and tools are rather limited in practice. In addition, the Android's event-driven paradigm heavily involves asynchronous system callbacks and user interaction, which makes it even more difficult to apply existing QIF mechanisms. Considering the huge number of Android apps and their frequent updates, we need a more efficient and scalable approach.

Our Approach. In this paper, we propose a lightweight and efficient approach to ranking apps based on how they use sensitive resources. In particular, we take the location data of the mobile device as a starting point. Meanwhile, the technique is also applicable to other data types, such as the device ID and the phone number. The idea is to summarize the sequence of key operations on the location data into a *data usage pattern*, which represents the app's internal logic of the location data usage. By comparing the usage patterns for different apps, we group apps with similar functionality and rank them according to their potential leakage of the location information.

Compared to existing data flow analysis techniques that only detect the presence of sensitive data flows, we focus on identifying the important operations on the sensitive data in such flows, which reflect to what extent the data are leaked. Specifically, we propose PatternRanker, which statically analyzes how an app utilizes the location data by analyzing its Dalvik bytecode, and extracts a general and comprehensive pattern representing the location data usage by identifying key operations on the location data. We collect all the possible operations by leveraging static program slicing and taint-based techniques, and then generate the data usage patterns through pre-defined heuristics. The applicability of the data usage pattern is not limited to app ranking. It can also efficiently assist further analysis, such as accelerating existing QIF solutions by applying their current mechanisms on our extracted patterns instead of on the raw logic of apps. We evaluate PatternRanker on 100 top location-related apps, and our experiments show that PatternRanker effectively extracts the data usage pattern for ranking apps. PatternRanker also achieves an average analysis time of 27s per app, which is sufficiently small for analyzing real-world apps.

To sum up, our work has the following contributions:

- We propose a lightweight and scalable approach to ranking apps' threats to user privacy based on the usage pattern of sensitive information.
- We build a static tool to automatically analyze how Android apps utilize the sensitive data and identify the key operations.

– We evaluate a set of 100 top location-related Android apps, and demonstrate the effectiveness of our approach in ranking these apps and classifying them into different categories according to different data usage patterns.

2 PatternRanker Design

Key Design Decisions. We rank one app based on data operation analysis, instead of its leaking bits. For example, Android provides standard APIs *distanceBetween/distanceTo* for apps to calculate the distance between two points. However, some apps implement their own methods to complete the same task through complex mathematical computation (including *toRadians*, *sin* and *cos*). It is extremely difficult to measure which set of math operations may leak more bits of the raw data. It is also improper to conclude that one is safer for leaking fewer bits of the raw data in one particular run, because they are semantically equal even though they may get slightly different results at runtime. Therefore, considering practicality for analyzing real-world apps, we aim to rank apps through identifying the data usage patterns, more specifically, a sequence of key operations on the sensitive data along one flow reflecting the semantic effectiveness whether they preserve the raw data or not, instead of finding a metric of calculating number of bits that one app may leak.

Therefore, we aim to define a pattern to represent how an app operates on the location data, including not only a present flow from pre-defined sources to sinks, but also the key operations in the flow. The data usage pattern indicates two aspects: through which channel and to what degree the sensitive data are leaked. We use the pattern as our ranking metric. For two different usage patterns, we assign a higher rank to the one that leaks less information in the flow, and a lower rank to the one that has only simple data propagation from a source to a sink. We also consider various sink channels for ranking. For example, it gains a higher rank to share the sensitive data with a trusted service than an uncertain domain.

Assumption. The Android system provides well-defined Java interfaces for Android apps to access resources. It also supports NDK that allows developers to design their apps as native code. However, the native code is usually designed for performance improvement in CPU-intensive scenarios like game engines and physics simulation, instead of Android-specific resource access. Hence, the native code is out of the scope in our analysis. In this section, we detail our design of the data usage pattern and a static approach to automatically extracting it.

2.1 Pattern Definition

We focus on analyzing the types of operations on sensitive data in a data flow. To represent how close the output of one operation is to the raw sensitive data, we attach an attribute *Capacity* to the sensitive data during their propagation. Higher value means the output is closer to the original sensitive data. Thus we define the *Pattern* as a sequence of key bytecode operations, which aims to expressively identify the changes of the capacity in one data flow. The sensitive data enter at the source point with the

maximum capacity. During the data flow, an operation may reduce the output's capacity. We also aim to use the pattern to indicate the influence of the sensitive data on the control flow, i.e., whether a code branch is conditionally triggered by the sensitive data. Thus we classify the operations into five categories: **Source**, **Sink**, **Branch**, **Capacity-preserving** and **Capacity-reducing**. Next we explain them in detail.

Source/Sink/Branch. The existing work Susi [15] has given a concrete definition for Android sources and sinks. For sources, we only consider the sources related to the location permission. Additionally, the Android system supports callbacks (e.g., *onLocationChanged*) to pass sensitive data (e.g., GPS). We also consider these sensitive callbacks as sources. In addition to standard sinks, such as network APIs, we treat system state-related APIs (e.g., *setRingerMode*) and IPC channels (e.g., *startActivity*) as sinks. To avoid duplicate analysis on known trusted services and advertising libraries, we also treat these interfaces as sinks. Branch operations refer to the bytecode operations which are essential for exploring execution paths, such as `if-*` and `goto`.

Capacity-preserving/reducing. Different operations on the sensitive data may generate outputs with different capacities. According to the capacity of the output, we classify the operations into capacity-preserving (the output has the same capacity as the input) and capacity-reducing (the output has lower capacity than the input). When summarizing the operation sequence for one pattern, we ignore capacity-preserving operations because the sensitive data have only direct flow without any change of the capacity. Our goal is to identify those key operations that reduce the capacity of one flow. Next, we illustrate our idea through a simple snippet of sequential operations below.

```
1  invoke-virtual {v0, v1}, Landroid/location/LocationManager;-> getLastKnownLocation
       (Ljava/lang/String;)Landroid/location/Location;
2  move-result-object v2
3  ...
4  invoke-virtual {v2, v3}, Landroid/location/Location;-> distanceTo(
       Landroid/location/Location;)F
5  move-result v4
6  move v4, v5
7  cmpg-float v5, v5, v6
8  if-gtz v5, :cond_1
9  :cond_1
10 ...
11 invoke-virtual {v7, v8}, Landroid/media/AudioManager;->setRingerMode(I)V
```

In the above code, we can easily identify its source and sink as Line 1 and Line 11. Line 1 accesses the current location and moves it to `v2` (Line 2). Line 4 calculates the distance between the current location with another point, marked as capacity-reducing operation (CRO). The result is moved to `v4` and then to `v5`. Then Line 7 compares the distance with one value, and sets the comparison result in `v5`. Line 8 uses the comparison result as a condition to trigger a code branch that contains the sink API. The pattern for this code snippet is shown as follows.

```
1  E(SOURCE): invoke-virtual, Landroid/location/LocationManager;->
       getLastKnownLocation(Ljava/lang/String;)Landroid/location/Location;
2  E(CRO): invoke-virtual, Landroid/location/Location;-> distanceTo(
       Landroid/location/Location;)F
3  E(CRO): cmpg-float
4  E(BRANCH): if-gtz
5  E(SINK): invoke-virtual, Landroid/media/AudioManager;->setRingerMode(I)V
```

It is challenging to precisely distinguish whether an operation is capacity-preserving or capacity-reducing because it varies in different contexts due to two main scenarios: *Uncertain Operand* and *Uncertain Method*. Whether one opcode preserves the capacity depends on its operands. For example, `add-int vx,vy,vz` calculates `vy+vz` and puts the result into `vx`. Supposing `vy` is the raw sensitive data, it depends on `vz` whether the result `vx` maintains the same amount of sensitive information. The operand may come from an external source, such as *SharedPreferences*, external storage, network and Android-specific IPC channels, which is difficult for static analysis to determine. Fortunately, most above uncertain scenarios are rare to happen among the popular apps that we have studied. Thus we treat them as constant values. Similarly, if it invokes an external method that is out of our analysis code base, we are uncertain about what kinds of operations will be possibly performed on the sensitive data. In this case, we treat the external method invocation as capacity-preserving (the worse case) by default. To reduce the overestimation, we semantically model frequently used libraries, such as String, Math and parts of Android APIs.

2.2 Ranking Metric

Our ranking is based on two factors: 1) through which channel the data are leaked; 2) to what degree the data are leaked. Note that one app may contain multiple patterns. Here we demonstrate the metric for ranking one pattern. We use the lowest one to represent the rank for the whole app. According to the various sinks, we classify patterns into two main categories: *In-App Usage* with a higher rank and *Sharing* with a lower rank.

For the category of in-app usage, we further classify into two subcategories: capacity-reducing pattern with a higher rank and capacity-preserving pattern with a lower rank. In a capacity-preserving pattern, the sink in the flow outputs the same amount of information as the raw sensitive data, while a capacity-reducing pattern only infers less amount of information. However, it is difficult to justify two capacity-reducing patterns, for it is improper to claim that two capacity-reducing operations (e.g., *subString*) leak less information than one. In future work, we will consider more metrics as references to compare two capacity-reducing patterns, such as how many bits of information are leaked generally in multiple runs, by applying symbolic execution-based mechanisms [11, 12] on our extracted patterns to efficiently simulate multiple runs of the program and evaluate the impact of these extracted key operations on the sensitive data. For now, as a first step, we only target on identifying these key operations from large-scale real-world Android apps, and thus give a rough classification while leaving further analysis as future work.

For the category of sharing, considering the scenario that it is more acceptable for users to share even the raw location data to trusted services, such as Google Map service, than to share one bit of information with untrusted domains, we further group them into three subcategories according to various sharing domains, from high rank to low rank which are *Known Trusted Services*, *Advertising Libraries* and *Uncertain Parties*. We use a whitelist to maintain the known trusted services and advertising libraries. We give the lowest rank to those apps that transfer the location data to uncertain third party domains through network APIs, WebView APIs, SMS APIs and IPC channels. Usually, apps use dedicated libraries for common services and advertising (e.g., Google

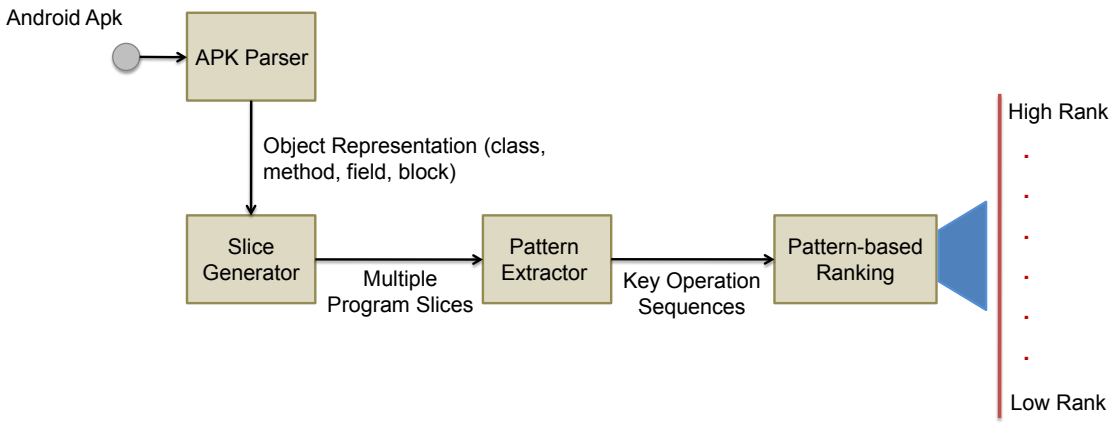

Fig. 1. The Architecture of PatternRanker

map), instead of re-implementing their own through raw Android interfaces (e.g., network APIs). The uncertain channels are mostly used to share content and resources with apps' own third party servers or can only be determined at runtime. Therefore, even though more information (e.g., the recipient's network address, phone number and package name) via these uncertain channels can be mined by applying backward analysis or dynamic instrumentation, they still fall into the uncertain subcategory. Instead, we simply categorize the sharing domains through a whitelist-based filter for common trusted services and advertising libraries collected from large-scale real-world apps.

2.3 PatternRanker Architecture

Figure 1 illustrates the overall architecture. *APK Parser* parses the Android apk into appropriate object representations, such as Smali classes, methods and fields. *Slice Generator* uses slicing technique to generate all the program slices that start from accessing the location data. *Pattern Extractor* extracts the pattern by identifying the key operations in each slice. We design a *Pattern-based Ranking* to rank the apps based on various patterns. Next we explain each component in detail.

Apk Parser. We design static analysis directly on Android disassembled Smali code, which overcomes limitations of the Dalvik-to-Java bytecode transformation [5]. The apk is parsed into Smali files and represented as multiple Smali classes. Each Smali class object is represented as a set of methods and fields. Each method can be further decomposed into several sequential blocks according to its internal branches. Therefore, inside one block, the instructions are sequential without any control flow. The block is treated as a minimum unit for our further analysis.

Slice Generator. We first identify the source points in the app and then perform bottom-to-top analysis. To explore all possible flow paths, we consider field-sensitive flow and Android-specific callbacks into the API hierarchy. To support field-sensitive flow analysis, if the sensitive data is put into one field of a Java object in one method, say *M*,

we also mark all the methods reading that field as top methods of *M*. Specially, the Android's event-driven paradigm supports asynchronous invocation, such as *Handler*, *Thread* and *AsyncTask*. We also bridge the data flow for these scenarios. However, we do not preserve the data flow if it flows outside the app, such as file system, network and IPC channels. From bottom to top, we analyze each method in the call chain. Intuitively, we start tracking the sensitive data from the source point and propagate the taint tags to its top methods.

Now we explain how we analyze inside one method. The method is composed of multiple blocks. We start tainting the sensitive data from the first block with the per-register and per-field granularity. At the beginning of our analysis for each block, we allocate a set of tracked registers and a set of tracked fields. Inside one block, the execution is sequential and the taint tags are propagated according to pre-defined simple propagation rules for each bytecode instruction (e.g., `move` instruction and common math operations). During the tainting, we dynamically update the tracked register set and field set. Specially, if one method invocation involves any tainted input, we dive into the callee method to figure out its internal logic. Note that to support field-sensitive analysis, the input/output of one method include not only the parameters and the return value, but also all the object fields that may be accessed inside it. For efficiency, we do not dive into any method in the publicly known libraries, such as Android SDK, advertising/analytics libraries and other known third-party services. After finishing one block, we go further to its next blocks. The tracked register set and field set are used as the initial sets for its next blocks. To record the control flow, if a branch operation involves a tainted operand, then all the sinks in its branch are treated to be related with sensitive data leakage. However, it may cause overtainting problem to simply taint all the next blocks if the branch condition is tainted. Thus to mitigate this problem, we only maintain the control flow relationship among blocks when the next block of one branch operation has only one reachable path determined by this branch condition.

Pattern Extractor. After we get the program slices, we post-process them to extract patterns. Each slice is treated as sequential operations. We identify whether an opcode is capacity-preserving or capacity-reducing through pre-defined heuristics. Specially, we treat all external methods as capacity-preserving. To reduce false positive, we semantically mark certain Math/String APIs and location-related Android APIs as capacity-reducing, such as *distanceTo/distanceBetween*. For common math operations, `cmp-*`, `cmpl-*`, `cmpg-*`, `rem-*`, `and-*`, `or-*`, `neg-*` are capacity-reducing operations, while `add-*`, `mul-*`, `div-*`, `rsub-*`, `sub-*`, `shl-*`, `*-to-*` are capacity-preserving operations. It is a capacity-reducing operation if it satisfies the following property: even if the sensitive operand is operated with a constant, the result value is still generally unable to recover the sensitive value, such as `and-int`. Specially, we treat `*-to-*` as capacity-preserving, which indicates the data conversion, such as `double-to-int`. Although the conversion may lose the accuracy of the raw data, it semantically behaves like a `move` operation.

Pattern-based Ranking. As described in Section 2.2, our ranking system is based on two factors: through which channel and to what degree the data are leaked inside one pattern. Thus, we classify the extracted patterns by checking their sinks (indicating the leaking channel and the possible receiver) and their capacities (indicating whether

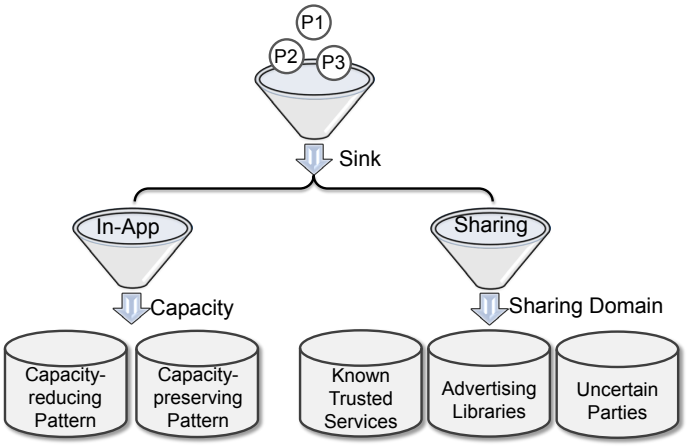

Fig. 2. Pattern-based Ranking Schema

one pattern contains any capacity-reducing operation). As shown in Figure 2, we first classify the apps into two main categories: in-app usage and sharing, by checking the sinks of the patterns. For the category of sharing, we further classify them into three subcategories based on different sharing domains by grouping various sinks. For the category of in-app usage, we group them into two subcategories: capacity-reducing pattern and capacity-preserving pattern, by checking whether the capacity of sensitive data at the sink point is smaller than that at the source point.

3 Evaluation

We implement a standalone tool via Java, which directly works on disassembled Smali code. It leverages the existing tool SAAF [10] to disassemble Android apks and parse the Smali files into appropriate object representations, such as blocks and fields. We implement the slice generator and pattern extractor. We collected 100 top location-related Android apps from the official Android market (i.e., Google Play) as our sample set, and ran our PatternRanker prototype in a Debian system on a server of Intel Xeon E5-2640@2.50GHz with 64G memory. Next, we show our evaluation results in detail.

3.1 App Analysis on Location Usage

Specifically, we found that 28 apps in our sample set have capacity-reducing patterns for in-app usage. Sharing is an appealing feature on the mobile platform, especially as the social networking becomes popular. From our observation, a common usage for location data is to share the raw location data with trusted services and advertising libraries. According to our ranking design, we list them in the following categories from high rank to low rank, shown as Figure 3. One app may include multiple patterns. Here the statistics for each category shows the number of apps having that pattern.

1) In-App Usage: Capacity-reducing Pattern. We identified that 28 apps have capacity-reducing patterns. 15 of them do the distance calculation operation, among which 10 use the default *distanceTo/distanceBetween* Android APIs and the rest 5 implement distance calculation by themselves using Math libraries. Next we take two examples to demonstrate the extracted patterns from them.

Auto Profile Switcher app, with 1,000 - 5,000 downloads, allows users to configure several profiles, such as *home* and *work*. It automatically switches the profile based on the current location. The extracted pattern for it is shown below. They implement the distance calculation through Math library. The pattern indicates that the app makes a complex mathematical computation on the location data and compares the result with a special value via `cmpg-float`. Through manual analysis, we observed that this special value is read from local storage implemented as *SharedPreferences*. This information can be obtained automatically by applying backward dependency analysis on key operands, which we consider as future work. In the end, the comparison result determines the invocation of a sink API *setRingerMode*.

```
1 E(SOURCE): invoke-virtual, Landroid/location/LocationManager;->
      getLastKnownLocation(Ljava/lang/String;)Landroid/location/Location;
2 E(CRO): invoke-virtual, Landroid/location/Location;->getLatitude()D
3 E(CRO): invoke-virtual, Landroid/location/Location;->getLongitude()D
4 E(CRO): invoke-static/range, Ljava/lang/Math;->toRadians(D)D
5 E(CRO): ...
6 E(CRO): invoke-static/range, Ljava/lang/Math;->atan2(DD)D
7 E(CRO): cmpg-float
8 E(BRANCH): if-gtz
9 E(SINK): invoke-virtual, Landroid/media/AudioManager;->setRingerMode(I)V
```

Another example shows the internal logic of how the GPS data affect the display. *GPS Speedometer* is a functional speedometer app with 50,000 - 100,000 downloads. It displays the user's location and travel history. One capacity-reducing pattern extracted from this app is shown below. The sink is a display API *setText*. However, the pattern shows that the GPS data go through multiple comparisons and branch opcodes to finally decide the string to be displayed. Through manual analysis, we observed that the app essentially compares the current bearing with a set of constant values and then decides to display a string from `N,NE,E,ES,S,SW,W,NW`.

```
1 E(SOURCE): Lcom/ape/apps/speedometer/SpeedometerMain;-> onLocationChanged(
      Landroid/location/Location;)
2 E(CRO): invoke-virtual, Landroid/location/Location;->getBearing()F
3 E(CRO): cmpl-double
4 E(BRANCH): if-gez
5 E(CRO): cmpg-double
6 E(CRO): ...
7 E(BRANCH): if-gez
8 E(SINK): invoke-virtual, Landroid/widget/TextView;-> setText(
      Ljava/lang/CharSequence;)V
```

2) In-App Usage: Capacity-preserving Pattern. Information display is one important feature for apps to rich their functionality and convenient users. 25 apps displayed GPS-related information, such as position and signal strength of satellites, accuracy, speed, acceleration and altitude. We observed the following UI-related sinks: *TextView(19), Canvas(7), Toast(3), RemoteViews(2), EditText(1), Notification(1)*.

30 apps logged the GPS data locally. 13 of them directly sent the data to the `LogCat`, which is a public channel for all the installed apps with the `READ_LOGS` permission.

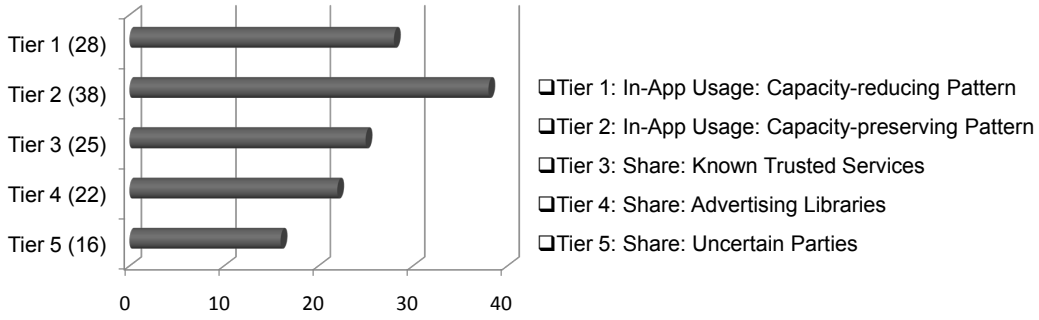

Fig. 3. Ranking Results of Extracted Patterns

However, through manual analysis on the path condition, we found most of them only logged the GPS data in debug mode. This can be verified through the dynamic instrumentation framework [18]. We also observed other logging channels: *database(10), Bundle(4), Java/IO(8)* and *SharedPreferences(6)*.

3) Share: Known Trusted Services. From our statistic in this category, 25 apps used Google map service while the rest 9 used other third-party services.

4) Share: Advertising Libraries. In-app advertising has become an important revenue-generating model for mobile apps. Many of them requested the GPS location for providing targeted advertisement. We observed 22 apps include advertising/analytics libraries potentially accessing the location data from 18 different advertising providers. Commonly one app includes multiple advertising libraries to increase its ads revenue.

5) Share: Uncertain Parties. We also observed that 8 apps share the GPS via IPC channels. For the rest uncertain scenarios, we observed the following sinks: *org/apache/*/HttpPost (5)*, *java/net/DatagramSocket (1)*, *WebView→loadDataWithBaseURL (1)* and *sendTextMessage (1)*. We manually analyzed the WebView and SMS scenarios. In the app *GPS QIBLA LOCATOR*, it uses the WebView API to load an iframe with a URL composed of a fixed third party domain and the geolocation as the parameters. *Mobile Chase-GPS Tracker* registers an *onLocationChanged* listener, inside which it composes an SMS message using the location information and sends it to a phone number stored in the *SharedPreferences*.

3.2 Analysis Time

Figure 4 shows the distribution of analysis time for all the apps. Note that the analysis time excludes the apk parsing that can be pre-processed. The average of analysis time is about 27s per app. 35% of apps finished within 1 sec, due to the simple flow of the location data in them. Within one minute, we achieved 87% coverage. Comparing with other analysis tools [1, 19] at the scale of minutes or larger for real-world apps, the average analysis time of our approach is sufficiently small for ranking a large number of Android apps.

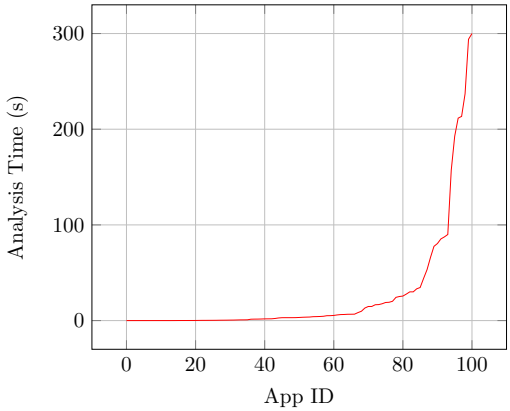

Fig. 4. Analysis Time Distribution

4 Conclusion

To provide users with more intuitive measures to understand how apps treat their privacy, we build a tool PatternRanker to automatically extract the data usage pattern to express it. Comparing to existing taint-based techniques that focus on detecting the presence of one flow, our approach effectively identifies the key operations in the flow. Our experiments on the real-world apps demonstrate its effectiveness and efficiency for ranking a large number of apps.

Acknowledgments

We thank anonymous reviewers for their valuable feedback. We thank Prateek Saxena for his comments on an early presentation of this work. This research is partially supported by the research grant R-252-000-519-112 from Ministry of Education, Singapore. Xinshu Dong is supported by the research grant for the Human Sixth Sense Programme at the Advanced Digital Sciences Center from Singapore's Agency for Science, Technology and Research (A*STAR).

References

1. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In: PLDI (2014)
2. Chan, P.P., Hui, L.C., Yiu, S.M.: DroidChecker: Analyzing Android Applications for Capability Leak. In: WISEC (2012)
3. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege Escalation Attacks on Android. In: ISC (2011)

4. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: OSDI (2010)
5. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: USENIX SECURITY (2011)
6. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android Permissions: User Attention, Comprehension, and Behavior. In: SOUPS (2012)
7. Gibler, C., Crussell, J., Erickson, J., Chen, H.: AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In: TRUST (2012)
8. Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic Detection of Capability Leaks in Stock Android Smartphones. In: NDSS (2012)
9. Heusser, J., Malacaria, P.: Quantifying Information Leaks in Software. In: ACSAC (2010)
10. Hoffmann, J., Ussath, M., Holz, T., Spreitzenbarth, M.: Slicing Droids: Program Slicing for Smali Code. In: SAC (2013)
11. Jeon, J., Micinski, K.K., Foster, J.S.: SymDroid: Symbolic Execution for Dalvik Bytecode. Technical Report CS-TR-5022, Univ. of Maryland (2012)
12. Kim, J., Yoon, Y., Yi, K., Shin, J.: ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. In: MOST (2012)
13. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In: CCS (2012)
14. McCamant, S., Ernst, M.D.: Quantitative Information Flow as Network Flow Capacity. In: PLDI (2008)
15. Rasthofer, S., Arzt, S., Bodden, E.: A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In: NDSS (2014)
16. Sbîrlea, D., Burke, M.G., Guarnieri, S., Pistoia, M., Sarkar, V.: Automatic Detection of Inter-application Permission Leaks in Android Applications. Technical Report TR13-02, Rice University (2013)
17. Wu, L., Grace, M., Zhou, Y., Wu, C., Jiang, X.: The Impact of Vendor Customizations on Android Security. In: CCS (2013)
18. Yan, L.K., Yin, H.: DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In: USENIX SECURITY (2012)
19. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., Wang, X.S.: AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In: CCS (2013)