PrivacyStreams: Enabling Transparency in Personal Data Processing for Mobile Apps

YUANCHUN LI, Peking University, China
FANGLIN CHEN, Carnegie Mellon University, USA
TOBY JIA-JUN LI, Carnegie Mellon University, USA
YAO GUO*, Peking University, China
GANG HUANG, Peking University, China
MATTHEW FREDRIKSON, Carnegie Mellon University, USA
YUVRAJ AGARWAL, Carnegie Mellon University, USA
JASON I. HONG, Carnegie Mellon University, USA

Smartphone app developers often access and use privacy-sensitive data to create apps with rich and meaningful interactions. However, it can be challenging for auditors and end-users to know what granularity of data is being used and how, thereby hindering assessment of potential risks. Furthermore, developers lack easy ways of offering transparency to users regarding how personal data is processed, even if their intentions are to make their apps more privacy friendly. To address these challenges, we introduce PrivacyStreams, a functional programming model for accessing and processing personal data as a stream. PrivacyStreams is designed to make it easy for developers to make use of personal data while simultaneously making it easier to analyze how that personal data is processed and what granularity of data is actually used. We present the design and implementation of PrivacyStreams, as well as several user studies and experiments to demonstrate its usability, utility, and support for privacy.

CCS Concepts: • **Security and privacy** → *Software security engineering*; *Usability in security and privacy*; • **Software and its engineering** → *API languages*;

Additional Key Words and Phrases: Personal data; mobile apps; transparency; functional programming; data granularity

ACM Reference Format:
Yuanchun Li, Fanglin Chen, Toby Jia-Jun Li, Yao Guo, Gang Huang, Matthew Fredrikson, Yuvraj Agarwal, and Jason I. Hong. 2017. PrivacyStreams: Enabling Transparency in Personal Data Processing for Mobile Apps. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 3, Article 76 (September 2017), 26 pages.
https://doi.org/10.1145/3130941

*Corresponding author.

Authors' affiliations and addresses: Y. Li, Y. Guo and G. Huang, Key Laboratory on High-Confidence Software Technologies (Ministry of Education), School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China; F. Chen, T. J.-J. Li, M. Fredrikson, Y. Agarwal, and J. I. Hong, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA..

2474-9567/2017/9-ART76 $15.00
https://doi.org/10.1145/3130941

1 INTRODUCTION

Developers often access and use a great deal of personal data about users on their smartphones to create apps with rich and meaningful interactions. This personal data is comprised of both sensor data as well as log data, and includes where we go (GPS locations), what we see (photos, videos), who we know (contact list) and talk to (SMS logs, call logs), what we are interested in (search history, browser history), and more.

Apps might use personal data at different granularities, depending on the goals of the app. For example, an app might need location data at the city level just once an hour, or as continuous exact coordinates. Similarly, an app might use the microphone to record full-fidelity raw audio, or simply check the sound level of the surroundings. The granularity of this personal data is one important dimension in influencing perceptions of privacy.

In this paper, we focus on three related challenges for personal data. First, it is challenging for auditors[1] and end-users to understand what granularity of personal data is used in an app. Second, there is no easy-to-use method for well intentioned app developers to expose such fine-grained information. Third, it is difficult for developers to make use of personal data, due to different kinds of APIs and data structures.

Disclosing the granularity of data used in an app without the app developer's help is challenging. Existing systems (such as Android and iOS) provide an all-or-nothing permission mechanism, describing whether a type of raw data can be accessed but not its granularity. It is also hard to analyze the code making use of personal data, since the relevant code can be spread out across multiple classes and methods and have semantics that are hard to analyze. As a result, it is very hard to analyze the granularity of data using state-of-the-art code analysis techniques [3, 22].

Letting app developers actively expose the granularity information is also difficult. Existing methods such as asking developers to provide a privacy policy often require extra efforts from developers and still suffers from inaccuracy [44]. Furthermore, developers might not have a strong incentive to put effort into it, as exposing the granularity information does not always provide direct benefits.

We take a holistic approach to address the above challenges. Our framework, called PrivacyStreams, is intended to help developers become more efficient in making use of personal data, while also making it easier to analyze how the data is processed and what granularity of data is actually used. PrivacyStreams offers a simple and uniform functional programming model for accessing and processing many kinds of personal data available on smartphones. Developers can request, process, and take action on personal data by setting up a single chain of code, thus centralizing data processing steps in a single location. PrivacyStreams also offers a set of commonly used and well-defined operators to make it easier for developers, which also makes it easier for us to analyze the semantics of the code.

For example, a sleep monitor can access microphone loudness with PrivacyStreams on Android as follows (where UQI stands for Uniform Query Interface, our API for requesting personal data):

```
1  UQI.getData(Audio.recordPeriodic(10*1000, 2*60*1000), Purpose.HEALTH("monitoring sleep"))
       // Record a 10-second audio periodically with a 2-minute interval.
2      .setField("loudness", calcLoudness(Audio.AUDIO_DATA))  // calculate loudness.
3      .onChange("loudness", callback)  // callback with loudness value when the loudness changes.
```

The above code will access the raw audio every two minutes for 10 seconds, transform it into loudness using a built-in operator for calculating loudness for audio, and then return the result via a callback. Developers only need to care about the logic and operations over raw audio data instead of the actual technical details on how they are calculated. Note that due to how the chain of code is setup, the developer does not actually ever see the raw audio, but rather just the final processed result. Meanwhile, auditors and end-users can use other tools we

[1]Here we use auditors to represent third-party stakeholders such as app markets, smartphone makers, or government agencies, who are interested in auditing what kinds of sensitive information are accessed by each app and how they are used.

have developed to analyze that this app uses microphone to get loudness every two minutes. The developer can also specify a purpose that can serve as a hint to auditors, end-users, and any generated privacy displays.

The design of PrivacyStreams is inspired by popular functional programming frameworks [14, 47, 57] and uses best practices on API design [7]. We borrow the idea of pipelines from stream processing models. In PrivacyStreams, a pipeline is a sequence of three types of functions, including a *Provider*, a list of *Transformation*s and an *Action*. The *Provider* gets raw data from data sources (such as sensors, databases, etc.) and converts it to a standard-format stream. The *Transformation*s define the operations over the stream, such as `filter`, `map`, `group`, etc. Finally, the *Action* is used to output the stream to the app requesting it, for example, collecting the items to a list, notifying the app with a callback, or calculating a statistic over the data. The *Transformation*s can complete various types of operations by passing different operators as parameters. The operators are crafted based on common personal data use cases in popular apps, including common operators (such as `eq`, `add`, `count`, etc.) and domain-specific operators (such as `calcLoudness` for audios, `inArea` for locations, `getMetadata` for images, etc.). There are also several privacy-related operators (such as `hash` for text, `round` for numbers, and `blur` for photos) for developers to integrate privacy-preserving features into their code.

This kind of stream processing programming model also makes compiled apps easier to analyze. First, the functional programming model centralizes the data processing steps to a single method (even a single line of code), eliminating a lot of external states. Thus, the analyzer only needs to deal with the data flow inside each method body separately, instead of the data flow across multiple methods and threads. Second, as each data processing step is defined by built-in operators that have known semantics, it is possible to understand how one's personal data is processed step by step. We introduce a data structure, PrivacyStream data flow graph (DFG) to describe the personal data processing steps implemented with PrivacyStreams, as well as an algorithm to extract PrivacyStream DFGs from an app.

We have implemented PrivacyStreams for Android. The implementation includes a library and a static analyzer for determining how personal data is processed by an app and at what granularities. Developers can import the library to their apps to use the PrivacyStreams APIs to access and process personal data. Auditors and app stores can use the static analyzer to extract detailed steps of personal data processing from the apps developed with PrivacyStreams. As a result, they can provide useful and accurate descriptions for end-users to help them better understand the privacy sensitivity of mobile apps.

To evaluate PrivacyStreams, we conducted an in-lab user study and a free-form field study to demonstrate its utility and usability to developers. Our results show that developers can understand how to use PrivacyStreams quickly and complete programming tasks efficiently and correctly with PrivacyStreams. Based on experiments performed on five apps developed with PrivacyStreams in the field study, our code analyzer can extract detailed steps of personal data processing accurately and quickly (less than 15 seconds per app). We also show the feasibility of generating a privacy description from this analysis through a proof-of-concept description generator on the five apps developed with PrivacyStreams.

In summary, we make the following contributions:

- We study around 100 popular apps from the Google Play store as well as research papers related to personal data, summarizing common use cases of personal data. We show a range of how personal data is used at different granularities in different apps, and motivate how many apps do not need full access to the raw data. These findings demonstrate the need for PrivacyStreams and informed our API design.
- We introduce PrivacyStreams, a framework that provides a functional programming model and a set of uniform APIs for accessing and processing personal data in mobile apps. Using PrivacyStreams to access and process personal data can make it easier to analyze detailed steps of data processing while easing app developers' programming efforts.

- We implement PrivacyStreams in Android as a library. The library is open-sourced for other app developers to use. We also provide a static analyzer for extracting the data processing steps in Android apps developed with PrivacyStreams, which can be used to generate useful privacy descriptions for end-users.
- We demonstrate the functionality and usability of PrivacyStreams through user studies and show its support for privacy by running static analysis on apps developed with PrivacyStreams.

2 BACKGROUND AND MOTIVATION

2.1 Personal Data & Apps Using Personal Data

The definition of what constitutes personal data varies based on context. For example, from the perspective of privacy laws, personal data is similar to *Personally Identifiable Information* (PII), which is information that can be used on its own or with other information to identify, contact, or locate a single individual. In mobile health research, the term "small data" is used to refer to the data derived from people's individual digital traces [18]. In the context of this paper, "personal data" refers to any programmatically-available data that describes a person, such as the person's behavior (app events, search history, etc.), content (files, photos, etc.) and context (location, device status, etc.).

On smartphones, many types of personal data items can be accessed using standard APIs. For example, Android apps are able to access users' call logs, SMS messages, photos, and locations using the Android SDK. Similar APIs exist on iOS. The ability to access personal data enables intelligent features in apps, such as location-based services and context-based behaviors. Researchers have proposed even more advanced use cases and applications, such as using smartphone data to infer social relationships [36, 50] and analyzing large-scale personal data to model the crowd [10, 42].

The popularity of personal data-based apps also leads to significant privacy concerns. For example, many malicious Android apps are stealing users' private information from smartphones [58], and many seemingly innocuous apps have been shown to frequently access personal data often unknown to the user [1, 54, 56]. Personal data leakage and misuse may harm users financially, socially, or even physically [58].

2.2 Personal Data Access Control

Popular mobile platforms such as iOS and Android adopt a permission-based access control mechanism to protect personal data. If an app wants to access a certain type of personal data, it must request a permission, while the end user may decide whether to grant the permission. Typically, once the user grants the permission, that app gets complete access to the raw data. For example, if an Android app is given the LOCATION permission, it can access users' location coordinates with the `getLastKnownLocation` method or `requestLocationUpdates` method. If an app obtains the MICROPHONE permission, it can record audio from microphone using `MediaRecorder` APIs.

In order to make informed decisions, it is helpful for users to understand what granularity of data is being requested by an app. For example, a user will likely be less concerned if a sleep monitoring app accesses microphone loudness, as compared to the app accessing full-fidelity raw audio data. Similarly, users are likely to be less concerned about granting their city-level location to a weather app, instead of their exact location, which might lead to their detailed movements being tracked. In other words, the *granularity* and the *purpose* of private data accesses are important dimensions in determining how comfortable users are with the app behavior, as well as influencing the decision to install the app in the first place.

3 PERSONAL DATA GRANULARITY IN REAL APPS

We examined several real apps to understand common use cases in popular apps, as well as to motivate the potential benefit of a framework like PrivacyStreams. The apps we investigated include 99 popular apps from

Table 1. The granularities of personal data needed in popular market apps and research papers. Each column represents a type of granularities over personal data. We considered two types of granularity (**how often/much** and **what form**) and two levels of granularities (**fine** and **coarse**) for each type. The "#apps" column is the number of market apps belonging to that granularity level, and the "research apps" column is the references to the research papers belonging to that level. As can be observed, a lot of apps do not need the finest granularities of data. For each level of data, we highlight some typical usage patterns and operations. Note that the granularity levels were determined based on what the apps need for their features, instead of what the apps actually access. In summary, most apps do not need the fine-grained raw data, and there are several common usage patterns and operations to get the coarse-grained data.

Data type	**How often/much**			**What form**		
(#apps)	granularity	#apps	research apps	granularity	#apps	research apps
Location (68/99)	**fine** - continuous tracking **coarse** - occasional access - user-driven access	3 65	[6, 15, 31, 38, 42, 45, 52, 53] [10, 25]	**fine** - precise coordinates **coarse** - coarse location - speed, bearing, etc. - distance to a place - semantic location	3 65	[15, 25, 45, 52, 53] [6, 10, 31, 38, 42]
Microphone (23/99)	**fine** - continuous recording **coarse** - periodic sampling - user-driven recording	0 23	[52, 53] [9, 10, 15, 35]	**fine** - raw audio **coarse** - loudness - whether is talking - amplitude samples - text from speech	21 2	 [9, 10, 15, 35, 42, 52, 53]
Contacts (54/99)	**fine** - all contacts **coarse** - recent called contacts - contacts with emails	19 35	[4, 36] [31]	**fine** - contact details **coarse** - only phones / emails - hashed phone number - elided phone number	20 34	[4, 31, 36]
Messages (17/99)	**fine** - all messages **coarse** - recent messages - sent messages - incoming messages - from a certain address	9 8	[33, 36] [4, 25, 27, 31, 38]	**fine** - message details **coarse** - only the contact info - obfuscated messages - message length - message count	4 13	[25] [4, 27, 31, 33, 36, 38]
File (89/99)	**fine** - all files **coarse** - only images - files under a dir	4 85	[21] [45, 49]	**fine** - raw file content **coarse** - file type/size - image metadata - blurred image	85 4	[21] [45, 49]
Overall #cases: 251	**fine** **coarse**	13.9% 86.1%		**fine** **coarse**	53.0% 47.0%	

Google Play (we chose the top 3 free apps in each of the 33 categories) and 20 apps described in recent research papers (the papers are selected primarily from conferences such as UbiComp and CHI).

We considered five types of personal data and two kinds of granularities in our analysis, as shown in Table 1. For each app, we summarize its main features by testing the app and/or reading the descriptions (papers) in order to decide what granularity of data is needed by the app. Note that we are interested in understanding the range of data granularities that a framework needs to support rather than a detailed analysis of the behavior of apps.

Our goal is not to give a complete or detailed taxonomy of data granularities. Instead, we summarize two typical kinds of granularities: how often/much and in what form. The first type of granularity is based on the volume of data, i.e. how often the data is accessed or how much data the app needs. The intuition is that higher quantities of data may lead to higher privacy concerns. For example, an app reading all text messages is much more sensitive than one that only reads messages from one specific address. The second type of granularity is based on the form or level of data needed. In many cases, abstractions can be less sensitive than the raw data. For example, it is likely that people will be more willing to share their "city" location rather than their "exact" location. We also summarized several common usage patterns for different granularity levels. For example, microphone loudness is a commonly-used abstraction, as is filtering files by specific file types. These usage patterns guided the design of our framework.

Based on the study of personal data use cases in real apps, we make the following key observations:

- Apps' privacy sensitivity can be differentiated based on the data granularity, such as how much, how often, what form of personal data is accessed;
- Most apps do not need full access to raw data for their core functionality, implying that it is possible to reduce the granularity at which they access private data about the user, thereby reducing privacy concerns;
- There are some common operations used by apps to filter and process personal data, which downgrade the granularities of data being accessed in the apps.

These findings demonstrate the need for a framework that can disclose the granularity of data being accessed in an app, while the common operations and usage patterns suggest the range of operators that the framework should support.

4 PRIVACYSTREAMS OVERVIEW

We propose PrivacyStreams, a programming framework to make it easier for app developers to access and process personal data, while automatically exposing the detailed steps of data processing, thus enabling greater transparency to users about an app's behaviors. In doing so, PrivacyStreams aims to address key limitations of current mobile platform APIs which provide little transparency into what granularities of data is actually used in an app. The key idea of PrivacyStreams is to *centralize the steps of user data access and processing with a uniform functional programming model and offer a set of built-in operators to customize each step.* The centralized steps can make it easier for code analysis, especially since each step is based on built-in operators with relatively simple semantics. Meanwhile, the uniform APIs and rich set of operators can ease developers' programming efforts. Our goal is to support well intentioned app developers, who do not hide app behaviors purposefully and who also want to streamline their development processes.

We illustrate the workflow of PrivacyStreams with a running example (as described in Section 4.1) throughout this section. Section 4.2 describes the architecture of the programming framework, and Section 4.3 talks about how to analyze data processing steps from an app developed with PrivacyStreams.

4.1 A Running Example

Suppose we want to develop an app that uses call log data to infer a user's relationships with others [36, 50]. The app's personal data-related behavior is described as follows:

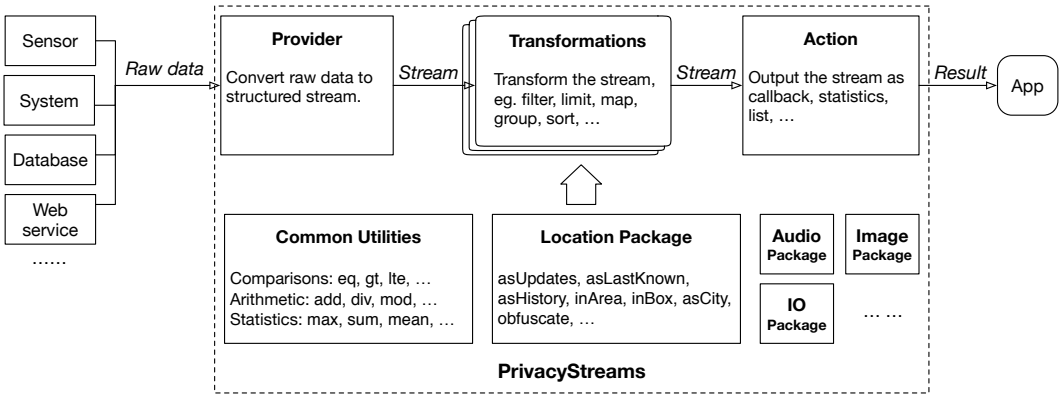

Fig. 1. Overview of PrivacyStreams framework. PrivacyStreams acts as a bridge between the raw data and the result delivered to the app. There are three types of basic functions in PrivacyStreams: *Provider* converts the raw data to a structured stream, *Transformation* transforms the streams based on many built-in operators, and *Action* outputs the stream to the app.

> **Task:** Get the user's call logs during the past year. For each contacted phone number, calculate the total number of phone calls and total duration of phone calls with that particular contact.

Implementing this task with the existing Android standard APIs may involve writing substantial code and use of data structures. Developers need to get raw call logs using a SQL-style API and manually filter and process them. As this task might be resource and time-consuming, developers will typically need to use multiple threads to avoid blocking the user interface. If the app is supposed to run on devices above Android 6.0, they also need to handle runtime permissions. Lots of external states and multiple functions make it hard to analyze with respect to overall behavior, specifically, what granularity of call log data is actually accessed. However, all these tasks can be easily implemented using PrivacyStreams with a few lines of Java code:

```
1  UQI.getData(Call.getLogs(), Purpose.SOCIAL("inferring relationships with others."))
2     .filter(recent(Call.TIMESTAMP, 365*24*60*60*1000L)) // keep the calls in the recent year
3     .groupBy(Call.CONTACT) // group by the contact field
4     .setGroupField("#calls", count())) // count the number of grouped items
5     .setGroupField("Dur.calls", sum(Call.DURATION))) // calculate the sum of durations
6     .setField("hashed_phone", hash(Call.CONTACT)))) // hash the phone number
7     .project("hashed_phone", "#calls", "Dur.calls") // only keep three fields
8     .forEach(callback) // callback for each item, i.e. for each contact
```

Code 1. Using PrivacyStreams to collect the number of calls and total duration of calls for each phone number.

This code will output the number of calls and total duration of calls for each contact with the hashed phone number in a callback, which is the only channel for the data flowing into the app. Meanwhile, each step of data processing can easily be exposed to end-users. We explain the details of each step in the following sections, which will also serve to describe how PrivacyStreams is designed.

4.2 PrivacyStream Pipeline

In PrivacyStreams, the steps of personal data processing are organized as a pipeline. The pipeline model describes the operations taken along the information flow from the raw data (the raw values directly produced by data sources) to the abstract data (the results delivered to the app). While developers set up the pipeline, they can only see the final results rather than the intermediate results.

As shown in Figure 1, a PrivacyStream pipeline is a sequence of three types of functions: a data providing function (i.e. *Provider*), a list of transformation functions (i.e. *Transformation*s) and a data output function (i.e. *Action*). First, the *Provider* gets raw data from the data source and converts it to a stream in a standard format. Then the stream is processed with several *Transformation* functions. Each *Transformation* takes a stream as input and produces another stream as output. Finally, the stream is output by an *Action* function into the result needed by the app. A pipeline describes a single path from the data source to the results, thus it does not support multiple data sources (such as sensor fusion). However, a single-path pipeline can describe the majority of personal data use cases in market apps (as described in Section 3).

The architecture of PrivacyStreams is inspired by popular functional stream processing frameworks such as Spark Streaming [57] and Java 8 Streams [47]. The centralized code structure in the functional programming model is one of the main features we adopted when designing PrivacyStreams. The major difference between PrivacyStreams and existing functional programming frameworks is that PrivacyStreams is tailored for personal data, in terms of data sources and operations.

Different types of data may adopt different APIs and formats, and *Provider*s convert them to a uniform format and provide a uniform API. The uniform data format is a stream of items, and each item is a Map with just one level of key-value pairs. The initial keys (i.e. fields) for each type of data are selected based on common use cases. For example, a location provider (*Geolocation.asUpdates*()) gets the raw location data from GPS and converts it to a stream of location items. Each location item has the same set of fields, including `COORDINATES`, `TIMESTAMP`, `ACCURACY`, etc. A call log provider (*Call.getLogs*()) gets raw call log from the database and converts it to a stream of call log items. Each item contains fields like `TIMESTAMP`, `CONTACT`, `DURATION`, etc. The initial fields for each type of data can be found in our API documentation [39]. With the uniform data format, developers can access and process different types of data using the same set of APIs.

The *Transformation* functions define the operations over the streams. Like other stream processing frameworks [47, 57], PrivacyStreams provides common transformation functions like `filter`, `map`, `group`, etc. For example, the `filter` function checks all the items in a stream and only keeps the ones that fulfill a condition. The `map` function converts each item in a stream to a new item with a one-to-one mapping function. The `group` function groups a set of items together to form a new item. An app can use `filter` to include or exclude the locations in a geofence, use `map` to covert the coordinates to the neighborhood name, or use `group` to cluster the locations every certain period of time. Note that a data processing pipeline may contain multiple *Transformation*s chained together one after the other.

We craft many built-in operators based on common use cases shown in Section 3. The *Transformation*s can behave differently by using different operators as the parameters. For example, the `filter` *Transformation* uses a predicate operator to decide whether to keep an item, while the `map` *Transformation* uses a item-to-item operator to convert each item in the stream. The operators include both common operators and domain-specific operators. Common operators are generic for all types of data, such as logical operators (`and`, `or`, etc.), comparators (`eq`, `gt`, etc.) and arithmetic operators (`add`, `multiply`, etc.). Domain-specific operators are specific for certain types of data (such as location and audio) or certain types of operations (such as I/O and machine learning). Since each built-in operator has easy-to-extract semantics, the pipeline using them can be expressed transparently to end-users.

Finally, an *Action* function is used to output the stream as the result needed by the app. Common *Action*s include collecting the items into a list, calculating a statistic of the data, sending the data over network, notifying the app with callbacks, etc. For example, an app can use the `asList` Action to collect the items as a list. It can also use a `forEach` Action to get callbacks for each item. An *Action* ends the pipeline, and the information flow after an *Action* is no longer under the control of PrivacyStreams.

The data processing pipeline corresponding to Code 1 is illustrated in Figure 2. Line 1 in the code gets a stream of call log items from *Call.getLogs*() (the *Provider*). Line 2 to line 7 transform the stream with some *Transformation*s,

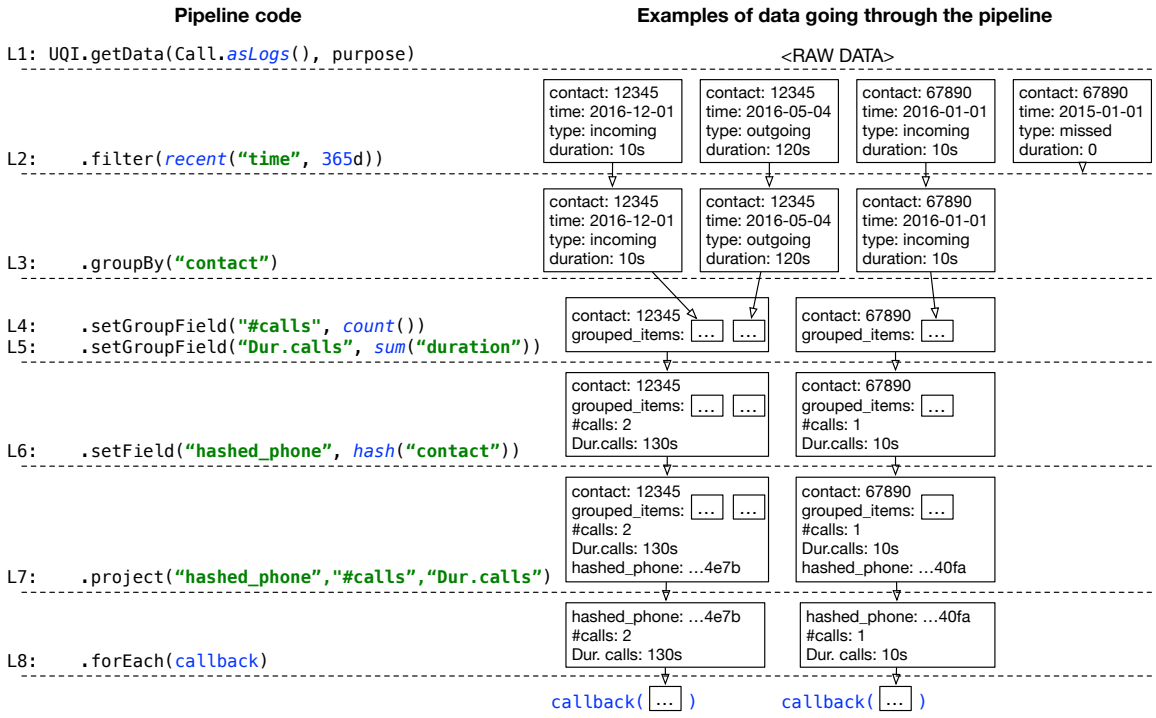

Fig. 2. A demonstration of the personal data processing pipeline described in the running example. The actual example code is shown on the left (Code 1), while the data being processed through the pipeline is illustrated on the right. Note, we have simplified some field names and used dummy values for contacts.

including using a *filter* function to keep the logs in recent 365 days (L2), grouping the logs with same phone number together (L3), setting a new field named "#calls" for each group item as the total number of grouped call logs (L4), setting another new field named "Dur.calls" for each group as the total duration of grouped call logs (L5), computing hash of the CONTACT field for each group and put the hashed value to a new field "hashed_phone" (L6), and keeping the hashed phone number, the number of calls and the total duration of calls (L7). Finally, Line 8 output the result with an *Action* - *forEach*(), which passes each item to the app as a callback.

As can be seen from Figure 2, the granularity of the personal data, and consequently its sensitivity from a privacy perspective decreases as it goes through the PrivacyStream pipeline. Initially, the highly sensitive raw data is accessed and is also the most difficult for developers to use. The *Provider* makes the data simpler for processing while the data remains as sensitive as the raw data. *Transformation*s and *Action* reduce the data sensitivity by abstracting and excluding some information from the stream. Finally, the pipeline produces an abstract result, which is less privacy sensitive and easier for the app to use than the raw data.

Describing the granularity at which an app accesses data becomes easier as the number of operations that are implemented using PrivacyStreams is increased. While not every app will be able to implement their entire pipeline using PrivacyStreams, they can still achieve transparency to a certain extent by implementing part of the pipeline with PrivacyStreams.

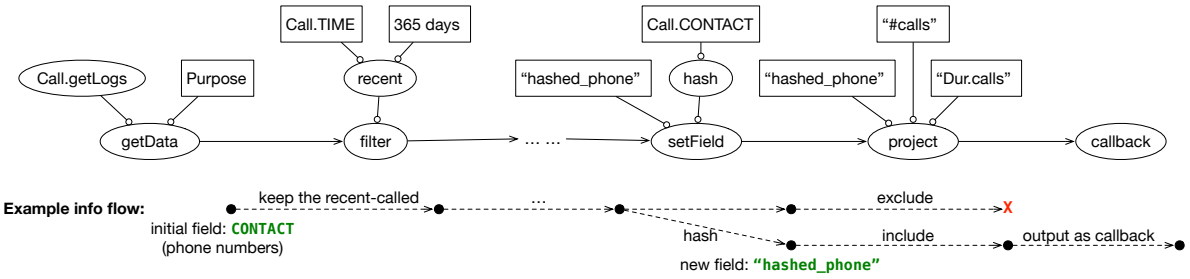

Fig. 3. The *PrivacyStream data flow graph* (DFG) corresponding to Code 1 and an example information flow in the DFG. Each node in the DFG on the horizontal path represents a data processing operation. The PrivacyStream DFG is a data structure to help understand how personal data is processed and what granularity of data is produced.

4.3 PrivacyStream Data Flow Graph

As the code written with PrivacyStreams is centralized and based on a common set of APIs, it becomes easier to analyze the detailed steps of data processing in PrivacyStreams compared to using the standard Android APIs. We introduce a data structure called PrivacyStream data flow graph (DFG) to represent the data processing steps:

Definition 1. *A **PrivacyStream DFG** is directed graph $G = \langle N, E \rangle$, which represents the steps of data processing in PrivacyStreams. Each node $n \in N$ is an operation represented as a tree, where the parents are function names and the children are parameters. Each edge $e \in E$ means the information flows between the operation in source node to the operation in target node.*

The purpose of PrivacyStream DFG is to describe how personal data is processed step by step. Each node represents a step, and the step has easy-to-extract semantics. Specifically, the first node describes what data items are accessed and what initial fields each item has (based on the *Provider*'s definition), the intermediate nodes describe how each field is filtered and transformed (based on the *Transformations*' meaning), and the last node describes what fields are passed to the app and how (based on the *Actions*' meaning). Based on a DFG and the semantics of each function in the DFG, we are able to know the information flow between fields and the meaning of each operation along the flow.

For example, Figure 3 shows the PrivacyStream DFG of the running example. Let's consider the information flow of phone numbers in the DFG. The first node indicates that call log is accessed and the phone number is one of the initial fields. The second node (`filter`) means that only the phone numbers contacted in recent 365 days are kept. After several steps, the `setField` node creates a new field "hashed_phone" whose values are hashed from the original phone numbers. The `project` node means only the hashed phone numbers are included while the original phone numbers are excluded. Finally, the `callback` node means the hashed phone numbers are outputted as callbacks. Combining above steps together, we are able to know that the app gets the hash value of the phone numbers contacted in recent 365 days.

Extracting a PrivacyStream DFG from a program using static analysis involves the following basic steps:

(1) Locate the invocation to the `UQI.getData()` function by scanning the code. If found, create a PrivacyStream DFG with a single node N_0. The output of N_0, i.e. the return value of `UQI.getData()`, is a stream instance S_0;
(2) Find where S_0 is used as the base parameter of a function, which should be either a *Transformation* or an *Action*. If found, denote the function as N_1 and create an edge E_1 from N_0 to N_1;
(3) If the function (N_1) found in step 2 is a *Transformation*, then denote the return value (which is a stream instance) as S_1 and repeat step 2 for S_1. Else (the function is an *Action* or the function is not found) it means

the stream is outputted. After this step, the nodes and edges in PrivacyStream DFG (e.g. the horizontal path in Figure 3) are found;

(4) Recursively backtrace the definitions of the parameters of pipeline functions (N_0, N_1, ...). That is, if the parameter is a constant value (e.g. "hashed_phone"), simply add the value to the node tree. If the function parameter is an operator (e.g. `hash()`), add the operator to the tree and continue tracing the parameters of the operator. After this step, the parameter tree for each node is constructed, thus the PrivacyStream DFG is fully extracted.

The algorithm described above is fast and accurate. Because the programming model of PrivacyStreams centralizes the pipeline to a single method, the analyzer only needs to consider the control flow inside a method body (intra-procedural analysis). Meanwhile, PrivacyStreams reduces external states by offering built-in common operators, thus the DFG extracted within a method body can precisely describe the whole data processing steps. However, if a novice or careless developer separates the pipeline into different methods, our analyzer will only extract the steps in the method where `UQI.getData` is located, as analyzing the cross-method pipeline (interprocedural analysis) is time-consuming and less accurate.

With the help of static analysis, PrivacyStream DFGs can be exposed before installing the app, enabling a number of useful scenarios. For example, auditors can assess the privacy risk level of an app based on the statically extracted PrivacyStream DFG and potentially assign them privacy grades [32]. The DFG can also be used by app markets to generate a description for end-users, helping them understand the granularity of data used in the app apriori.

5 PROGRAMMING INTERFACE OF PRIVACYSTREAMS

5.1 Unified Data Format and Unified Query Interface

The basic data types in PrivacyStreams are *Stream* and *Item*. *Stream* is the data structure being produced, transformed and output in a PrivacyStream pipeline. A *Stream* is a sequence of *Item*s while an *Item* represents a personal data item (e.g. location item, audio record, image, etc.). The two types of streams in PrivacyStreams are *MStream* (short for multi-item stream, e.g. location updates, call logs, etc.) and *SStream* (short for single-item stream, e.g. current location, audio record, etc.).

App developers can use a uniform query interface (UQI) to create a PrivacyStream pipeline. The template of how to create a pipeline with UQI is as follows:

$$UQI.getData(Provider, Purpose)[.transform(Transformation)]^*.output(Action)$$

Specifically, `UQI.getData(Provider, Purpose)` declares the data source (*Provider*) and the purpose of the data access. `UQI.getData()` returns a stream, and the stream is transformed with several *Transformation*s and output with an *Action*.

We designed the *Purpose* parameter to enable and directly motivate app developers to specify the purpose of data use. The purpose information can help end-users better understand privacy [32] and is not easy to analyze [37, 51]. Such design would make it much easier to get the purpose information, although it is just a placeholder currently and we haven't done anything with it.

5.2 Pipeline Functions

Pipeline functions, including *Provider*s, *Transformation*s, and *Action*s, directly deal with *Stream*s. Table 2 shows some examples of pipeline functions. For example, `Call.getLogs()` is a *Provider* that produces a *MStream*, `filter()` is a *Transformation* that converts a *MStream* to another *MStream*, and `forEach()` is an *Action* that outputs a *MStream* in callbacks.

Table 2. Examples of pipeline functions in PrivacyStreams. The second column is the input and output types of the function and the third column is the signature (name and parameters) of the function. A function parameter can be an item field name (e.g. $fieldName$, $newFieldName$, etc.), a constant value (e.g. $milliseconds$, $index$, etc.), an operator (e.g. $f_{Item \mapsto Bool}$, $f_{Item \mapsto Item}$, etc.) or a callback (e.g. $Callback_{Item}$, $Callback_{TValue}$). The subscripts of operators and callbacks are the input and output types, where $TValue$ represents the generic type of a field value in an item. Note that this table only contains a subset of functions, and the full list of functions can be found in the API documentation [39].

Function Set	Input ↦ Output	Name (Parameters)
Providers	$Null \mapsto MStream$	Call.getLogs(), Geolocation.asUpdates()
	$Null \mapsto SStream$	Audio.record(), Geolocation.asCurrent()
Transformations	$MStream \mapsto MStream$	filter/limit ($f_{Item \mapsto Bool}$)
		timeout ($milliseconds$)
		sortBy ($fieldName$)
		mapEachItem ($f_{Item \mapsto Item}$)
		groupBy/localGroupBy ($fieldName$)
		ungroup ($listFieldName$, $newFieldName$)
	$MStream \mapsto SStream$	getItemAt ($index$), getFirst ()
		select ($f_{Items \mapsto Item}$)
		reduce ($f_{Item,Item \mapsto Item}$)
	$SStream \mapsto MStream$	ungroup ($listFieldName$, $newFieldName$)
	$SStream \mapsto SStream$	mapItem ($f_{Item \mapsto Item}$)
Actions	$MStream \mapsto Void$	forEach/onChange/ifPresent ($Callback_{Item}$)
		forEach/onChange/ifPresent ($fieldName$, $Callback_{TValue}$)
	$MStream \mapsto List_{Item}$	asList ()
	$MStream \mapsto List_{TValue}$	asList ($fieldName$)
	$SStream \mapsto Void$	ifPresent ($Callback_{Item}$)
		ifPresent ($fieldName$, $Callback_{TValue}$)
	$SStream \mapsto Item$	asItem ()
	$SStream \mapsto TValue$	getField ($fieldName$)

5.3 Built-in Operators

Operators are also functions, but unlike pipeline functions that directly deal with *Stream*s, operators usually deal with *Item*s and field values.

We provide a lot of built-in operators based on common use cases in market apps (as described in Section 3). There are mainly two types of operators, including common operators that deal with general types of data and domain-specific operators that are designed for certain types of data or operations. Specifically, there are some operators tailored for privacy-preserving operations. This section will introduce each group of operators respectively.

5.3.1 Common operators. Table 3 shows some examples of common operators. For example, `eq` operator takes an item as the input and a boolean value as the output, and `eq("x", 1)` returns true if the "x" field of the input item equals 1. By passing `eq("x", 1)` operator to `filter()` (a *Transformation* function), it will keep the items whose "x" field equals 1.

Sometimes nested functions might appear redundant, thus we wrap some common combinations of functions to one function for simplicity. For example, `.filter(eq("x", 1))` can be simplified as `.filter("x", 1)` and `.mapEachItem(setField("y", ...))` can be simplified as `.setField("y", 1)`.

Table 3. Examples of common operators in PrivacyStreams. The operators are normally passed to pipeline functions (as shown in Table 2) as parameters.

Input ↦ Output	**Name (Parameters)**
$Item \mapsto Item$	setField ($newFieldName$, $f_{Item \mapsto TValue}$)
	setGroupField ($newFieldName$, $f_{Items \mapsto TValue}$)
	project ($fieldName1$, $fieldName2$, ...)
$Item \mapsto TValue$	getField ($fieldName$)
$Items \mapsto Item$	getItemWithMax/getItemWithMin ($numFieldName$)
$Items \mapsto Bool$	checkAll/checkExist ($f_{Item \mapsto Bool}$)
$Items \mapsto Num$	count (), max/min/average/range/... ($numFieldName$)
$Items \mapsto String$	longest/shortest/concat/... ($stringFieldName$)
$Items \mapsto List$	union/intersaction/... ($listFieldName$)
$Item \mapsto Bool$	eq/ne/gt/lt/contains/... ($fieldName$, $TValue$)
$Item \mapsto Num$	add/sub/mul/div/mod ($numFieldName1$, $numFieldName2$)
$Item \mapsto String$	subStr ($stringFieldName$, $startIndex$, $length$)
...	...

Table 4. Examples of functions (*Provider*s and operators) in location package (one of the domain-specific packages in PrivacyStreams). Developers can use these functions together with basic functions to access and process location data. Similar to the location package, there are many other domain-specific packages (audio, image, message, sensor, etc.) in PrivacyStreams.

Function Set	**Input ↦ Output**	**Name (Parameters)**
Location Providers	$Null \mapsto SStream$	asLastKnown ($level$), asCurrent ($level$)
	$Null \mapsto MStream$	asUpdates ($frequency$, $level$)
Location Operators	$Item \mapsto Bool$	inCircle ($latLonFieldName$, lat_c, lon_c, r)
		inSquare ($latLonFieldName$, $lat1$, $lon1$, $lat2$, $lon2$)
	$Item \mapsto LatLon$	distort ($latLonFieldName$, $distortDistance$)
	$Item \mapsto Double$	distanceBetween ($latLonFieldName$, $LatLon$)
	...	...

5.3.2 Domain-Specific Packages. Domain-specific packages mainly consist of two types of functions, including *Provider*s that are designed to support various types of personal data and operators that are designed to support more types of operations.

Currently, we have implemented several packages for common data types (file, location, etc.) and certain types of operations (I/O, audio processing, etc.). Table 4 shows some examples of functions in the location package. With the location package, an app is able to access location data with *Provider*s such as `asLastKnown()`, `asUpdates()`, etc. It can also process location items using location-specific operators such as `inCircle()`, `distort()`, etc. More complex operations (signal processing, machine learning, etc.) have not been implemented yet, as they are rarely used in market apps.

5.3.3 Privacy-Preserving Operators. Among all kinds of operators, there are privacy-preserving operators designed to nudge app developers into considering preserving user privacy. Research has shown that a key reason for many apps being privacy-invasive is developers' lack of awareness and knowledge of user privacy [5]. Developers can be nudged into preserving users' privacy by offering privacy-preserving APIs [24].

Table 5. Examples of privacy-preserving operators in PrivacyStreams. Developers can use these operators to easily integrate privacy-preserving features in their apps.

Data type	Input ↦ Output	Name (Parameters)
Number	$Item \mapsto Number$	roundUp/roundDown ($numFieldName$, $numStep$)
String	$Item \mapsto String$	md5/sha1/sha256 ($stringFieldName$)
		elide ($stringFieldName$, $index$, $length$)
Location	$Item \mapsto LatLon$	distort ($latLonFieldName$, $distortDistance$)
		inCircle ($latLonFieldName$, lat_c, lon_c, r)
Audio	$Item \mapsto Num$	calcLoudness ($audioFieldName$)
	$Item \mapsto Bool$	hasHumanVoice ($audioFieldName$)
Image	$Item \mapsto Bool$	hasFace/hasCharacter ($imageFieldName$)
	$Item \mapsto ImageData$	blur ($imageFieldName$)
...	...	...

Table 5 shows some privacy-preserving operators for different kinds of data. For example, if a developer wants to collect users' message history with each contact while do not have to know each contact's identity, he or she can use the `elide()` operator as a parameter of `setField()` to get the elided phone numbers (such as "xxx-xxx-1234") instead of the original phone numbers. If an app is to provide local information in a city, it can use the `inCircle()` operator as the parameter of `filter()` to exclude the location items out of the city. There is no clear boundary between privacy-preserving operators and other operators, as some normal operators (such as `project()`, `count()`, etc.) can also reduce the data sensitivity, thereby reducing privacy concerns.

5.4 Support for Debugging and Testing

PrivacyStreams is compatible with existing debug and test frameworks, thus traditional methods and test suites can still be used in PrivacyStreams. In addition, PrivacyStreams provides several easy-to-use methods tailored for debugging and testing personal data streams. For example, developers can use `logAs()` function to instrument any intermediate step in a pipeline for debugging. They can also use `recordAs()` and `replay()` functions to record streams from real data sources and replay the streams for testing. Details about the debugging and testing support can be found in our API documentation [39].

6 IMPLEMENTATION

We implemented a prototype of PrivacyStreams in Android, including a library for developers to use PrivacyStreams in their apps and a static analyzer for auditors and app markets to analyze the apps developed with PrivacyStreams. Both the library and the static analyzer are available online. [2]

App developers can simply import the library into their Android projects to use PrivacyStreams APIs. At the time of writing of this paper, our library supports many common data types (including location, motion/environment sensors, contact, call log, messages, notification, browser event, etc.) and common operators related to these types of data. Our library is available as open-source software and more types of data and operations can be further added by either the authors or other contributors.

The static analyzer is implemented based on Soot [48]. The analyzer adopts the algorithm described in Section 4.3 to extract PrivacyStream DFGs from Android APKs (the bytecode of Android apps), which can describe how personal data is processed and what granularity of data is actually accessed. In order to guarantee the integrity of the results of the analysis, the analyzer also performs checks to ensure that the app only uses

[2] https://github.com/PrivacyStreams/PrivacyStreams

Table 6. The programming tasks used in the lab study. These tasks were crafted based on common use cases in popular market apps. In the study, participants were given a detailed task description including the use case, the expected output format, and a sample output. The code of UI was already given, thus participants only needed to fill a method body related to personal data, although they were allowed to add methods and classes.

Task ID	Task description
Task 1	Getting a list of email addresses of all contacts on the device. This task is common in social apps when they try to find friends based on the address book.
Task 2	Getting last known location. This task is common in location-based apps, such as an app that checks local weather information or searches nearby restaurants.
Task 3	Waiting for an SMS message from a given phone number and getting a 6-digit code in the message content. This feature is commonly used for two-factor authentication.
Task 4	Getting a list of local images on the device. This task is common in photo browsers and photo-editing apps.
Task 5	Monitoring location and getting notified when entering or exiting a given region (geofencing). This task is common in navigation apps and some location-based games.

PrivacyStreams to access personal data (by scanning all Android APIs used in the app) and the PrivacyStreams library packaged in the app is not modified from the original version (by comparing the library code structure with the original version).

With the static app analyzer, an auditor can directly assess privacy risk of Android apps, and the operators of an app market can generate privacy-related descriptions for apps to help users. We also implemented a proof-of-concept privacy description generator based on the analyzer, in order to help end-users understand the granularities of data used in their apps. It also has the potential to automatically generate more sophisticated descriptions (such as privacy policies).

7 EVALUATION

We evaluated PrivacyStreams by primarily looking at two aspects:

- Is PrivacyStreams easy to use for app developers? Specifically, is PrivacyStreams faster and easier to use than the standard Android APIs for common use cases? Also, what kinds of subjective feedback do developers give?
- Does PrivacyStreams lead to any privacy benefits? Specifically, is it feasible and accurate to extract the data processing steps from an app developed with PrivacyStreams and generate privacy descriptions based on it?

To answer these questions, we conducted two user studies, including a lab study and a free-form field study, and ran experiments on the apps developed in these studies. Both studies were conducted primarily at our university, with the study approved by our university's Institutional Review Board (IRB).

7.1 Lab Study

The goal of the lab study is to evaluate whether PrivacyStreams is faster and easier to use than the standard Android APIs for common use cases. In addition, we wanted to collect subjective feedback from developers about their usage of PrivacyStreams.

7.1.1 Study setup. We recruited 10 Android developers to come to our lab and complete programming tasks involving personal data. As shown in Table 6, we have five tasks crafted based on common use cases of popular

apps. The study used a within-subjects design, where participants used both the standard Android APIs and PrivacyStreams.

Participants in this study were students with Android development experience recruited from our campus. Originally we recruited 12 participants but excluded 2 of them due to lack of Android experience. Thus, in total, we had 10 participants with at least 3 months of Android development experience. We paid each participant 30 dollars plus a 0-10 dollar bonus according to the quality of completion (speed, correctness, etc.).

Participants were asked to come to our lab. We gave them a desktop computer with necessary development environment (Android Studio, emulator, browser, etc.) set up. The computer was also instrumented to record the screen during the tasks.

The study took about 100 minutes for each participant, including two 50-minute sessions. After a quick briefing about the study purpose and payment, we randomly assigned each participant an ID from 1A to 5A and 1P to 5P. The number (1 to 5) in the ID represents which task he/she needs to complete first and the letter (A or P) represents which group of APIs (Android or PrivacyStreams) to use in the first session. For example, participant 3A was asked to complete task 3 using Android APIs in the first session, then complete task 3 again using PrivacyStreams in the second session. The participants were randomly assigned other extra tasks if they complete the first task earlier.

Each session included a 10-minute walk-through, a 35-minute programming period, and a 5-minute debriefing. The walk-through was to help participants become familiar with PrivacyStreamsor refresh on Android APIs by guiding them through a warm-up task (getting loudness periodically). In the programming period, participants were asked to complete the assigned tasks. They were free to search the Internet for documentation and solutions. For a fairer comparison, we did not require the participants to request permissions at runtime using Android standard APIs, as PrivacyStreams automatically requests and handles runtime permissions. In the debriefing period, participants were asked to complete a questionnaire and take a semi-structured interview about their experiences in the session.

7.1.2 Programming efficiency and correctness. The results of the lab study are shown in Table 7. Overall, most participants spent less time (average of 17.8 vs. 25.6 minutes) and used less code (9.2 vs. 26.1 lines of code) for completing the tasks using PrivacyStreams than using the standard Android APIs. Developers were able to complete one extra task with PrivacyStreams on average, while no one was able to complete an extra task with Android standard APIs. Meanwhile, the code written with PrivacyStreams introduced fewer errors.

Among the 10 participants, one participant (4A) failed to complete the task with Android standard APIs. According to our interview, he had not used Android for a long time although he had 3 years of experience before. One participant (4P) completed the task faster using Android APIs (32 min) than using PrivacyStreams (34 min), because he spent too much time on locating the documentation. Three participants (1P, 5A, 5P) tried to request permissions at runtime using Android standard APIs although it is not required. We exclude the time spent and code used for that in our result.

We manually checked the participants' code and found two types of errors, including:

- **E1 – Corner cases not considered.** 4 participants in Android and 1 participant in PrivacyStreams missed some corner cases. For example, 2A and 2P didn't consider the case if last known location is unavailable. 3A and 3P directly tried to get a substring of the message before checking the message length.
- **E2 – Resource not released.** 5 participants forgot to release the resource after getting the data using Android APIs. For example, 1A, 1P and 4P didn't close the cursor after querying the database. 3A and 3P didn't stop the SMS broadcast receiver after getting the needed message. The main reason for such errors was that they copied the incorrect or inappropriate code from the Internet.

PrivacyStreams was able to avoid E2 automatically as it hides these technical details internally. It also made the program logic clearer so that developers noticed E1 errors more easily.

Table 7. The result of task completion in the lab study. Each row in the table represents a participant. The first column is the participant ID representing the first task ID and order of API groups for the participant. For example, participant 2P was asked to complete task 2 using PrivacyStreams in the first session, then complete task 2 again using Android standard APIs in the second session. The extra tasks after the first were randomly assigned. The "Time" column is the duration of time (in minutes) spent by the participant on the first task. The "LOC" column is the number of lines of code used for the first task (only Java code was counted, and each pipeline function (`filter`, `asList`, etc.) in PrivacyStreams was counted as a new line even though it was not an actual new line). The "Errors" column represents the types of issues found in the code. E1 is "corner cases not considered" and E2 is "resource not released". The "Extra" column is the IDs of extra tasks completed after the first task (T1, T2, ... are short for Task 1, Task 2, ...). The "Using Android APIs" columns for 4A are empty because the participant failed to complete the given task using Android standard APIs. The result shows that developers can complete tasks more efficiently and correctly with PrivacyStreams.

	# Years Experience		Using Android APIs				Using PrivacyStreams			
	Java	Android	Time	LOC	Errors	Extra	Time	LOC	Errors	Extra
1A	5	3	25	41	E2	-	16	9		T3, T4
1P	4	1.5	15	32	E2	-	8	7		T2
2A	3	0.3	24	9	E1	-	8	8		T5, T1
2P	3	2	21	10	E1	-	9	9		T4
3A	1	0.5	34	35	E1, E2	-	23	9	E1	T3
3P	2	1.5	35	36	E1, E2	-	27	14		-
4A	4	3	-	-	-	-	28	7		-
4P	4	3	32	17	E2	-	34	12		-
5A	5	2.5	27	30		-	12	9		T2
5P	5	3	17	25		-	13	8		T1, T4
Average	3.6	2.0	25.6	26.1		0	17.8	9.2		1

Table 8. Performance of using PrivacyStreams to complete the tasks (as shown in Table 6) as the first task and as an extra task. Each cell contains the count of completions, followed by the average time spent for each completion. The result shows that developers can complete extra tasks quickly after getting familiar with the APIs in the first task.

	As the first task	As an extra task
Task 1	2 (12.0)	2 (7.5)
Task 2	2 (8.5)	2 (10.5)
Task 3	2 (25.0)	1 (11.0)
Task 4	2 (31.0)	4 (7.3)
Task 5	2 (12.5)	1 (11.0)
Overall	10 (17.8)	10 (8.7)

Our results also suggest that PrivacyStreams is easy to learn. We only gave participants a 10-minute tutorial for them to learn the PrivacyStreams APIs. Despite this short amount of time, developers were more efficient in completing the tasks compared with using standard Android APIs. Meanwhile, after completing the first task, they became even more efficient on the extra tasks. As shown in Table 8, the average time spent on an extra task is 8.7 minutes, which is much shorter than the first task. The reasons may include (1) the developers had a better understanding of the APIs after completing the first task, and (2) PrivacyStreams uses a set of uniform APIs for different data types, making it easy to migrate from one data type to another. Based on these results, we believe developers can learn PrivacyStreams APIs quickly in real-world scenarios.

Table 9. Subjective feedback collected from questionnaires in the lab study. The five columns are the five usability characteristics we wanted to evaluate. Each participant was asked to give a rating for each characteristic, and each rating is an integer in a Likert scale of 1 to 7 indicating the level of satisfaction with the characteristic, with the median and the interquartile range (IQR) shown. We compared the two sets of ratings for Android standard APIs and PrivacyStreams with a two-tailed Mann-Whitney U test, and the "p-value" column is the possibility to accept the NULL hypothesis (i.e. there is no significant difference between two samples). If the p-value is less than 0.05, then the NULL hypothesis is rejected, meaning there is a significant difference between the two set of ratings. The result shows that PrivacyStreams is generally better than Android standard APIs in most usability aspects with a significant difference.

		Simplicity	Learnability	Productivity	Debug and test	Maintainability
Android APIs	Median	5	4.5	4	4	4
	IQR	4 - 5	3 - 6	3.5 - 4	3 - 4.5	4 - 4.5
PrivacyStreams	Median	6	6	6.5	6	6.5
	IQR	5.25 - 6.75	4.25 - 6	6 - 7	4.5 - 6	5 - 7
p-value		0.02088	0.25848	0.00152	0.02320	0.00736

7.1.3 Subjective feedback. In order to understand what developers thought of PrivacyStreams, we solicited feedback through a post-study questionnaire and a short exit interview. The questionnaires and interviews probed five usability characteristics: simplicity in completing the tasks, ease of learning, productivity, ease to debug and test, and ease in maintaining the code.

Participants were given some statements related to the characteristics for each API. For example, the statement "The tasks were easy to complete with Android APIs" is related to simplicity. "I can become productive quickly with PrivacyStreams" is related to productivity, etc. We let participants rate their levels of agreement with each statement on a 7-point Likert scale from 1 (strongly disagree) to 7 (strongly agree). The results are shown in Table 9.

Overall, developers were more positive with using PrivacyStreams compared to using Android standard APIs. PrivacyStreams obtained a higher median for all five characteristics. A Mann-Whitney U test showed statistically significant differences for simplicity, productivity, debug and test, and maintainability (p-value < 0.05). However, there was no statistically significant difference in learnability. One possible explanation is that we only gave a 10-minute tutorial to our users to learn our APIs. Most participants (7/10) mentioned the difficulty to get familiar with the APIs in such a limited time. However, as mentioned earlier, despite this short learning period, our results show that these developers were still able to complete the tasks successfully and faster using PrivacyStreams.

We also received helpful feedback from our post-study interviews. For example, as PrivacyStreams uses weak types for item field values, two participants suggested an IDE plug-in that supports automated type inference, so that developers do not have to manually check the documentation to determine the type of fields. Two participants suggested adding support for connecting PrivacyStreams with other stream frameworks (e.g. Java 8 Streams [47] and RxJava [41]) for even better utility.

7.2 Field Study

The goal of the field study was to evaluate the utility and usability of PrivacyStreams with respect to real apps.

7.2.1 Study setup. We recruited 5 experienced Android developers to develop Android apps with PrivacyStreams for this study. One of them was recruited online (by contacting the email address found on GitHub) and the others were from our campus. All of them have at least 2 years of Android development experience, and 3 of them have released apps on Google Play or created open-source apps with more than 1,000 stars on GitHub. We paid each participant 70 dollars upon successfully completing the task.

Table 10. The number of lines of Java code modified (including removed and added, in the "Modified LOC" column) to develop apps with PrivacyStreams, and the total number of lines of Java code (in the "Total LOC" column) in the developed apps.

App	Modified LOC	Total LOC	Source code link
Speedometer	-0 +43	43	https://github.com/PrivacyStreams/speedometer
Lock screen app	-41 +16	239	https://github.com/PrivacyStreams/lastcaller
Weather app	-35 +19	795	https://github.com/PrivacyStreams/rex-weather
Sleep monitor	-21 +21	15,887	https://github.com/PrivacyStreams/sleepminder
Album app	-86 +55	33,453	https://github.com/PrivacyStreams/Album

We selected 5 apps as the tasks (one app for one participant) by searching on GitHub for Android apps, including an album app (accessing local image files and camera), a sleep monitor (accessing microphone), a weather app (accessing location), a speedometer (accessing GPS), a lock-screen app that shows call log on the screen (accessing call log). The speedometer was developed from scratch, while the other apps were rewritten from existing open-source apps. Participants were asked to only focus on the personal data processing part and were allowed to simplify other parts (e.g. UI, network, etc.) or reuse from the original apps.

Participants were briefed about the study through e-mail and then assigned a task (the app to develop). They were given sufficient time (2 weeks) to complete the tasks. During the tasks, they were allowed to ask any questions over email. After successfully completing the tasks, they were asked to upload the source code. We also collected subjective feedback over email.

7.2.2 Programming effort. Table 10 shows the list of apps developed with PrivacyStreams. Most of them only need to modify less than 100 lines of code to fit into PrivacyStreams. For the rewritten apps, using PrivacyStreams can reduce the lines of source code (more code can be removed if we recursively delete the methods and classes used in the deleted code).

7.2.3 Subjective feedback. In this section, we highlight some of the qualitative results and lessons learned from the interview. Most developers reported that PrivacyStreams was easy to install and easy to use. Particularly, one developer praised the automated runtime permission mechanism. Another liked the uniform APIs for personal data that was originally difficult to access using Android standard APIs. A third developer liked the functional programming model that aligns with the idea of reactive programming [41], which is widely used in market apps.

However, there were some aspects of PrivacyStreams that can potentially be improved. One developer was worried about the difficulty to rewrite a large-sized app (for instance, a navigation app that accesses low-level information, such as the events of location provider changed, the events of accuracy changed, etc.), as PrivacyStreams omitted some low-level data for simplicity. We may need to conduct more common use cases to decide what kind of data can be omitted. Two developers complained about the lack of examples, and two developers had found bugs in the PrivacyStreams source code (which have been fixed). Both the documentation and the library need several iterations before an official release.

7.2.4 Runtime performance. We compared the runtime performance of the apps rewritten using PrivacyStreams with the original versions by running them both for an hour. There was no noticeable difference on latency and battery lifetime as expected since the personal data processing component of the apps are a small fraction of the entire app.

Table 11. The performance of static analysis for the apps developed with PrivacyStreams. The "Time spent" column is the time spent by the analyzer to extract PrivacyStream DFGs from the APK, and the "Generated description" column is the privacy descriptions generated based on the extracted PrivacyStream DFGs.

App	Time spent (s)	Generated description
Speedometer	12.17	This app requests LOCATION permission to get the speed continuously.
Lock screen app	2.94	This app requests CALL_LOG permission to get the last missed call.
Weather app	14.72	This app requests LOCATION permission to get the city-level location.
Sleep monitor	13.03	This app requests MICROPHONE permission to get how loud it is.
Album app	14.36	This app requests STORAGE permission to get all local images.

7.3 Support for Privacy

The goal of this study is to evaluate the feasibility and accuracy of PrivacyStream DFG analysis and privacy description generation for apps developed with PrivacyStreams.

7.3.1 Study setup. We used the 5 apps developed in the field study for analysis. Our analysis included a PrivacyStream DFG extraction step as described in Section 4.3 and a proof-of-concept description generation step. The analysis was performed on a MacBook Pro with a 2.3 GHz Intel i7 processor and a 16 GB RAM, and the operating system was MacOS Sierra 10.12.4.

7.3.2 PrivacyStream DFG extraction. We applied the Android static analyzer on the apps developed with PrivacyStreams. As shown in Table 11, all five apps can be analyzed, and we can extract detailed steps of data processing (i.e. PrivacyStream DFG) within 15 seconds. Note that the time spent for analysis was not correlated with the number of source lines of code shown in Table 10 because the actual sizes of apps were influenced by the third-party libraries used in the apps. For example, the analysis for the lock screen app was fast because it did not use any heavy-weight third party libraries. Based on a manual assessment on the analysis result, all of the extracted PrivacyStream DFGs precisely described the personal data processing steps.

7.3.3 Generating privacy descriptions. Privacy description generation is not the main contribution in this paper. Here we only describe the implementation of a simple rule-based prototype to show its feasibility.

The description generator was designed to describe the permissions used in a PrivacyStreams app. The format we used to generate permission descriptions is "This app requests WHAT_PERMISSION to get WHAT_DATA", where WHAT_PERMISSION is the Android permission needed to access the data and WHAT_DATA is the data actually accessed by the app. WHAT_PERMISSION can be inferred from the provider, and WHAT_DATA is inferred based on the transformations in PrivacyStream DFG. For example, the lock screen app contains the following code:

```
1 UQI.getData(Call.getLogs(), Purpose.UTILITY("showing on lock screen")) // get call logs
2     .filter(Call.TYPE, "missed") // only keep the items whose TYPE is "missed"
3     .sortBy(Call.TIMESTAMP)  // sort the items by TIMESTAMP field, in ascending order
4     .reverse() // reverse the items, the new stream is in descending order of TIMESTAMP
5     .getFirst() // get the first item, i.e. the call log with largest TIMESTAMP
```

It is easy to determine that the requested permission is "CALL_LOG" based on the provider (`Call.getLogs()`). The text for WHAT_DATA is transformed along the pipeline. Specifically, the initial WHAT_DATA text is "all calls" in the first line, meaning the first line gets all call logs. Then it is transformed to "all missed calls" after the second line, as the second line only keeps the calls whose TYPE is "missed". Then the text is transformed to "the last missed call" after line 3-5, as the pipeline selects an item with the largest TIMESTAMP (i.e. last). Finally,

combining the text for WHAT_PERMISSION and WHAT_DATA, the privacy description is generated as "This app request CALL_LOG permission to get the last missed call".

Table 10 shows the generated descriptions for the five apps. We did not evaluate the user's attitude towards these descriptions, which is not the main focus of this paper. However, we believe these descriptions with granularity information can help users better understand privacy, as compared to the situation without PrivacyStreams, where users can only see what permissions are used in an app. Further evaluation will be performed in our future work.

8 DISCUSSION

8.1 Design Considerations

In this section, we highlight several considerations about why we adopt current design instead of other alternatives.

Why functional programming model? There might be alternative approaches that can also achieve transparency, such as an imperative programming model with a lot of fine-grained APIs offering different levels of data. We choose a functional programming model mainly because of two reasons. First, a functional programming model can centralize the data processing steps and reduce external states, making it easier to analyze. Second, functional programming is widely used in data processing domain (Spark [57], LINQ [8], etc.) and is becoming popular on mobile platforms (ReactiveX [41], Java 8 Streams [47]), which demonstrates its convenience for developers.

Why generalizing non-stream data as stream? We used a uniform class (Stream) to represent all kinds of personal data. Such design may lead to a little confusion for developers as some types of data (such as the current location or an audio record) are not a stream. We generalize all types of data mainly for ease of use, as developers can use the same set of operations for different kinds of personal data. Moreover, developers often will not see the Stream class abstraction as they only need to use the functions to create a pipeline.

8.2 Limitations

This section lists a few of limitations of PrivacyStreams.

Some complex use cases are not covered. There are some use cases that are not covered by PrivacyStreams currently. For instance, if an app uses machine learning on raw data to infer context, although the context information accessed by the app is not at the finest granularity, PrivacyStreams is unable to disclose the granularity as the operations are not supported by the library. We plan to add more data types and operators (e.g. machine learning algorithms) to PrivacyStreams in order to support more complex use cases. However, considering that most apps in the app store do not use such complex operations on personal data, we believe PrivacyStreams is able to cover a lot of use cases.

Third-party libraries may access personal data. Some apps may include third-party libraries that access personal data using Android standard APIs. PrivacyStreams cannot analyze how the libraries use the data because it requires the data being accessed and processed using PrivacyStreams APIs. One solution is to identify the third-party libraries and analyze the app and libraries separately. Another solution, although not immediately applicable, is to require all third-party libraries be developed with PrivacyStreams APIs.

Code obfuscation may bypass static analysis. Our static analyzer did not consider code obfuscation, which is a technique used in released apps to minify app size and protect apps from reverse-engineering. With code obfuscation, the identifiers and control flow may be modified and even hidden from the source code, making the analyzer hard to extract data processing steps and verify that personal data is only accessed through PrivacyStreams. One solution to address this issue is to deobfuscate the app based on the code structure that cannot be obfuscated. Currently, developers can enable static analysis by keeping the PrivacyStreams library unobfuscated in their apps.

8.3 Future Work

Besides addressing the limitations, we also plan to explore the following directions in the future.

How to generate better privacy descriptions for end-users. We implemented a simple rule-based privacy description generator based on PrivacyStreams data flow graph, which can generate simple sentences to describe what granularities of data are accessed. We believe that the PrivacyStreams DFG can be used to generate more detailed and easier-to-understand description or visualization for end-users.

How to make use of the purposes given by developers. PrivacyStreams adopts a "purpose" parameter in API design, in order to enable and motivate developers to explain the purpose of data use. It is just a placeholder currently, and we have not done anything with it yet. However, we believe that the "purpose" parameter can help with some interesting applications in the future, such as generating privacy policies and verifying the purpose given by developers against the actual behavior written in the code.

9 RELATED WORK

9.1 Privacy-Aware APIs and Languages

Many frameworks have been designed to preserve privacy in apps by offering new APIs or programming languages. One of the main concepts of such frameworks is letting apps access locally-computed high-level results instead of raw data. For example, RePriv [20] and MoRePriv [12] propose a platform for app personalization, where apps can use platform APIs to access user's personalities instead of accessing the raw data and inferring on their own. Similarly, openPDS/SafeAnswers [13] introduces a personal data store that allows applications ask questions about the data (by sending the code to be run against the data) and the answer will be sent back to them. Unlike RePriv and MoRePriv that only support a specific type of data (users' personalities) and openPDS that requires data stored in their database, PrivacyStreams is a more general approach as it allows apps access various types of personal data directly from the system.

Another main concept of such APIs and languages is to make apps easier to audit by improving the transparency of personal data processing. Yang *et al.* introduce a new language named Jeeves [55] that is able to enforce privacy policies (eg. only Alice can see my location) automatically. Ernst *et al.* [17] propose a type system for Android, in which app developers can annotate their code to expose information flow. We also focus on improving transparency in personal data processing, however, we don't need developers to put extra efforts on privacy (such as writing privacy policies and annotating code), instead, we make it easier for developers to make use of personal data.

9.2 Easing Developers' Efforts on Personal Data Accessing and Processing

There are many approaches proposed to make it easier for developers to access and process personal data. For example, Funf [2] and AWARE [19] can help developing mobile data collection apps and context-sensing apps. They offer a framework that continuously collects all kinds of raw data into a local database so that developers can read data from the database directly. Senergy [26], Li *et al.* [29] and PADA [34] aim to assist developing energy-aware apps. The Context Toolkit [43] and Google Awareness API make it easier to develop context-aware applications. Epistenet [11] facilitate the access and processing of semantically related personal data. PrivacyStreams can also ease app developers' efforts, but its main focus is the privacy aspect.

9.3 Analyzing Apps for Privacy-Related Properties

Many approaches have been proposed to analyze mobile apps for permission use and privacy. Some of them analyze code to understand the information flow in apps. Main techniques in such approaches include static data flow analysis [3, 22] and dynamic taint tracking [16, 46]. However, static analysis is hard to capture the information flow under the pressure of code complexity, such as inter-process communication, implicit flows,

aliases, obfuscations, etc. Dynamic analysis often requires to run apps manually or using an automated input generator [30], which either suffer from poor scalability or low code coverage. Some other approaches analyze non-code resources to understand the purpose of personal data use. The resources can be the text descriptions in the app market [37, 40], the textual information in app code [51], and the user interface [23, 28]. The effectiveness of such approaches highly depends on the quality of these resources, and there is no guarantee that these resources can correctly reflect apps' actual behaviors. PrivacyStreams also uses code analysis to disclose what granularity of data is accessed and how it is processed in mobile apps. However, PrivacyStreams uses a new set of APIs for personal data to make the code much easier to analyze.

9.4 Data Processing and Functional Programming

Many data processing tools and frameworks adopt a functional programming model. The widely-used text processing tools in Unix, sed and AWK, are early examples of data-driven programming, which is similar to functional programming as its code describes the processing required rather than defining a sequence of steps to be taken. In the era of cloud computing and big data, many large-scale data processing frameworks emerge. The most influential ones of them, Hadoop MapReduce [14] and Apache Spark [57], adopt a functional programming model. Functional programming is becoming popular in more areas and platforms. LINQ [8] and Java 8 Streams [47] provide functional programming models to query and process data in .NET and Java respectively. ReactiveX [41], an popular cross-platform API for asynchronous programming, also uses a functional programming model. The popularity of these frameworks demonstrates the benefits of using functional programming for data processing, which partly inspires PrivacyStreams. However, PrivacyStreams is different from existing frameworks because it is tailored for personal data on mobile devices, where privacy is important.

10 CONCLUDING REMARKS

This paper presents PrivacyStreams, a framework for enabling transparency in personal data processing for mobile apps. PrivacyStreams offers a functional programming model with a set of uniform APIs, which allow auditors and end-users to better understand the privacy implications of mobile apps through exposing the detailed processing steps and data granularity of personal data accesses. Meanwhile, PrivacyStreams can also ease developers' programming efforts on dealing with personal data in their apps.

Currently, PrivacyStreams is implemented for Android, including a library for app developers and a static analyzer for auditors and app markets. Based on lab and field studies, we demonstrate that PrivacyStreams API is easy for developers to use when compared to the existing standard Android API. Through experiments on the apps developed with PrivacyStreams, we show that the static analyzer can extract detailed steps of data processing accurately and efficiently, which could become an essential step toward better access control and privacy protection for mobile sensitive data.

ACKNOWLEDGMENTS

The work is supported in part by the National Key Research and Development Program 2016YFB1000105, the National Natural Science Foundation of China under Grant No. 61421091, the National Science Foundation under Grant No. CSR-1526237 and CNS-1564009, the scholarship from China Scholarship Council under Grant No. 201606010240 and multiple Google Faculty Research Awards on mobile and IoT privacy. This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-15-2-0281. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

REFERENCES

[1] Yuvraj Agarwal and Malcolm Hall. 2013. ProtectMyPrivacy: Detecting and Mitigating Privacy Leaks on iOS Devices Using Crowdsourcing. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*. ACM, New York, NY, USA, 97–110. https://doi.org/10.1145/2462456.2464460

[2] Nadav Aharony, Alan Gardner, and Cody Sumter. 2011. Funf open sensing framework. (2011). Retrieved July 1, 2017 from http://funf.org/

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[4] Daniel Avrahami and Scott E. Hudson. 2006. Responsiveness in Instant Messaging: Predictive Models Supporting Inter-personal Communication. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*. ACM, New York, NY, USA, 731–740. https://doi.org/10.1145/1124772.1124881

[5] Rebecca Balebako, Abigail Marsh, Jialiu Lin, Jason I Hong, and Lorrie Faith Cranor. 2014. The privacy and security behaviors of smartphone app developers. In *Proceedings of Workshop on Usable Security*.

[6] Louise Barkhuus, Barry Brown, Marek Bell, Scott Sherwood, Malcolm Hall, and Matthew Chalmers. 2008. From Awareness to Repartee: Sharing Location Within Social Groups. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 497–506. https://doi.org/10.1145/1357054.1357134

[7] Joshua Bloch. 2006. How to Design a Good API and Why It Matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 506–507. https://doi.org/10.1145/1176617.1176622

[8] Don Box and Anders Hejlsberg. 2007. LINQ: .NET Language-Integrated Query. (2007). Retrieved July 1, 2017 from https://msdn.microsoft.com/en-us/library/bb308959.aspx

[9] Zhenyu Chen, Mu Lin, Fanglin Chen, Nicholas D. Lane, Giuseppe Cardone, Rui Wang, Tianxing Li, Yiqiang Chen, Tanzeem Choudhury, and Andrew T. Campbell. 2013. Unobtrusive Sleep Monitoring Using Smartphones. In *Proceedings of the 7th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth '13)*. 145–152. https://doi.org/10.4108/icst.pervasivehealth.2013.252148

[10] Yohan Chon, Nicholas D. Lane, Fan Li, Hojung Cha, and Feng Zhao. 2012. Automatically Characterizing Places with Opportunistic Crowdsensing Using Smartphones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp '12)*. ACM, New York, NY, USA, 481–490. https://doi.org/10.1145/2370216.2370288

[11] Sauvik Das, Jason Wiese, and Jason I. Hong. 2016. Epistenet: Facilitating Programmatic Access & Processing of Semantically Related Mobile Personal Data. In *Proceedings of the 18th International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI '16)*. ACM, New York, NY, USA, 244–253. https://doi.org/10.1145/2935334.2935349

[12] Drew Davidson, Matt Fredrikson, and Benjamin Livshits. 2014. MoRePriv: Mobile OS Support for Application Personalization and Privacy. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, New York, NY, USA, 236–245. https://doi.org/10.1145/2664243.2664266

[13] Yves-Alexandre de Montjoye, Erez Shmueli, Samuel S. Wang, and Alex Sandy Pentland. 2014. openPDS: Protecting the Privacy of Metadata through SafeAnswers. *PLOS ONE* 9, 7 (July 2014), 1–9. https://doi.org/10.1371/journal.pone.0098790

[14] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. 2012. Efficient Big Data Processing in Hadoop MapReduce. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 2014–2015. https://doi.org/10.14778/2367502.2367562

[15] Afsaneh Doryab, Jun-Ki Min, Jason Wiese, John Zimmerman, and Jason I Hong. 2014. Detection of Behavior Change in People with Depression.. In *AAAI Workshop: Modern Artificial Intelligence for Health Analytics*.

[16] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 393–407. http://dl.acm.org/citation.cfm?id=1924943.1924971

[17] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. 2014. Collaborative Verification of Information Flow for a High-Assurance App Store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 1092–1104. https://doi.org/10.1145/2660267.2660343

[18] Deborah Estrin. 2014. Small Data, Where N = Me. *Commun. ACM* 57, 4 (April 2014), 32–34. https://doi.org/10.1145/2580944

[19] Denzil Ferreira, Vassilis Kostakos, and Anind K Dey. 2015. AWARE: mobile context instrumentation framework. *Frontiers in ICT* 2 (2015), 6.

[20] Matthew Fredrikson and Benjamin Livshits. 2011. RePriv: Re-imagining Content Personalization and In-browser Privacy. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*. IEEE Computer Society, Washington, DC, USA, 131–146. https://doi.org/10.1109/SP.2011.37

[21] Stylianos Gisdakis, Thanassis Giannetsos, and Panos Papadimitratos. 2016. Android Privacy C(R)Ache: Reading Your External Storage and Sensors for Fun and Profit. In *Proceedings of the 1st ACM Workshop on Privacy-Aware Mobile Computing (PAMCO '16)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2940343.2940346

[22] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe.. In *NDSS*. Citeseer.

[23] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1036–1046. https://doi.org/10.1145/2568225.2568301

[24] Shubham Jain and Janne Lindqvist. 2014. Should I protect you? Understanding developers' behavior to privacy-preserving APIs. In *Workshop on Usable Security*.

[25] Younghee Jung, Per Persson, and Jan Blom. 2005. DeDe: Design and Evaluation of a Context-enhanced Mobile Messaging System. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*. ACM, New York, NY, USA, 351–360. https://doi.org/10.1145/1054972.1055021

[26] Aman Kansal, Scott Saponas, A.J. Bernheim Brush, Kathryn S. McKinley, Todd Mytkowicz, and Ryder Ziola. 2013. The Latency, Accuracy, and Battery (LAB) Abstraction: Programmer Productivity and Energy Efficiency for Continuous Mobile Context Sensing. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 661–676. https://doi.org/10.1145/2509136.2509541

[27] Joon-Gyum Kim, Chia-Wei Wu, Alvin Chiang, JeongGil Ko, and Sung-Ju Lee. 2016. A Picture is Worth a Thousand Words: Improving Mobile Messaging with Real-time Autonomous Image Suggestion. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications (HotMobile '16)*. ACM, New York, NY, USA, 51–56. https://doi.org/10.1145/2873587.2873602

[28] Yuanchun Li, Yao Guo, and Xiangqun Chen. 2016. PERUIM: Understanding Mobile Application Privacy with permission-UI Mapping. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '16)*. ACM, New York, NY, USA, 682–693. https://doi.org/10.1145/2971648.2971693

[29] Yuanchun Li, Yao Guo, Junjun Kong, and Xiangqun Chen. 2015. Fixing sensor-related energy bugs through automated sensing policy instrumentation. In *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 321–326. https://doi.org/10.1109/ISLPED.2015.7273534

[30] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A Lightweight UI-guided Test Input Generator for Android. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, Piscataway, NJ, USA, 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[31] Robert LiKamWa, Yunxin Liu, Nicholas D. Lane, and Lin Zhong. 2013. MoodScope: Building a Mood Sensor from Smartphone Usage Patterns. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*. ACM, New York, NY, USA, 389–402. https://doi.org/10.1145/2462456.2464449

[32] Jialiu Lin, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. 2012. Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp '12)*. ACM, New York, NY, USA, 501–510. https://doi.org/10.1145/2370216.2370290

[33] Xuan Lu, Wei Ai, Xuanzhe Liu, Qian Li, Ning Wang, Gang Huang, and Qiaozhu Mei. 2016. Learning from the Ubiquitous Language: An Empirical Analysis of Emoji Usage of Smartphone Users. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '16)*. ACM, New York, NY, USA, 770–780. https://doi.org/10.1145/2971648.2971724

[34] Chulhong Min, Seungchul Lee, Changhun Lee, Youngki Lee, Seungwoo Kang, Seungpyo Choi, Wonjung Kim, and Junehwa Song. 2016. PADA: Power-aware Development Assistant for Mobile Sensing Applications. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '16)*. ACM, New York, NY, USA, 946–957. https://doi.org/10.1145/2971648.2971676

[35] Jun-Ki Min, Afsaneh Doryab, Jason Wiese, Shahriyar Amini, John Zimmerman, and Jason I. Hong. 2014. Toss 'N' Turn: Smartphone As Sleep and Sleep Quality Detector. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. 477–486. https://doi.org/10.1145/2556288.2557220

[36] Jun-Ki Min, Jason Wiese, Jason I. Hong, and John Zimmerman. 2013. Mining Smartphone Data to Classify Life-facets of Social Relationships. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work (CSCW '13)*. ACM, New York, NY, USA, 285–294. https://doi.org/10.1145/2441776.2441810

[37] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications.. In *USENIX Security Symposium*. Washington, D.C., 527–542.

[38] Martin Pielot, Tilman Dingler, Jose San Pedro, and Nuria Oliver. 2015. When Attention is Not Scarce - Detecting Boredom from Mobile Phone Usage. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. 825–836. https://doi.org/10.1145/2750858.2804252

[39] PrivacyStreams. 2016. PrivacyStreams API documentation. (2016). Retrieved July 1, 2017 from https://privacystreams.github.io/

[40] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications*

Security (CCS '14). ACM, New York, NY, USA, 1354–1365. https://doi.org/10.1145/2660267.2660287

[41] ReactiveX. 2017. An API for asynchronous programming with observable streams. (2017). Retrieved July 1, 2017 from http://reactivex.io/

[42] L. Ruge, B. Altakrouri, and A. Schrader. 2013. SoundOfTheCity - Continuous noise monitoring for a healthy city. In *2013 IEEE International Conference on Pervasive Computing and Communications Workshops*. 670–675. https://doi.org/10.1109/PerComW.2013.6529577

[43] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. 1999. The Context Toolkit: Aiding the Development of Context-enabled Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, New York, NY, USA, 434–441. https://doi.org/10.1145/302979.303126

[44] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. 2016. Toward a Framework for Detecting Privacy Policy Violations in Android Application Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2884781.2884855

[45] Jacopo Staiano, Nuria Oliver, Bruno Lepri, Rodrigo de Oliveira, Michele Caraviello, and Nicu Sebe. 2014. Money Walks: A Human-centric Study on the Economics of Personal Mobile Data. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '14)*. ACM, New York, NY, USA, 583–594. https://doi.org/10.1145/2632048.2632074

[46] Mingshen Sun, Tao Wei, and John C.S. Lui. 2016. TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 331–342. https://doi.org/10.1145/2976749.2978343

[47] Raoul-Gabriel Urma. 2014. Processing Data with Java SE 8 Streams. (2014). Retrieved July 1, 2017 from http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

[48] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press. http://dl.acm.org/citation.cfm?id=781995.782008

[49] Emanuel von Zezschwitz, Sigrid Ebbinghaus, Heinrich Hussmann, and Alexander De Luca. 2016. You Can't Watch This!: Privacy-Respectful Photo Browsing on Smartphones. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 4320–4324.

[50] Dashun Wang, Dino Pedreschi, Chaoming Song, Fosca Giannotti, and Albert-Laszlo Barabasi. 2011. Human Mobility, Social Ties, and Link Prediction. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '11)*. ACM, New York, NY, USA, 1100–1108. https://doi.org/10.1145/2020408.2020581

[51] Haoyu Wang, Jason Hong, and Yao Guo. 2015. Using Text Mining to Infer the Purpose of Permission Use in Mobile Apps. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 1107–1118. https://doi.org/10.1145/2750858.2805833

[52] Rui Wang, Fanglin Chen, Zhenyu Chen, Tianxing Li, Gabriella Harari, Stefanie Tignor, Xia Zhou, Dror Ben-Zeev, and Andrew T. Campbell. 2014. StudentLife: Assessing Mental Health, Academic Performance and Behavioral Trends of College Students Using Smartphones. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '14)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2632048.2632054

[53] Rui Wang, Gabriella Harari, Peilin Hao, Xia Zhou, and Andrew T. Campbell. 2015. SmartGPA: How Smartphones Can Assess and Predict Academic Performance of College Students. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 295–306. https://doi.org/10.1145/2750858.2804251

[54] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. 2015. Android Permissions Remystified: A Field Study on Contextual Integrity.. In *USENIX Security*, Vol. 15.

[55] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/2103656.2103669

[56] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*. ACM, New York, NY, USA, 1043–1054. https://doi.org/10.1145/2508859.2516676

[57] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[58] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy (S&P '12)*. IEEE, 95–109.

Received May 2017; accepted July 2017