Privacy Enhancements for Android

Dan Hallenbeck, Michael Lack, Jason McPeak, Irwin Reyes
{dan.hallenbeck | michael.lack | jason.mcpeak | irwin.reyes}@twosixlabs.com

Motivation

Mobile devices such as smartphones and tablets are critical tools that billions of users worldwide [1] depend on for a variety of needs: communications, banking, news, navigation, and entertainment, among many others. A diverse ecosystem of third-party apps enables this versatility by utilizing rich telecommunications, data processing, and sensing capabilities common on mobile devices. These capabilities, combined with users' heavy reliance on these platforms, result in mobile devices holding a vast amount of highly sensitive personal data that apps can access.

Mobile operating systems such as Android implement *permissions* to mitigate the risk of apps misusing sensitive data. Android's permission system gives users control over what sensitive resources apps may access. For example, permissions regulate access to the user's contact list, current location, and SMSs [2]. However, recent investigations into mobile privacy have found issues in the usability and effectiveness of existing permission systems: apps can circumvent permissions [3]; users' privacy preferences are contextually dependent [4]; and invisible third-party services embedded in apps account for the bulk of private data collection [5]. Android's permission system gives insufficient transparency and control given the realities of the mobile app ecosystem and actual user expectations.

Approach

Privacy Enhancements for Android (PE for Android) [6] is a platform for exploring concepts in regulating access to private information on mobile devices. Our goal is to create an extensible privacy system that abstracts away the details of various privacy-preserving technologies. We strive to allow app developers to safely leverage state-of-the-art privacy techniques without knowledge of esoteric underlying technologies. Additionally, PE for Android helps users to take ownership of their private information by presenting them with more intuitive controls and permission enforcement. We developed PE for Android as a fork of the Android Open Source Project (AOSP) [7] release for Android 9 "Pie." This effort modified 11 of Android 9's Git repositories and added 3 new repositories of supplemental resources to the code base.

PE for Android is a set of extensions and interfaces integrated into the Android OS, similar to Google Play services and SE for Android. Our platform introduces new APIs, services, and a Privacy Abstraction Layer (PAL). The goal of these components is to move sensitive data processing out of the application process space, and into these new services that implement privacy preserving technologies. Once the sensitive information is transformed it may then be returned to the application.

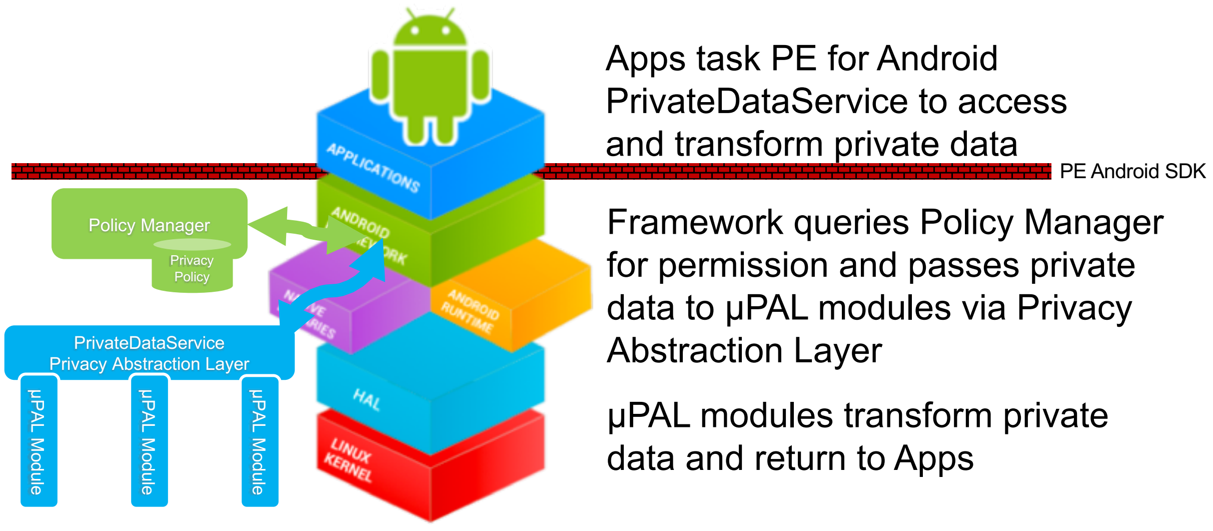

Apps task PE for Android PrivateDataService to access and transform private data
PE Android SDK
Framework queries Policy Manager for permission and passes private data to µPAL modules via Privacy Abstraction Layer
µPAL modules transform private data and return to Apps
Policy Manager
Privacy Policy
PrivateDataService
Privacy Abstraction Layer
µPAL Module
µPAL Module
µPAL Module
APPLICATIONS
ANDROID FRAMEWORK
NATIVE LIBRARIES
ANDROID RUNTIME
HAL
LINUX KERNEL

Figure 1 PE for Android architecture and data flow

As shown in Figure 1, these APIs facilitate the development of swappable components that are invoked when apps request private data. This includes *Policy Managers* that can log and help users decide on requests to access to sensitive information. Further, PE for Android offers the *Private Data Service* and associated modules dubbed *µPALs*, which transform private data into less sensitive forms (e.g., from full-resolution GPS coordinates to just a zip code). This modular architecture allows for the experimentation of various models for how apps consume sensitive data and how users can gain insight into their use.

Policy Managers regulate all access to dangerous permissions [8] on the system, as well as requests issued to the Private Data Service. In making these decisions, Policy Managers operate on run-time contextual factors such as the calling app, its visibility, and (if specified by the app developer) the purpose for the data request. Policy Managers also receive information about app metadata, declared permissions, and developer-set policies at install time. Developers and researchers can rapidly implement various permissions models as Policy Managers. Policy Managers are user-space Android apps, which eliminates the need to modify permissions logic within the Android platform itself. PE for Android allows users to install multiple Policy Managers and select the active one without needing to re-provision the device.

In lieu of existing permission-controlled APIs like LocationManager, apps targeting PE for Android can opt to send requests for sensitive data to the Private Data Service. Through this, apps obtain the information they need in a “least privileges” [9] manner, and do so without inadvertently extending overly broad permissions to bundled third-party services. The Private Data Service is the entry point for the data transformation and isolation techniques implemented by a µPAL. The Private Data Service and associated µPALs are a collection of trusted processes responsible for the direct access of sensitive resources (e.g., high-accuracy geolocation data) and the subsequent extraction of reduced-resolution information the app needs (e.g., a zip code for a weather app). This model eliminates the need for apps to have direct access to private data otherwise exposed by the stock Android API.

As Policy Managers and µPALs will have access to sensitive information, PE for Android places restrictions on the installation of these trusted components. Thus, developers must sign their Policy Manager and µPAL binaries with the PE for Android platform key in order to authorize its installation. The platform blocks the installation of improperly signed components. This mechanism allows for trusted third-party developers to contribute to the PE for Android ecosystem, while still enforcing a degree of review and curation on what Policy Managers and µPAL modules can do.

Permission model and policy enforcement

PE for Android invokes the current active Policy Manager upon app installation and at runtime as apps request sensitive data. The Policy Manager then returns a decision to allow or deny these requests. To aid in the Policy Manager's decision, permission checks in PE for Android may be tagged with a *purpose* [10] that describes how the data will be used. Existing research shows that purposes can be inferred in certain cases for apps targeting the stock Android API. [11]

Under Android's default permissions model, developers declare an app's required permissions in the *App Manifest* [12] file. PE for Android expands on this notion by allowing developers to specify the purpose of those permissions in an additional *App Policy* file. During app installation, the Policy Manager can use the app policy to decide whether to allow or block the installation. For example, the Policy Manager could show a dialog to the user displaying the app's requested permissions and their corresponding purposes. This gives an overview of how the app would interact with sensitive data, and let the user decide whether to continue with the installation.

Similar to Android's Runtime Permission feature [13], PE for Android forwards all runtime permissions checks to the Policy Manager, which implements the logic to grant or deny these requests. However, rather than relying on the requesting app to provide context to the user via a dialog, PE for Android provides the Policy Manager with contextual information about the permission check: a declared purpose (if supplied); the app component that issued the request; the top Activity visible when the request occurred; and a stack trace of all threads running in the requesting app. App developers can provide additional context via the declaration of purposes using PE for Android's APIs. For example, a maps app can set the purpose to "Navigation" before making a call to "requestLocationUpdates()". When a declared purpose is not available, a Policy Manager might attempt to infer it from the remaining contextual information. For example, if the stack trace shows that the request originated from a known 3rd party advertising library, the Policy Manager could inform the user that this request is likely meant for advertising.

Likewise, the Policy Manager controls access to sensitive data accessed via the Private Data Service. To do this, the Policy Manager receives the type of data being transformed (e.g. Location), and the purpose of the request (e.g. Weather). It also receives a human-readable description of the transformation that will occur. µPAL modules must provide this description for the user to understand what transformed data the app will receive. In the example of the zip code µPAL (described in a further section below), the description string reads "Returns the US zip code of the current location." This text can be displayed within a Policy Manager's UI to inform the user about the µPAL when configuring the policy.

Every request for sensitive data goes through the Policy Manager. In addition to deciding whether to grant or deny these requests, Policy Managers can also serve as a transparency tool for users. Policy Managers are in a position to log all requests and provide visualizations in its own UI accessible through the system settings. PE for Android also provides hooks into the Android SystemUI that enable additional interactions with users. The Policy Manager can generate notifications via a dedicated privacy channel for each app to alert the user about an app's privacy-related behavior. Additionally, the Policy Manager can post interactive tiles in the Android Quick Settings drop-down panel. [14] The icons, text, and behaviors of these toggles are configurable, and can provide a method for users to quickly make policy changes without needing to open the full Policy Manager UI.

Private data isolation

PE for Android supports a new way for apps to access sensitive data types such as location, contacts, and SMSs. The *Private Data Service* handles the retrieval and transformation of sensitive data on behalf of apps (subject to the permission model implemented in the active Policy Manager). Under this model, only this trusted architecture---not the requesting app---has direct access to the full scope of sensitive data available through the system API.

Our platform internally represents the various sensitive data types as *Item*-derived classes from the PrivacyStreams project. [15] These encapsulate a key-value mapping specific to each sensitive data type. Raw Items are not visible to the requesting app. Instead, they act as a convenient internal representation prior to the data undergoing a privacy transformation down to just what the app needs.

µPAL modules

Prior to delivery to the requesting app, sensitive data undergoes a transformation step to reduce its granularity to just what the app needs. Small user-space services called *µPAL modules* (read as "micro-PAL") perform these operations. As inputs, µPAL modules accept unfiltered data (from the Private Data Service's direct calls to privileged system API functions) and optional app-provided parameters. For example, a µPAL may receive the full contact list from the system and a given phone number from the requesting app. The µPAL can then return only the name of the corresponding contact with that number, if available. This allows apps to look up names for a known phone number without having full access to the contact list.

µPAL modules specify what sensitive data they intend to transform. This could be a particular data type like location, phone state, or contacts. Alternatively, a µPAL may declare "none" to indicate it will only operate on the input parameters (e.g., a µPAL module that only operates on app-provided parameters), or "any" to specify that it can transform all system-provided private data (e.g., an encrypting µPAL module).

Limitations

We acknowledge that PE for Android's current implementation has some practical limitations, and thus invite further investigation and development.

The µPAL paradigm limits the sensitive data that apps are able to access. That is, rather than calling system APIs directly, apps can invoke the Private Data Service. However, this also bypasses an existing mechanism used to provide transparency into what data apps can access. Users and app stores rely on declared permissions to determine what sensitive data apps may read. [16] Although Policy Managers can provide this transparency (i.e., by notifying the user when an app invokes the PAL), they currently can only do so for µPALs at run-time; users receive no information on an app's potential behavior regarding µPALs prior to installing it. One potential mitigation is to require app developers to specify µPAL usage within the app policy file that the Policy Manager can process at install time. Another potential mitigation would be to extend the permissions information specified in the app's manifest to provide this information to the Policy Manager.

A second limitation of PE for Android is that the current PAL implementation only exposes "read" operations on sensitive resources. For example, an app developer can opt to use the PAL paradigm to read the contact list without needing privileges that allow direct access to it in full. However, under the existing design, the same app developer would indeed need those privileges if the app needs to write to the contact list (or any other store of sensitive data, in fact). The PAL architecture does not currently offer a path through which apps can write data to otherwise permission-protected system resources. Requesting "write" permissions necessarily gives "read" privileges to access all the data presently stored in those resources. This warrants further design and development to safely support this feature.

PE for Android intercepts requests for sensitive data (fulfilled through either the PAL architecture or the stock system API) and invokes the Policy Manager, which decides if the request comports with policy. Policy Managers inspect every attempt to access sensitive data. Developers implementing these components must exercise extreme caution if relying on any user judgment for this purpose. Prior research has shown that prompting the user too frequently results in decision fatigue and habituation, [17] where users naively approve the requested privileges merely to dismiss nagging dialogs. This produces suboptimal privacy outcomes. Any prompts should be properly contextualized so users make informed decisions in the (ideally infrequent) cases that require their input. PE for Android's mechanism for purposes helps address this. However, the current implementation does not require Policy Managers to consider purposes at all. We encourage developers of Policy Managers to rely on heuristics that take this context into account and minimize the need for active user decision-making. [18]

Examples

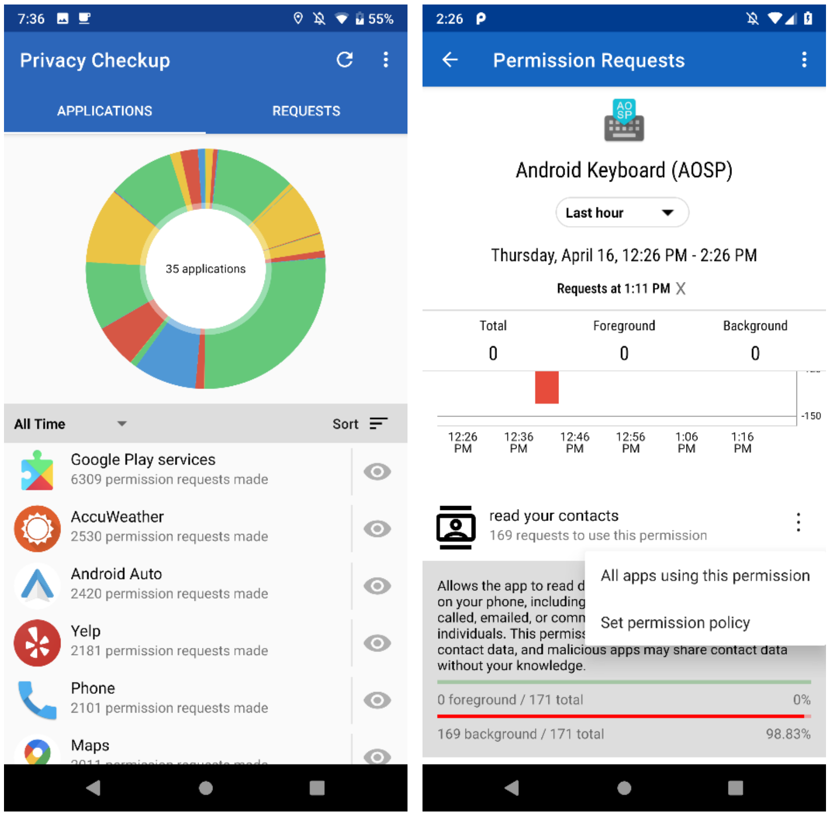
In this section, we discuss practical examples of Policy Managers, µPALs, and µPAL-enabled applications. We implemented these to illustrate practical use-cases for our platform's various capabilities: instrumenting the permission system, performing privacy transformations on sensitive data, and integrating with existing applications without reducing functionality.

Policy Managers

In PE for Android, Policy Managers have full visibility and control over all dangerous permission requests and Private Data Service calls. Our platform's API gives Policy Managers a wide variety of capabilities from that critical position: approve or deny access attempts; log requests and display to the user; and alert the user of real-time activity via the notification channel. A Policy Manager does not necessarily need to use all these capabilities. However, PE for Android offers these in order to support a variety of efforts to explore transparency and control mechanisms for users.

Privacy Checkup

A major potential use case for Policy Managers is in providing transparency into when and how apps request access to sensitive data. We offer the Privacy Checkup tool as a functional proof-of-concept for this application. The Privacy Checkup logs all dangerous permission requests and µPAL calls, then offers historical statistics to the user. This gives the users insight into when apps are accessing sensitive pieces of information, and particularly if that behavior occurred invisibly in the background. The Privacy Checkup continues to capture app activity when users are in other apps and even when the phone is idle with the screen off.



Google Play services
6309 permission requests made

AccuWeather
2530 permission requests made

Android Auto
2420 permission requests made

Yelp
2181 permission requests made

Phone
2101 permission requests made

Maps

Android Keyboard (AOSP)

Thursday, April 16, 12:26 PM - 2:26 PM

Requests at 1:11 PM X

read your contacts
169 requests to use this permission

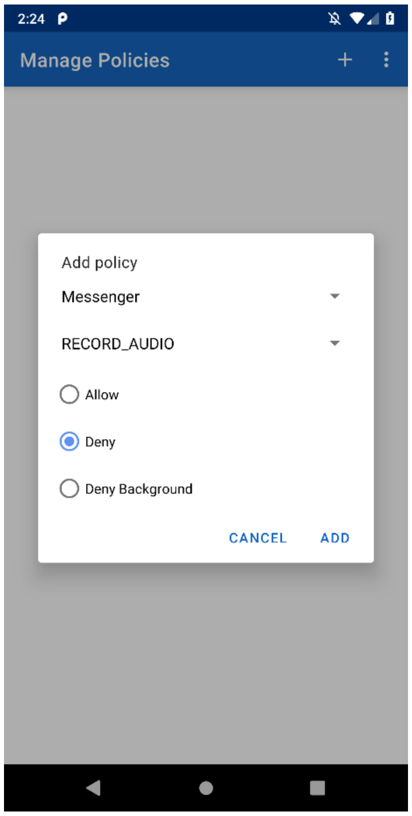
Allows the app to read d
on your phone, including
called, emailed, or comn
individuals. This permis
contact data, and malicious apps may share contact data without your knowledge.

0 foreground / 171 total 0%

169 background / 171 total 98.83%

Figure 2 Privacy Checkup offers users transparency to how installed apps are accessing private data.

This tool presents both raw access counts and minute-by-minute activity. A user gains insight into when apps are collecting sensitive data unexpectedly; for example, when a keyboard app is polling the contact list even when the user is doing something else (as shown in Figure 2). With this knowledge, users can use the built-in policy editor (shown below in Figure 3) to restrict when individual apps can receive particular permissions; in this example preventing the Facebook Messenger app from accessing the microphone.



Add policy

Messenger

RECORD_AUDIO

Allow [✓ Deny] Deny Background

Figure 3 Privacy Checkup built-in policy editor allows users to control what private data apps can access.

Source code available at:
https://github.com/twosixlabs/PE_for_Android_privacycheckup

Purposes Policy Manager

A second example Policy Manager was developed by Carnegie Melon University. The Purposes Policy Manager lets people view and set policies for individual apps as well as for all apps on the smartphone. When viewing an app, a person can see all of the permissions an app might use, with additional information about specific purposes also shown. These permissions are also organized into first-party access (where data is accessed by the app itself and possibly shared with the app developers) and third-party access (where data is accessed by third-party libraries and potentially sent to their servers). Users can set the policy for an entire permission or for just a specific permission-purpose pair, setting it to always allow, ask at runtime, or always deny. Users can also set similar policies for all apps on the smartphone, e.g. all apps using location for social media -> deny. More details and examples of Purposes Policy Manager are available at https://github.com/CMUChimpsLab/cmu_mobile_brandeis.

µPAL modules

We provide example µPAL modules to showcase the broad development norms and potential applicability of these components. Our general guideline is to envision µPALs as lightweight self-contained functions that have minimal state or external dependencies. They should reduce or obfuscate sensitive data, then return the result to the requesting app quickly enough that apps can safely treat µPAL calls as synchronous. However, this is not strictly enforced, and we designed the PAL architecture to tolerate µPAL designs beyond these recommended practices. Source code to these sample µPAL modules are provided as part of the PE for Android open source release.

Name ⟵⟶ phone number

System API data source: contact list
App-provided parameter (required): known phone number or known name

This µPAL module allows apps to retrieve the corresponding name or phone number given one of those pieces of information, and do so without exposing the whole contact list to the app. The calling app assembles a request for contact list data and includes a known number or name, which the µPAL module uses for this cross-referencing operation.

Source code available at:
https://github.com/twosixlabs/PE_for_Android_example_upals/blob/master/app/src/main/java/com/twosixlabs/exampleupals/NameToNumberPAL.java
https://github.com/twosixlabs/PE_for_Android_example_upals/blob/master/app/src/main/java/com/twosixlabs/exampleupals/NumberToNamePAL.java

Fine location → zip code

System API data source: geolocation functionality
App-provided parameter (optional): location provider

This µPAL module allows apps to receive rough (US zip-code level) location data without getting the user's exact location. The calling app issues a request for geolocation data and optionally specifies a location provider (e.g., satellites, cell towers, wi-fi routers, last known, or fused). The µPAL receives this data and performs a local lookup to report the US zip code for the current location, if it exists.

Source code available at:
https://github.com/twosixlabs/PE_for_Android_example_upals/blob/master/app/src/main/java/com/twosixlabs/exampleupals/ZipcodeMicroPAL.java

Calendar Events → Busy/Free Time Bitmap

System API data source: calendar
App-provided parameter (required): start/end dates, time resolution (Day, hour, 15 minutes)

This µPAL module allows apps to receive a bitmap showing times when the user is busy or free according to events on their calendar. The calling app puts together a request for the bitmap and specifies the start and end times, as well as the resolution of timesteps. The Private Data Service delivers a list of calendar events scheduled within the time window to the µPAL. The µPAL iterates through the list of events and generates an array of timeslots with a value of 0 if the user is free or 1 if they are busy.

Source code available at:
https://github.com/twosixlabs/PE_for_Android_example_upals/blob/master/app/src/main/java/com/twosixlabs/exampleupals/AvailabilityBitmapMicroPAL.java

Private Data → SGX Encrypted Data Capsule

System API data source: geolocation, contact lists, and text messages
App-provided parameter (required): DuetSGX address and port, maximum allowable value for differential privacy parameter epsilon

Developed by the University of Vermont and the Brandeis Helio team, the DuetSGX µPAL module submits data to a DuetSGX server, where it can be used to answer differentially private queries specified in the Duet [19] programming language. DuetSGX is designed so that submitted data can **never** be viewed by **any** party - the only possible use of the data is for answering differentially private queries.

On the DuetSGX server, submitted data is stored encrypted, with the private key available only to the DuetSGX software running in a protected SGX enclave. When the DuetSGX process (and its enclave) exit, the key is discarded and the submitted data is effectively lost. The server administrator specifies a total privacy budget for differentially private queries when starting the DuetSGX server; each submitted query subtracts from this budget, and the DuetSGX process ends (destroying the data) when the budget reaches zero.

Source code available at:
https://github.com/uvm-plaid/duet-sgx

Apps

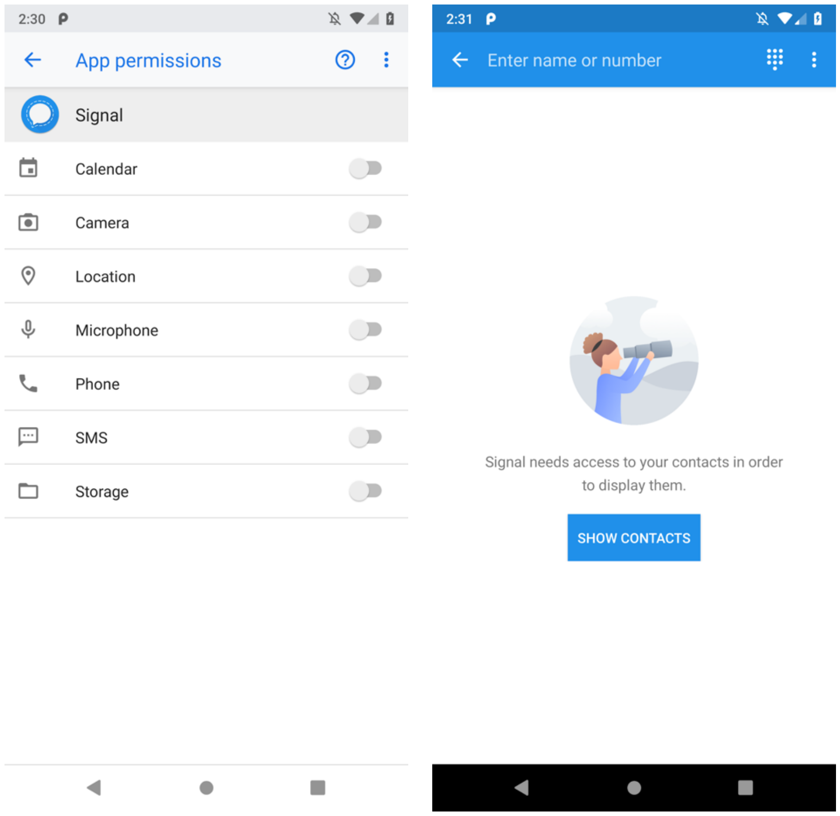
We demonstrate how µPAL concepts apply to real-world use cases by modifying existing publicly-available open source apps. We used the PAL paradigm to eliminate a permission from those apps---which would otherwise have granted them direct access to a sensitive resource in full---and used µPAL calls to achieve similar functionality with reduced privileges.

Signal Private Messenger [20]

The Signal Private Messenger is an end-to-end encrypted instant messaging service that operates on a variety of platforms including Android. As a communications app, Signal reads the user's contact list to help connect them to the people they know. As a result, the stock Android implementation of permissions and contact list APIs allows Signal to access the user's entire contact list.

We sought to eliminate the permissions *GET_ACCOUNTS*, *READ_CONTACTS*, and *WRITE_CONTACTS* from Signal's manifest. These permissions expose sensitive data that we deemed overly broad and unnecessary for a privacy oriented messaging app. To that end, we used the PAL architecture and Android's built-in trusted contacts picker to give Signal the contact information it needs, but only for contacts the user selects.

We first removed those permissions from the manifest and discovered that Signal handles permission denials gracefully. When Signal does not have permission to access the contact list, its built-in contact picker guides the user to the settings that would allow the user to grant it. In the official version of Signal, the “Show Contacts” button displays those settings as shown in the rightmost screenshot in Figure 4 below.



Signal

Calendar	off
Camera	off
Location	off
Microphone	off
Phone	off
SMS	off
Storage	off

Signal needs access to your contacts in order to display them.

Figure 4 Screenshots of our modified Signal app. The left screenshot shows Signal without contacts or account permissions. The right shows the modified app gracefully catching the denial and guiding the user to grant the permission.

We redirected the “Show Contacts” button to instead issue an Intent to call Android’s built-in contact picker. [21] The contact picker is a system-provided Activity that operates outside of Signal’s process space. This means that only the picker can read the user’s full contact list, and Signal only receives what the user selects. Upon the user’s selection, the picker sends one contact to Signal, whose phone number we place in the search box (as if the user had manually typed it in) as shown in Figure 5.

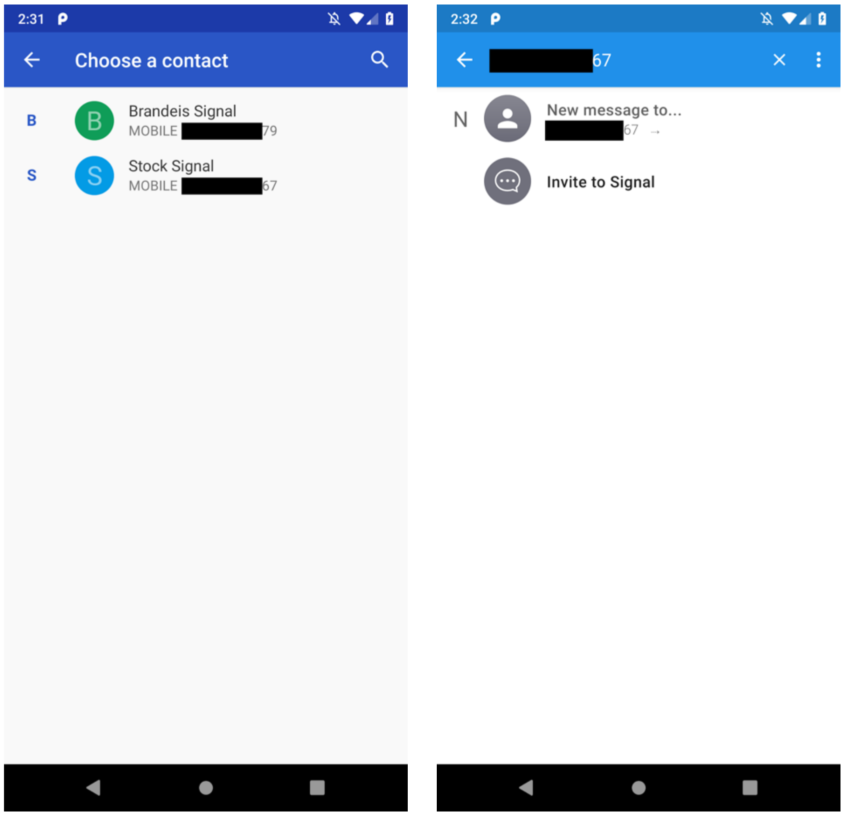


Figure 5 Screenshots of our modified Signal app. The left screenshot shows the system-provided contact picker. The right shows that after user selection, the phone number for 'Stock Signal' goes into the contact search box.

Users expect their contacts' names to appear in the UI, as names are more usable to humans than raw phone numbers. The contact picker only sends a phone number to Signal. In order to get the contact's name, we modified Signal to use the PAL paradigm: Signal makes a request for a "phone-number-to-name" µPAL module to receive the full contact list and an app-provided phone number (i.e., what the user had previously selected), then return the name associated with that phone number if it exists. Our modifications save the name and phone number in Signal's internal database, which Signal subsequently uses to populate the messaging UI, as shown below in Figure 6. Note that if the user received a message from a user who did not associate their name with Signal, this same µPAL module could be used to look up the sender's name based on their phone number.

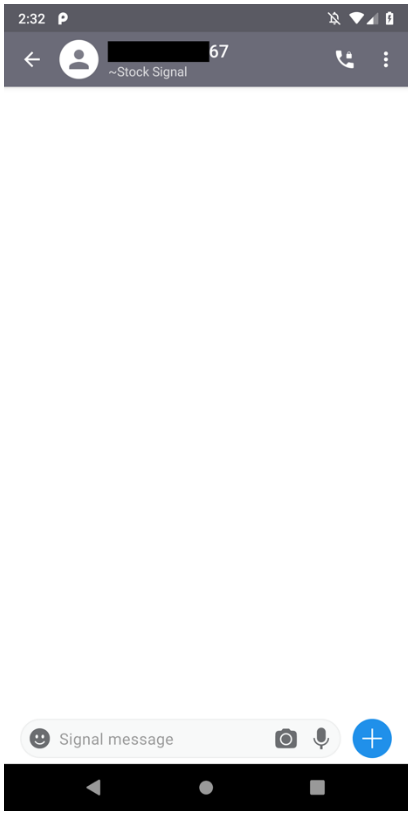


Figure 6 A μPAL module returns the name ("Stock Signal") associated with the selected number.

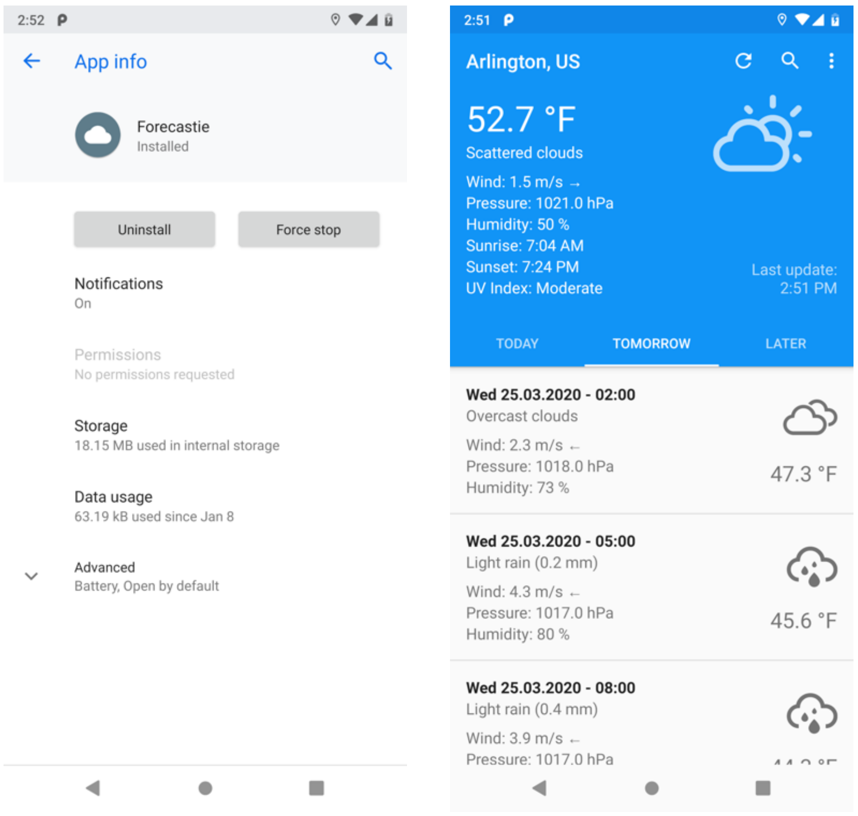
We implemented these changes and verified that our modified version of Signal continues to function properly with the Signal network. We successfully sent and received messages between our modified Signal and peers running stock versions of the Signal app on Android, iOS, and Linux.

Source code available at:
https://github.com/twosixlabs/PE_for_Android_example_signal

Forecastie [22]

Weather apps have been implicated in the unexpected collection and use of consumers' location data. [23] Most request the *ACCESS_FINE_LOCATION* permission, which gives apps access to the device's true location within a few meters. Even with the less-accurate *ACCESS_COARSE_LOCATION,* apps can still get the location within approximately 100 meters. Location-enabled apps have access to much higher resolution location information than what's necessary to provide a weather forecast. We sought to alleviate this problem using the PAL architecture.

Forcastie is an open-source weather app for Android that queries forecast information using public OpenWeatherMap APIs. [24] In its normal implementation, Forcastie requests the user's high accuracy location, then queries OpenWeatherMap using the latitude and longitude of the device. This exposes the user's location to both the Forcastie app and the backend weather service. To reduce the fidelity of the user's location information that is exposed, we modified Forcastie to get forecast information using only the zip code of the user's current location. The changes required for this were minimal. First, we replaced the code that requested Location (through the standard LocationManager APIs) with a call to our "zip-code" µPAL module. Then, we changed the call to OpenWeatherMaps from an API that uses latitude and longitude to one that uses zip code.


2:52

App info

Forecastie
Installed

Uninstall | Force stop

Notifications
On

Permissions
No permissions requested

Storage
18.15 MB used in internal storage

Data usage
63.19 kB used since Jan 8

Advanced
Battery, Open by default

2:51

Arlington, US

52.7 °F
Scattered clouds
Wind: 1.5 m/s →
Pressure: 1021.0 hPa
Humidity: 50 %
Sunrise: 7:04 AM
Sunset: 7:24 PM
UV Index: Moderate

Last update:
2:51 PM

TODAY | TOMORROW | LATER

Wed 25.03.2020 - 02:00
Overcast clouds
Wind: 2.3 m/s ←
Pressure: 1018.0 hPa
Humidity: 73 %
47.3 °F

Wed 25.03.2020 - 05:00
Light rain (0.2 mm)
Wind: 4.3 m/s ←
Pressure: 1017.0 hPa
Humidity: 80 %
45.6 °F

Wed 25.03.2020 - 08:00
Light rain (0.4 mm)
Wind: 3.9 m/s ←
Pressure: 1017.0 hPa

Figure 7 Our modified Forcastie app. The left side demonstrates that Forecastie does not request any permissions. The right shows the app presenting the weather in Arlington, VA based on zip code 22201.

As shown in Figure 7, with these small modifications, we were able to remove the Location permission from Forcastie, and greatly reduce the fidelity of location information that both the app and the backend weather service receive. Additionally, because the zip code µPAL uses a local database to perform the lookup converting a precise location to a zip code, the specific latitude longitude coordinates of the user are never sent off the device.

Source code available at:
https://github.com/twosixlabs/PE_for_Android_example_forecastie

Conclusion

We present PE for Android as a platform to support various privacy-focused efforts on mobile devices. Our work enables developers to deploy novel permission systems and privacy-preserving technologies without having to modify the OS itself. Through a variety of examples, we demonstrate its practical potential as it applies to user empowerment and reducing data exposure in actual apps. We invite developers and researchers alike to use PE for Android's broad applicability to expedite their work and improve privacy outcomes for users.

Acknowledgements

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) through a subcontract agreement with Raytheon BBN Technologies Corp. under Prime Contract No. FA8750-16-C-0006 with the Air Force Research Laboratory (AFRL).

The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Bibliography

[1] "Smartphone users by country worldwide 2019 - Statista.," [Online]. Available: https://www.statista.com/statistics/748053/worldwide-top-countries-smartphone-users/. [Accessed 17 April 2020].

[2] "Control your app permissions on Android 6.0 and up," [Online]. Available: https://support.google.com/googleplay/answer/6270602?hl=en. [Accessed 17 April 2020].

[3] J. Reardon, A. Feal, P. Wijesekera, A. E. Bar, N. Vallina-Rodriguez and S. Egelman, "50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System," [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/reardon. [Accessed 17 April 2020].

[4] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner and K. Beznosov, "Android Permissions Remystified: A Field Study on Contextual Integrity," [Online]. Available: https://www.usenix.org/node/190983. [Accessed 17 April 2020].

[5] I. Reyes, P. Wijesekera, J. Reardon, A. E. Bar On, A. Razaghpanah, N. Vallina-Rodriguez and S. Egelman, "“Won't Somebody Think of the Children?” Examining COPPA Compliance at Scale," [Online]. Available: https://petsymposium.org/2018/files/papers/issue3/popets-2018-0021.pdf. [Accessed 17 April 2020].

[6] "PE for Android," [Online]. Available: https://github.com/orgs/twosixlabs/teams/pe-for-android/repositories. [Accessed 17 April 2020].

[7] "Android Open Source Project," [Online]. Available: https://source.android.com/. [Accessed 17 April 2020].

[8] "Permissions overview," [Online]. Available: https://developer.android.com/guide/topics/permissions/overview. [Accessed 17 April 2020].

[9] "Least Privilege | CISA - US-CERT," [Online]. Available: https://www.us-cert.gov/bsi/articles/knowledge/principles/least-privilege. [Accessed 17 April 2020].

[10] Y. Li, F. Chen, T. J.-J. Li, Y. Guo, G. Huang, M. Fredrikson, Y. Agarwal and J. I. Hong, "PrivacyStreams: Enabling Transparency in Personal Data Processing for Mobile Apps," [Online]. Available: https://dl.acm.org/doi/10.1145/3130941. [Accessed 17 April 2020].

[11] H. Jin, M. Liu, K. Dodhia, Y. Li, G. Srivastava, M. Fredrikson, Y. Agarwal and J. I. Hong, "Why Are They Collecting My Data?: Inferring the Purposes of Network Traffic in Mobile Apps," [Online]. Available: https://dl.acm.org/doi/10.1145/3287051. [Accessed 17 April 2020].

[12] "App Manifest Overview," [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro. [Accessed 17 April 2020].

[13] "Request App Permissions," [Online]. Available: https://developer.android.com/training/permissions/requesting. [Accessed 17 April 2020].

[14] "android.service.quicksettings," [Online]. Available: https://developer.android.com/reference/android/service/quicksettings/package-summary. [Accessed 17 April 2020].

[15] "PrivacyStreams," [Online]. Available: https://github.com/PrivacyStreams/PrivacyStreams. [Accessed 17 April 2020].

[16] "Permissions | Privacy, Security, and Deception - Google Play," [Online]. Available: https://play.google.com/about/privacy-security-deception/permissions/. [Accessed 17 April 2020].

[17] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe and D. Wagner, "How to Ask for Permission," [Online]. Available: https://www.usenix.org/conference/hotsec12/workshop-program/presentation/felt. [Accessed 17 April 2020].

[18] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner and K. Beznosov, "The Feasibility of Dynamically Granted Permissions: Aligning Mobile Privacy with User Preferences," [Online]. Available: https://www.ieee-security.org/TC/SP2017/papers/278.pdf. [Accessed 17 April 2020].

[19] "DuetSGX GitHub," [Online]. Available: https://plaid.w3.uvm.edu/duet-sgx. [Accessed 17 April 2020].

[20] "Signal Private Messenger," [Online]. Available: https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms&hl=en_us. [Accessed 17 April 2020].

[21] "Pick contact directly from contact picker intent," [Online]. Available: https://stackoverflow.com/questions/41327416/pick-contact-directly-from-contact-picker-intent. [Accessed 17 April 2020].

[22] "Forecastie - Weather app," [Online]. Available: https://play.google.com/store/apps/details?id=com.casticalabs.forecastie&hl=en_us. [Accessed 17 April 2020].

[23] "LA Sues Over Weather Channel App's 'Misleading' Data Collection," [Online]. Available: https://www.usnews.com/news/national-news/articles/2019-01-04/los-angeles-sues-over-weather-channel-apps-misleading-data-collection. [Accessed 17 April 2020].

[24] "Weather API - OpenWeatherMap," [Online]. Available: https://openweathermap.org/api. [Accessed 17 April 2020].