Context-Specific Access Control: Conforming Permissions With User Expectations

Amir Rahmati
University of Michigan
rahmati@umich.edu

Harsha V. Madhyastha
University of Michigan
harshavm@umich.edu

ABSTRACT

Current mobile platforms take an all-or-nothing approach to assigning permissions to applications. Once a user grants an application permission to access a particular resource, the application can use that permission whenever it executes thereafter. This enables an application to access privacy sensitive resources even when they are not needed for it to perform its expected functions.

In this paper, we introduce "Context-Specific Access Control" (CSAC) as a design approach towards enforcing the principle of least privilege. CSAC's goal is to enable a user to ensure that, at any point in time, an application has access to those resources which she expects are needed by the application component with which she is currently interacting. We study 100 popular applications from Google Play store and find that existing applications are amenable to CSAC as most applications' use of privacy sensitive resources is limited to a small number of contexts. Furthermore, via dynamic analysis of the 100 applications and a small-scale user study, we find that CSAC does not prohibitively increase the number of access control decisions that users need to make.

1 Introduction

Applications running on mobile devices such as smartphones, wearables, and tablets can access private information from a range of sensors (e.g., GPS, camera, microphone) and from several sources of user-generated data (e.g., emails, photos, call history). However, not all applications need access to all sources of private information. Therefore, operating systems (OSes) for mobile platforms enable users to control an application's access to these resources.

Recognizing the need to limit the number of access control decisions that a user needs to make, so as to prevent decision fatigue [9], mobile OSes do not notify the user every time an application requests access to a resource. Instead, for every resource, OSes require the user to decide only once per application as to whether to grant the application access to that resource. In Android, the user makes this decision at application install time, and in iOS, the user decides whether to grant an application access to a resource when the application attempts to access the resource for the first time.

SPSM'15, October 12 2015, Denver, CO, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3819-6/15/10.$15.00
DOI: http://dx.doi.org/10.1145/2808117.2808121

Permission	Use
Camera	Take picture of individuals to search for a match in database
Location	Record location of suspect sighting
Photos	Look up individuals from previously taken photos
Device identifier	Associate purchases with user ID

Table 1: **JailBase's need for various permissions.**

We observe that this current status quo of access control on mobile platforms—once an application is granted access to a resource, it has access to it forever thereafter—violates the principle of least privilege. Even when applications have valid reasons for using privacy sensitive resources, they seldom need to have constant access to it. Since much of the private data available on mobile devices change over time (e.g., location, video, stored photos), it is necessary to not only limit *which* resources an application can access, but also *when* it has access to those resources.

Mobile environments are especially well-suited for implementing finer-grained access control. Unlike traditional desktop environments, where multiple applications are concurrently active and in view, user interactions in mobile environments typically focus on a single active application and interaction with background applications are limited to notifications and alerts. Furthermore, each mobile application is intrinsically divided into isolated components using *Activity* in Android and *UIViewController* in iOS. Each application can have at most one Activity/UIVuewController active at any given time.

As a case study, consider the JailBase app. JailBase, one of the top news applications in Google Play store, allows users to search through public arrest records. To implement its functionalities, JailBase requires use of camera and access to location, photos, and device identifier. Table 1 describes how each of these permissions are legitimately used by the application.

While a user has a legitimate reason for allowing JailBase to access each of these resources, the all-or-nothing approach of current permission systems provides no meaningful privacy assurances if the application either mistakenly or intentionally misuses its permissions.[1] For example, while the user is searching for information, the application can capture video without his knowledge; or, it can continuously track the user's location and transmit data to a server

[1]We are using JailBase as a potential example and are not implying that it is misusing user data in practice.

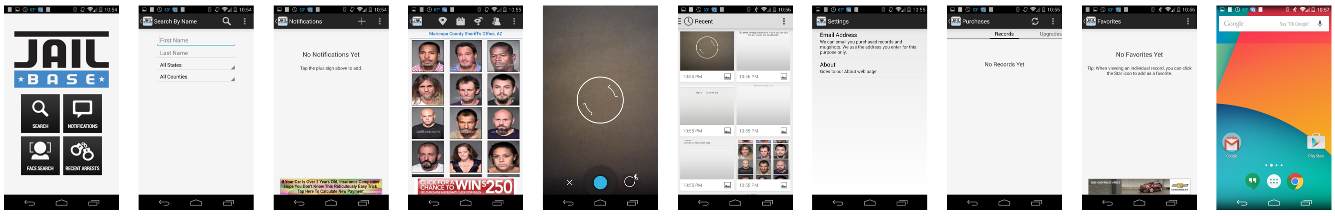


Figure 1: **Different contexts of the JailBase application.**

Context	Required Permissions
Home	None
Search	None
Notifications	None
Arrests	None
Face Recognition	Camera, Location
Gallery	Photos
Settings	None
Purchases	Device ID
Favorite	None
Background	None

Table 2: **Minimal permission set required by JailBase in different contexts.**

even when the application is in the background. In general, though a user may grant an app access to a specific resource since that permission is necessary for the app to fulfill one of the functions that it offers, the app can potentially also use this permission when the user is interacting with the app for other purposes. Instances of permission overuse has been reported in several popular Android applications such as WhatsApp, Shazam, and TuneIn Radio [16].

We propose "Context-Specific Access Control" (CSAC) to remedy this situation. We observe that a user's interaction with an application can be categorized to a limited number of contexts with different permission requirements. Therefore, by separating an application's states into a few meaningful contexts, CSAC can ensure that an application is limited to its minimum set of required permissions at any point of time, while requiring the user to only decide on the application's access to a resource once per context. Looking back at our example application, the state space of the JailBase application can be divided into ten different contexts described in Table 2, seven of which do not require access to any privacy sensitive resource. Figure 1 provides a snapshot of each of these contexts.

To implement CSAC on the Android platform, we propose dividing application contexts based on Activities. We evaluate the feasibility of this approach by examining the top 100 free applications in Google Play store. In our evaluation, we track the use of various permissions across different Activities as we explore the applications. Our results suggest that accesses to privacy-sensitive information are limited to less than 30% of Activities and most applications do not need constant access to these data. Moreover, our results suggest that CSAC significantly decreases access of applications to privacy-sensitive data while not increasing the decision overhead for most apps. On applications for which CSAC causes an increase in the number of access control decisions, its overhead is on average less than 4 decisions per application.

In summary, our contributions are three-fold:

- We introduce the notion of Context Specific Access Control (CSAC) and discuss its benefits and shortcomings.
- We demonstrate the feasibility of CSAC on the Android platform by looking at 100 popular Android applications' usage of permissions across different Activities.
- We evaluate decision overhead imposed on users when using CSAC by conducting dynamic exploration of 100 popular Android applications and a small scale user study.

2 Background

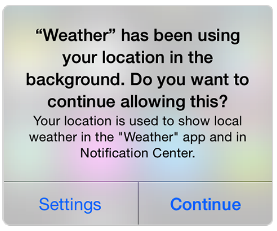
Before discussing CSAC, we provide a brief overview on Android and how popular mobile OSes currently handle access control.

Components of an Android app: Android applications are written in Java. An application is generally delivered in an APK (Android Package) containing compiled code, data, resources, and a manifest. An application consists of four components: Activities, Services, Content Providers, and Broadcast Receivers. Activities are used to implement various user interfaces while Services run in the background to perform long-running operations or to perform work for remote processes. Content Providers and Broadcast Receivers are respectively responsible for managing application data and respond to system-wide broadcasts. According to the Android developer's guide [1], an Activity should represent "a single screen with a user interface."

Access control in current mobile OSes: Both iOS and Android take an all-or-nothing approach to permission assignment. In iOS, a user is prompted upon an application's first use of a resource.[2] The user has the option to deny the permission and continue using the application with potentially limited functionality, although some applications may block their further use until a permission has been granted to them. When permission to a resource is granted to an application, it can use it in the future without prompting the user. While this approach allows a user to selectively block an application from using certain privacy-sensitive resources, it can be employed only if the application does not need that permission at *any* point of time. If the application is legitimately granted a permission for a particular task, it can use it later regardless of the user's desire or expectations.

Starting from iOS version 8, the user is prompted when the application uses the location data in the background and can selectively choose to share location with an application only when it is in the foreground. Figure 2 shows a sample of this alert box. While this binary context-awareness provides an improvement over the previous all-or-nothing approach, it still only covers one of the system resources and is not fine-grained enough to cover all misuse scenarios. An example of this would be a shopping application using GPS coordinates to provide nearest store locations, but additionally tracking a user's movements while the user is creating a wish list.

[2]An application's access control settings can later be changed in the system privacy settings.



"Weather" has been using your location in the background. Do you want to continue allowing this?

Your location is used to show local weather in the "Weather" app and in Notification Center.

Settings | Continue

Figure 2: **iOS background location usage alert. The binary context awareness protects user's location when application is not in-use.**

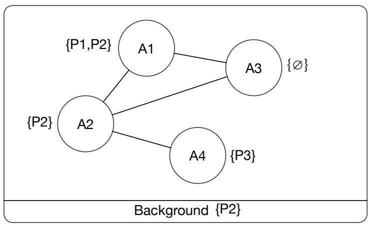


Figure 3: **Example of permission enforcement with CSAC. Each circle represents an Activity, with an associated permission set. Edges represent inter-Activity transitions. CSAC divides the application into 5 contexts (4 Activities + Background) and asks users to make a one-time decision upon first use of each permission in a context.**

In contrast, in Android, every application includes an "AndroidManifest.xml" that defines all of its required permissions. During install time, the package installer prompts the user to grant permissions to the requested resources. An application can only be installed if these permissions are granted. No further clarifications are sought from the user while an application is running. So an application is either granted a particular permission when installed, and can use that feature as desired, or the permission is not granted and any attempt to use the feature fails without prompting the user (even when the app does need to use the permission in order to fulfill the functionality expected by the user) [1].

More recently, Google has announced that starting with Android M, they will not only adopt iOS's "prompt on first use" policy but also allow users to later modify the permissions granted to an application [3].

3 Context-Specific Access Control

Threat model: CSAC's goal is to enable fine-grained control over permission access by differentiating between the various contexts in an application's execution. Using this approach, a user can select contexts within which an application can access any particular resource and stop applications from broad usage of resources. An example of this would be an application that only requires access to the user's location in one of its contexts but continuously monitors and records the user's location even when in other contexts. This scenario could be the consequence of either deliberate (e.g. an application that has been maliciously modified) or inadvertent (e.g. a bug in the application such as Apple hotspot database cache [2]) overuse by an application of permissions granted to it. In contrast to traditional access control systems, which make the user unable to protect himself against such privacy leaks, CSAC raises the bar for permission misuse and limits usage of system resources by applications to the limited contexts that match user expectations.

Envisioned implementation: With CSAC, we envision that the user will be prompted to make one-time access control decisions when a permission is used for the first time in a context. This decision can be recorded and reused later. Figure 3 presents an example of how this will work. Each application consists of a set of Activities $\{A1, A2, \ldots\}$, with each Activity accessing a set of permissions during its execution. Instead of allowing an application all-or-nothing access to privacy-sensitive resources, CSAC can associate each access control decision to the Activity that is active when the request for access is made. If none of the application's Activities are active, the application is in the background and the access control decisions made for it will be associated with its background context. The request to access a resource does not necessarily have to be made by an Activity and can be made by Services that are running simultaneously. Permission to access a resource in a Service can be evaluated based on the Activity that is active at the time of the request, or—to support asynchronous tasks—the Activity the Service has initiated in.

To augment CSAC to Android's permission system, mechanisms used to validate system calls (which are generally responsible for access control enforcement [8]) have to be modified to take into account the current active Activity when making access control decisions. In our example, using CSAC will help enforce the principle of least privilege by limiting the application to 33% ($\frac{5}{15}$) of the (Activity, permission) pairs that are possible with the all-or-nothing approach currently in use in Android and iOS.

Starting from Android 3.0, Google has introduced the notion of "Fragments" to enable better modularization of code, more sophisticated user interfaces, and GUI scaling for applications that target different screen sizes. A Fragment, which is always embedded in an Activity and represents a portion of the user interface, can be either combined with other Fragments in a single Activity or be reused across Activities. CSAC does not limit the use of Fragments or prohibit code reuse but ensures that permissions available to a Fragment depends on the Activity, as part of which it is being executed. Thus, a Fragment can potentially have permission to access a resource in Activity $A1$ but is not allowed to do so in $A2$.

Although designing an appropriate user interface to assign and later change permissions for each Activity remains a challenge, migration of Android OS to "prompt on first use" access control policy opens up the path for implementation of CSAC with minimum modifications to the platform. Instead of recording access control decisions per application, these decisions need to be recorded and used based on application and Activity pairings.

Benefits: In general, basing access to privacy-sensitive data on the user's context provides several unique benefits, compared to how access control is enforced on existing OSes.

- CSAC conforms the permissions used by an application with users' expectations: Prior work [12, 13] has highlighted the benefits of including users' expectation in configuring access control. Using CSAC, users can control the resources an application can access in any particular context and prevent applications from misusing broadly assigned permissions.

- CSAC allows the OS to define different levels of access for each resource: For example, a restaurant finding application can be given access to coarse-grained location when identifying relevant ads and to fine-grained location when listing nearby restaurants.

- CSAC can provide high privacy assurances to the user without compromising application functionality: As long as the user correctly grants an application the permissions it needs in each context, we believe that the application can operate without significant performance overhead or reduction in its functionality. Once the user grants a permission in a particular context, the OS can remember that decision and reuse it without prompting the user again when the application enters the same context.
- CSAC does not break the current application model or limit developers: CSAC does not force any changes on applications or make legacy applications unusable. Furthermore, unlike systems such as ACG [15], CSAC does not force application developers into using any specific GUI components.

Limitations: While providing many improvements to users' privacy, CSAC is by no means without limitations. Most noticeably, CSAC imposes a higher decision overhead on users as they potentially have to make access control decisions to a particular resource multiple times in an application. Furthermore, although CSAC moves us toward realizing the principle of least privilege, there are cases where it can still be coarse-grained, e.g., an application uses location information for nearby restaurants but continues to track the user when he is inspecting the results. CSAC also does not provide any benefits if an application is condensed into one or very few Activities (which we show later is rare among popular applications) or if an application chooses to terminal itself when a particular permission is not granted. Privacy-sensitive data, in particular data that is static (e.g. IMEI) or semi-static (e.g. contact list), can also be stored and reused by the application in other contexts. Mitigating information leakage would require data-flow analysis techniques such as taint tracking that impose a large overhead. In Section 5, we discuss how CSAC can be combined with data-flow analysis techniques to decrease their overhead.

4 Feasibility Study

While CSAC offers the potential for various privacy benefits, as described above, the feasibility of using CSAC and its utility in practice is contingent on several criteria:

1. Applications need to have been implemented such that different functionalities offered by an application are separated out into separate Activities; benefits of CSAC are only attainable if applications are appropriately modular.
2. The set of permissions used by an application should differ across Activities; if an application uses the same permissions in all of its Activities, then the current model of "grant permission upon first use" suffices.
3. The number of additional access control decisions that users have to make should not significantly increase. A disproportionate increase in the number of decisions can cause decision fatigue and result in users making incorrect (and potentially harmful) decisions.

In this section, we evaluate all of these criteria on 100 popular applications from Google Play store. We downloaded these applications from the "Top Free in Android Apps" list. After manually discarding applications that required an account to perform their main functionality (e.g., banking and mail apps), we conducted a dynamic analysis of these applications using the A^3E dynamic exploration tool [5]. We modify Android 4.4 to extract information on when Activities start and stop executing, and on the use of privacy-sensitive resources. We use the PhoneLab [14] Android ROM as our base code as it provides various logging instrumentations that assist our evaluation. We collect logs of these interactions that we later use in our analysis.

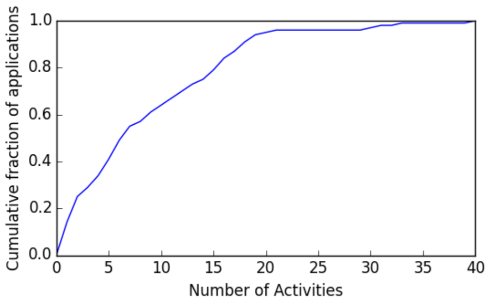


Figure 4: **CDF of number of Activities per application.**

4.1 How modular are Android apps?

In order for CSAC to enforce different permissions for an application in its different Activities, the application needs to be divided into multiple Activities that can be used to differentiate between contexts. While this means that CSAC does not provide much benefit for applications that make extensive use of native code, where the application is condensed mostly into a single Activity, Figure 4 shows that this is not the common case. From the figure, which plots the CDF of Activity count in the 100 popular applications used in our dynamic analysis, we see that we encounter 9 Activities on average per application. This shows that most applications are sufficiently modular for CSAC to be useful.

The coverage of our dynamic exploration tool is not perfect; for most applications, we visit a subset of the application's Activities. There are different factors that cause A^3E's coverage to be incomplete, some of which include use of native code and complex gestures, social network integration, and requiring an account or purchases to enable features [5]. However, previous work has shown that, on average, A^3E covers twice the number of Activities as compared to human subjects [5].

4.2 How does permission usage vary across Activities?

CSAC works by assigning permissions on an Activity basis. In the example JailBase application discussed in Section 1, we described how most of the Activities do not require access to privacy sensitive resources. CSAC benefits from this usage pattern as it can limit the application's access to a specific resource to a limited number of Activities.

Privacy-sensitive resources can be generally broken down into four categories: (1) Static values such as International Mobile Station Equipment Identity (IMEI); (2) semi-static data such as photos, messages, and contacts list; (3) communication channels such as Bluetooth and NFC; and (4) sensor data such as GPS. In our evaluation of top 100 applications, we focus on a selected number of privacy-sensitive resources

Resource	ID	Camera	Location	Bluetooth	Photos
Usage (/100)	89	38	55	9	23

Table 3: **For each tracked permission, number of apps in the set of 100 evaluated applications that use it.**

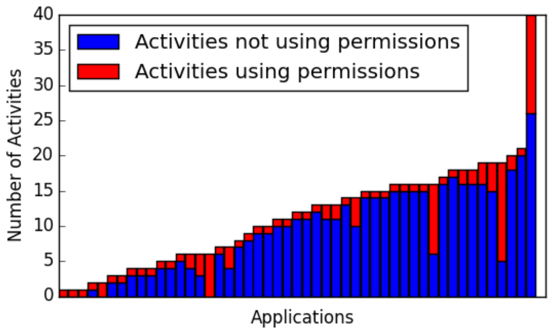


Figure 5: **Use of permissions in applications' Activities. Most Activities do not require access to any of the tracked privacy-sensitive resources.**

that are most widely used by the applications. Namely, we evaluate applications' use of camera, photos, location, device identifiers and Bluetooth. These resources account for more than 85% of privacy sensitive resources that are requested in these applications. Table 3 presents the usage of these permissions by 100 popular applications used in our evaluation according to their manifest.

In our analysis of dynamic exploration data, we observed that various device identifiers were frequently fetched across different Activities in many of the applications. Because these values are static and can be recorded and reused, allowing one time access to them in an application can be considered equivalent to giving the application lifetime access to it in all of its Activities. Hence, we only consider access to device identifiers once in our evaluation of permission usage across different Activities.

Figure 5 presents the number of Activities that require one or more of the permissions compared to the total number of Activities. Our results show that more than 70% of Activities in applications do not use any of the tracked privacy sensitive resources that was requested by the application. This result shows that a realization of CSAC that considers different Activities in an application as separate contexts can limit an application's access to privacy-sensitive resources to a (typically small) subset of the application's execution.

4.3 What is the decision overhead of CSAC?

Lastly, we consider the concern associated with CSAC that it can increase the decision load that is imposed on the user as the result of finer-grained access control. Both Android and iOS platforms assign each permission once per application. This approach results in low decision overhead for the user but enables an application to access privacy-sensitive resource at all times. Using CSAC, the decisions to allow access to a resource are assigned on a per-Activity basis. This can result in as many as $(\#Permissions \times \#Activities)$ decisions to be made by the user.

Using the dynamic analysis data, we calculate the number of decisions required to be made by the user when CSAC is in use and compare it against scenarios in which access control

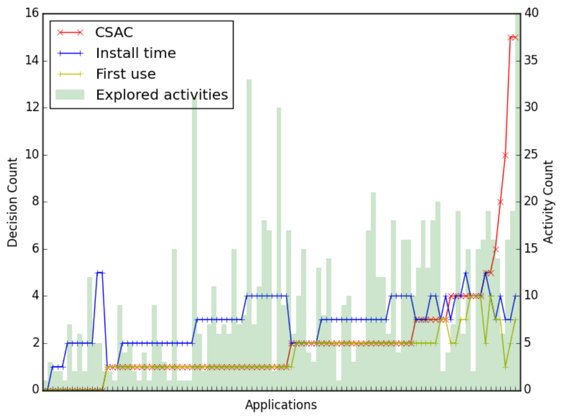


Figure 6: **Comparison of the number of decisions user has to make for each application under "grant permission during install time", "grant permission on first use", and CSAC model. CSAC does not increase the number of required decisions for most applications.**

decisions are made at install time or on first use. Similar to Section 4.2, we count decisions regarding device identifiers only once as they are static values which can be saved and reused by applications.

Figure 6 presents the results of our evaluation. On the one hand, for most applications, the number of decisions necessary with CSAC matches the number of decisions that had to be made in other approaches, so CSAC imposes no decision overhead on the user. There is also no direct relation between the number of decisions that needs to be made and the total number of Activities an application has.

On the other hand, even for applications that require more decisions when using CSAC, we expect the actual number of decisions required to be made by the user to be less than those presented in Figure 6. This is because previous work [5] has shown that the amount of coverage provided by the A^3E tool is on average twice those of normal users.

To test this hypothesis, we conducted a 5-user study on the five applications requiring the most number of decisions. Our five users were all CS-major graduate students with prior experience in using Android OS. Each user was given a description of each application's function and no time limit was imposed on the user during experiments. We asked the participants to explore the applications and use them as they normally would and tracked activation of Activities and usage of privacy-sensitive data across the applications. Table 4 presents the results of this experiment. Even when considering the union of users' exploration of the applications, the number of decisions that users had to make dropped compared to our experimental results as typical users explored fewer Activities and features.

5 Related Work

In this section, we discuss some of the other notable access control augmentations and examine how they compare and combine with CSAC.

Data-flow analysis: There is a large body of research on approaches and methods that can augment permission systems by using data-flow analysis. TaintDroid [7] enabled dynamic taint analysis in Android systems and uncovered many cases of potential misuse of permissions by follow-

	Decision Count								
	Dynamic Exploration			User Study					
Application	Install time	First use	CSAC	Union	User #1	User #2	User #3	User #4	User #5
Dictionary	3	2	15	6	4	4	3	3	4
Flipagram	4	3	8	2	2	1	2	1	1
GasBuddy	3	1	10	5	3	3	3	3	3
WeatherBug	3	3	6	6	3	2	3	2	2
Yelp	4	3	15	5	4	3	3	2	3

Table 4: **Number of access control decisions needed to be made using three different systems for the 5 most decision-intensive applications. Results suggest that CSAC's use will not significantly increase the number of access control decisions that the user will need to make.**

ing the dynamic data flow of sensitive data in applications. Tripp et al. [17] improved the accuracy of TaintDroid by quantifying the amount of information leakage at any use. AppFence [10], MockDroid [6], and TISSA [18] suggested replacing private data that the user is not willing to share with fake data.

While data-flow analysis techniques are great tools for understanding where and how information leaks are occurring, they still have shortcomings that make them inapplicable for typical users. First and foremost, while taint tracking can provide detailed data on when and where private data is being used, current systems do not provide an approach to reason about the legitimacy of each request. This results in taint tracking to either be used solely for offline analysis or to be naively used for coarse-grained augmentations such as faking device identifiers. These systems also generally suffer from noticeable overhead (14% for TaintDroid [7]) and limited capability to detect implicit flows of private data. While CSAC's approach of assigning permissions per Activity is more coarse-grained compared to data-flow analysis, it provides its improvements with no computationally heavy analysis and minimum decision overhead.

Secure GUI: Howell et al. [11] and Roesner et al. [15] have examined incorporating standard predefined GUI components in applications to extract implicit case by case access permissions from the user (e.g. clicking on a button in the shape of a camera implicitly grants camera permission to the application for a session). The main shortcomings of these approaches is that they only work for specific privacy sensitive data (e.g., camera, mic) and confine app developers into limited GUI options. It also only provides a start point for when a permission is granted and does not provide an end limit for the access. Using CSAC can limit the scope of each permitted access to the particular Activity it has been granted in and stop an application from continuous use of that permission in other Activities.

Other approaches: Many user studies have also been conducted on users' interaction with permission systems. Lin et al. [13] studied users' expectations of privacy and suggested a crowdsourcing scheme for assigning appropriate permissions to an application in Android. Agarwal et al. [4] implemented a similar crowdsourcing scheme for iOS. Using crowdsourcing is especially beneficial to CSAC as it can remove the limited decision overhead imposed on the user as the result of finer level control.

Based on another user study, Jung et al. [12] suggested rate limiting sampling of sensor data as a possible method to enhance privacy. Using CSAC alongside such an approach will allow users to adjust both the frequency and accuracy of access to privacy sensitive data in each Activity.

6 Conclusion

In this paper, we introduced Context-Specific Access Control as a design approach for achieving the principle of least privilege in mobile platforms. We evaluated the feasibility of using the Activity abstraction in Android to achieve finer granularity of access control. Our results show that most applications' use of privacy-sensitive data is confined to a few Activities, thus allowing for CSAC's use without significantly increasing the number of access control decisions.

7 References

[1] Android developer's guide. `https://developer.android.com/guide/`.

[2] Apple Q&A on location data - April 27, 2011. `http://www.apple.com/pr/library/2011/04/27Apple-Q-A-on-Location-Data.html`.

[3] Google I/O 2015. `https://events.google.com/io2015/`.

[4] Y. Agarwal and M. Hall. ProtectMyPrivacy: Detecting and mitigating privacy leaks on iOS devices using crowdsourcing. In *MobiSys*, 2013.

[5] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA*, 2013.

[6] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading privacy for application functionality on smartphones. In *HotMobile*, 2011.

[7] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *TOCS*, 2014.

[8] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS*, 2011.

[9] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *SOUPS*, 2012.

[10] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *CCS*, 2011.

[11] J. Howell and S. Schechter. What you see is what they get: Protecting users from unwanted use of microphones, camera, and other sensors. In *W2SP*, 2010.

[12] J. Jung, S. Han, and D. Wetherall. Enhancing mobile application permissions with runtime feedback and constraints. In *SPSM*, 2012.

[13] J. Lin, N. M. Sadeh, S. Amini, J. Lindqvist, J. I. Hong, and J. Zhang. Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing. In *UbiComp*, 2012.

[14] A. Nandugudi, A. Maiti, T. Ki, F. Bulut, M. Demirbas, T. Kosar, C. Qiao, S. Y. Ko, and G. Challen. Phonelab: A large programmable smartphone testbed. In *SenseMine*, 2013.

[15] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE S&P*, 2012.

[16] M. Sheppard. Smartphone apps, permissions and privacy. In *Office of the Privacy Commissioner of Canada*, 2013.

[17] O. Tripp and J. Rubin. A Bayesian Approach to Privacy Enforcement in Smartphones. In *USENIX Security*, 2014.

[18] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on Android). In *TRUST*. 2011.