A Contextually-Aware, Privacy-Preserving Android Permission Model

Arjun Baokar

Acknowledgement

I would like to deeply thank Primal Wijesekera for the tremendous amount of work he has done on this project; we have been working together from start to finish. I would also like to thank to Ashkan Hosseini and Lynn Tsai for their work. Serge Egelman and David Wagner were amazing advisors throughout the entirety of the project. Thanks also to Dawn Song for her ideas.

A Contextually-Aware, Privacy-Preserving Android Permission Model

by Arjun Baokar

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor David Wagner
Research Advisor

(Date)

* * * * * * *

Professor Dawn Song
Second Reader

(Date)

Contents

Abstract

Smartphones contain a large amount of highly personal data, much of it accessible to third-party applications. Much of this information is safeguarded by a permission model, which regulates access to this information. This work primarily focuses on improving the Android permission model, which is known to have notoriously large amounts of sensitive data leakage, but many of its findings can be analogously applied to other mobile operating systems.

We evaluate the two currently employed Android permission models: ask at *install-time* and *ask-on-first-use* to determine if they fulfill user expectations of privacy, and find that this is not the case for either model. We analyze the different facets that comprise user expectations and recommend a better mechanism to satisfy these expectations without excess effort from the user. This mechanism incorporates the contextual nature of privacy into the permission-granting process through the use of a machine learning classifier.

We contribute the most extensive instrumentation of the Android operating system targeting user behavior and related runtime states to our knowledge, spanning across nearly 40 classes in the Android platform. This instrumentation allows us to utilize user behavior and system-level features to determine context for permission requests. The data from this instrumentation is used to generate features for the classifier. We evaluate the classifier on a large labeled dataset we collect from over 200 users using our modified operating system, and recommend ways to employ such a system in the real world based on our analysis.

1 Introduction

Smartphones have grown to serve billions of people, outstripping their desktop predecessor handily. In many places around the world, smartphones serve as the primary computing device for users. Android and iOS are the two most popular mobile operating systems, accounting for over 1.5 billion users [26], over 2.5 million applications, and 250 billion app downloads [33]. As the popularity of the platforms and smartphones has grown, so has concern over user privacy.

Android, the primary platform we examined, employs a permission model to regulate application access to data. Before Android 6.0 (Marshmallow), the model employed an ask-on-install policy, where the only way to deny a permission was to not install the application. Android Marshmallow and iOS employ an ask-on-first-use policy, which asks a user to make a decision the first time an application accesses sensitive data [4]. This is in the form of a system prompt at runtime, which the user must respond to. However, users and developers often do not understand these permission models [13]. Furthermore, it is not known if the prompts focus on data accesses that users find concerning. It is also unclear if the user decision for an ask-on-first-use prompt is consistent across subsequent requests of the same application requesting the same data [35].

There has been some work done on creating systems to protect users from data exposure, either through static analysis of applications, dynamically enforcing an access policy specified by the user [2, 18], or dynamic taint tracking [9]. However, the static analysis systems do not provide real-time protection from sensitive data exposure, and the existing dynamic systems do not work for all applications and require copious configuration effort from the user, which the authors acknowledge is highly inconvenient for the average user. Thus, we hope to create a solution which is backwards-compatible for all applications, requires minimal effort from the user, and preserves privacy by monitoring permission requests in real-time.

We explore the fundamental question of how smartphone platforms can empower users to control their information without added burden. Some facets of this question involve determining the feasibility and effectiveness of runtime prompting, as well as understanding users' mental models of privacy in the context of mobile devices.

To address issues with current models, we draw on Nissenbaum's theory of *contextual integrity*, which posits that privacy violations occur when information streams defy user expectations [28]. For example, a navigation application requesting location data follows contextual norms, while Angry Birds' need to request location is less obvious to the user. Using the contextual integrity notion, we hope to increase the efficacy of current permission systems by focusing on sensitive permission requests that are likely to defy user expectations to reduce the number of prompts users have to respond to.

We propose a new permission granting mechanism which utilizes machine learning to infer an individual user's privacy preferences and evaluate its feasibility. By leveraging inference, we want to avoid excessively prompting the user, which leads to habituation. Overall, this mechanism's goal is to be able to make contextual decisions for each individual user as they would with less user burden.

1.1 Threat Model

We assume that the platform itself, the Android operating system in this case, is trusted. The new permission model we propose protects users from applications they have installed that request permissions and sensitive information through the standard mechanism dictated by the operating system. Malware or exploits that subvert the operating system or the security monitor guarding permissions are outside the scope of our system.

2 Understanding User Expectations

The first step to being able to build a new permission model is to understand the nature of user expectations of privacy, quantify how many sensitive data accesses applications make, and determine which factors contribute strongly to user privacy decisions. To explore these questions, we conducted a field study of users in which we provided them with our custom instrumented version of Android to use for a week as their primary phone. Users then partook in an exit survey, in which we showed them various instances in which sensitive data accesses occurred. We asked the users whether those accesses were expected or surprising, and whether they would have blocked them given the ability.

In this study, we focused on a group of *sensitive permissions* (permissions that irreversibly leak user-sensitive data) recommended by Felt et al. to be granted via runtime dialogs [11]. These 12 permissions compose about 16% of all Android permission types, and are listed in Table 1. Our follow-up work also focuses on these sensitive permissions, since we focus on improving runtime prompts.

This section focuses on our field study published at the Usenix Security Symposium [35]. I will concentrate primarily on my contributions to the work, while explaining the overarching experiment done by our team to provide context for the results and analysis.

2.1 Methodology

We broke the problem into two primary parts: measuring the frequency of permission requests, particularly sensitive ones, and understanding user reactions. We hoped to use these avenues to evaluate the viability of prompting the user on permission requests based on their frequency, and gain insight into user privacy preferences. Thus, we first instrumented Android 4.1 (Jellybean) to collect phone usage data and monitor data access endpoints. This instrumentation collected data during a 36-user field study on Nexus S devices lasting one week, culminating in an exit survey.

2.1.1 Data Collection

For every protected request, the Android operating system checks during runtime whether an application was granted a permission at install-time in its manifest. We added a logging framework to the platform to determine when requests were made to the system, allowing us to record each instance where an application received resources. Our logging system was split into many `Producers` and a single `Consumer`.

Permission Type	**Activity**
WRITE_SYNC_ SETTINGS	Change application sync settings
ACCESS_WIFI_ STATE	View nearby SSIDs
INTERNET	Open network sockets
NFC	Communicate via NFC
READ_HISTORY_ BOOKMARKS	Read users' browser history
ACCESS_FINE_ LOCATION	Read GPS location
ACCESS_COARSE_ LOCATION	Read network-inferred location (i.e., cell tower and/or WiFi)
LOCATION_ HARDWARE	Directly access GPS data
READ_CALL_LOG	Read call history
ADD_VOICEMAIL	Add voicemails to the system
READ_SMS	Read sent/received/draft SMS
SEND_SMS	Send SMS

Table 1: The 12 permissions recommended by Felt et al. to be granted via runtime dialogs [11]. These are the *sensitive permissions* referred to throughout the paper.

`Producers` were placed throughout the platform in the many places that permission requests are made. A key point we monitored was the `checkPermission()` call in the `Context` implementation to give us access to the names of specific functions called from user-space applications. We also instrumented the `ContentProvider` class which facilitates application access to structured data (e.g., Contacts, Calendar) and the permission checks involved with `Intent` transmission. `Intents` allow data transfer between applications when activities are about to start in the receiving application. `Producers` were further used to monitor the `Binder` IPC mechanism, which allow applications to communicate with the Android system. This instrumentation produced the most logs, as it monitored calls to Android's Java API, which all applications use heavily.

I primarily worked on the `Consumer` with another teammate. The `Consumer` consolidated logs from the `Producers` placed in various parts of the platform. These logs were written to internal storage, since the Android platform cannot write to external storage. Internal storage tends to be limited; on our Nexus S devices, it totaled to 1GB. This space was also shared with installed applications, requiring us to be very careful with space usage. We compressed log data every two hours. When this log data was shipped to our server (daily, when the user connected to WiFi), it was deleted from phone storage to further preserve space. The average compressed log file totaled to 108KB consisting of roughly 9,000 events over 2 hours.

The `Consumer` also stored context information and metadata for each logged event. An event was a granted permission request, and each event log contained the following information:

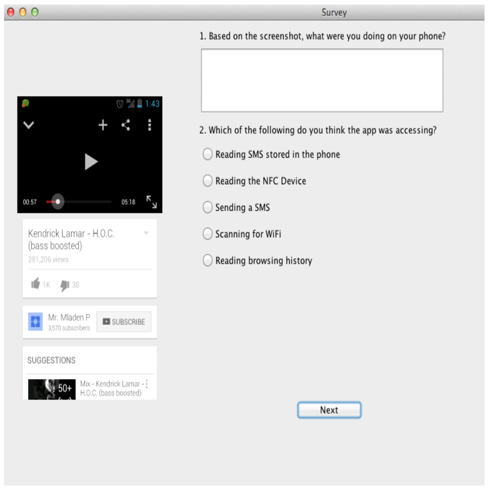
- **Timestamp:** The time when the permission request occured.
- **Permission name:** The permission requested.
- **Application name:** The application requesting the permission.
- **API method:** The calling method that resulted in the permission check. This could be used to infer specifically what data the application requested.
- **Visibility:** Whether the requesting application was visible to the user at request time. An application could be in any of four visibility states: running as the foreground process that the user was directly interacting with, running as a foreground process in the background for multitasking purposes (e.g., when a user switches applications and `onPause()` is called), running as a service with user awareness (e.g., sound, notifications), and running as a service without user awareness.
- **Screen Status:** Whether the phone screen was on or off.
- **Connectivity:** Whether the phone was connected to WiFi at the time, making data transfer possible.
- **Location:** This was the user's last cached physical location; we did not directly query the GPS to preserve battery.
- **View:** The name of the visible screen as defined by the XML DOM used to render UI in Android. This could be used to infer what the user was interacting with on the screen.
- **History:** Applications the user interacted with before the permission request.
- **Path:** In the event that `ContentProvider` was accessed for data stored on the phone, we recorded the path of the resource requested.

From these elements, I specifically worked on recording permission name, application name, screen status, connectivity, and view.

To maintain low system performance overhead, the `Consumer` also had rate-limiting logic. After 10,000 requests for any given {*application,permission*} pair, it checked to see if it exceeded 1 log per 2 seconds for that pair. If so, it recorded future requests for that pair with probability 0.1, and made a note to allow us to extrapolate during analysis to recover actual counts.

2.1.2 User Study

We recruited participants online in October 2014 through Craigslist by posting in the "et cetera job" section. We titled the listing "Research Study on Android Smartphones", explaining generally that the study was about how people interact with their smartphones. To avoid priming users, we avoided any mention of security or privacy. For each user, the study involved a 30-minute setup phase, a one week period where the user used our experimental phones as their primary mobile



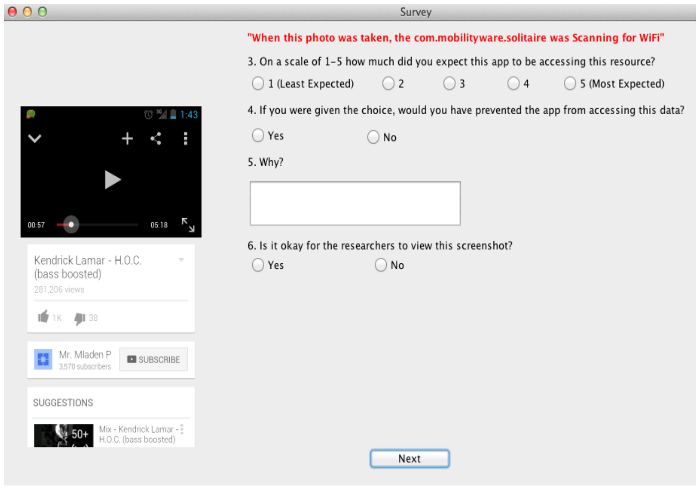
Survey

1. Based on the screenshot, what were you doing on your phone?

2. Which of the following do you think the app was accessing?

- Reading SMS stored in the phone
- Reading the NFC Device
- Sending a SMS
- Scanning for WiFi
- Reading browsing history

(a) Participants first establish awareness of the permission request based on the screenshot, and answered questions about context.



Survey

"When this photo was taken, the com.mobilityware.solitaire was Scanning for WiFi"

3. On a scale of 1–5 how much did you expect this app to be accessing this resource?

1 (Least Expected) 2 3 4 5 (Most Expected)

4. If you were given the choice, would you have prevented the app from accessing this data?

Yes No

5. Why?

6. Is it okay for the researchers to view this screenshot?

Yes No

(b) Participants then saw the resource accessed, and stated if they expected the permission and whether it should be blocked.

Figure 1: Exit Survey Application

device, and an in-person exit survey lasting between 30 to 60 minutes.

Potential participants were directed to download a mobile application that I developed. The application screened them to determine eligibility. Users would answer a short demographic survey on the application to ensure they were above 18 years of age, while the application determined their cell phone carrier, cell phone model, and installed applications. This information was important because our experimental Nexus S phones only could achieve 3G speeds on T-Mobile smartphones, and we checked the phone model to ensure that their SIM card was compatible with our Nexus S phones. The list of applications allowed us to pre-load the experimental phone we provided them with the applications they used to allow for a smooth transition to our phones, while minimizing setup time for them.

48 users made it through the application screening process, and arrived at our setup session. Here we screened 8 users out because their phones were MetroPCS (the screening app could not discern between T-Mobile and MetroPCS) and thus incompatible with our experimental phones. All 48 users were provided with $35 gift cards for attending the setup session, and 40 of them were given phones to use for a week. The setup session involved moving their SIM cards into the experimental phones, installing any paid applications (we only pre-installed free applications), and setting up their Google account on the phone to sync data and contacts. We had to manually sync the data if it was not linked to their Google account.

During the week, our logging system recorded usage data as well as sampled screenshots to provide context for questions posed to users during the exit survey, as seen in Figure 1a. At the end of the week, 36 users returned our experimental phones, and they were provided with another $100 gift card after the exit survey. We then flashed the phones to delete all user data from them. I personally conducted the setup phase and exit survey for half of our participants.

2.1.3 Exit Survey

We conducted the exit survey in a private room on a computer, with a researcher present to answer any questions without compromising user privacy. We did not view screenshots unless the participant gave us permission to do so. The survey was broken into three components: *screenshot-based questions*, *screen-off questions*, and *general personal privacy preferences*.

Screenshot-based questions first asked users to explain what they were doing on the phone (Figure 1a) in an open-ended response and asked them to guess which permission was being requested based on the displayed screenshot to set context. Permissions were displayed in simple English based on what resource they accessed. This first portion focused on understanding user expectations. The second portion (Figure 1b) revealed the accessed permission, and asked users how much they expected this access on a five-point Likert scale. We then asked if they would deny the permission if they had the option, followed by an open-ended question inviting them to explain their reasoning. Lastly, we asked for permission to view the screenshot. This process was repeated for between 10 and 15 screenshots for each user, determined through our *weighted reservoir sampling* algorithm. Reservoir sampling allowed us to choose a subset of items from a large set of unknown size, such as the number permission requests our users would encounter. We utilized weighting to ensure that applications that requested fewer permission requests would still be represented in our dataset.

The screen-off questions focused on understanding user expectations about protected resource accesses when users were not actively using their phone. Since the screen was off, there was no context for what they were doing. We asked users whether they would deny permission requests for 10 sampled {application, permission} pairs, and how expected these behaviors were, similarly to the screenshot-based questions.

To understand general personal privacy preferences, we had also users answer two privacy scales: Buchanan et al.'s Privacy Concern Scale (PCS) [5] and Malhotra et al.'s Internet Users Information Privacy Concerns (IUIPC) scale [24].

Three researchers, including me, independently coded the 423 responses for each of the open-ended questions. Before consensus, we disagreed on 42 responses for a 90% inter-rater agreement rate. Given the 9 possible codings per response, Fleiss' kappa yielded 0.61, indicating substantial agreement.

2.2 Results

We logged a total of 27M resource requests across more than 300 applications during the week-long period, which is equivalent to 100,000 requests per user each day. 60% of these requests were made while the screen was off, and an additional 15.1% were done by invisible background applications or services, for a total of 75.1% of requests invisible to the user. We focus on these requests, since they most likely defy user expectations, according to our study.

Permission	Requests
ACCESS_NETWORK_STATE	31,206
WAKE_LOCK	23,816
ACCESS_FINE_LOCATION	5,652
GET_ACCOUNTS	3,411
ACCESS_WIFI_STATE	1,826
UPDATE_DEVICE_STATS	1,426
ACCESS_COARSE_LOCATION	1,277
AUTHENTICATE_ACCOUNTS	644
READ_SYNC_SETTINGS	426
INTERNET	416

Table 2: The most frequently requested permissions by invisible applications and services per user/day.

Application	Requests
Facebook	36,346
Google Location Reporting	31,747
Facebook Messenger	22,008
Taptu DJ	10,662
Google Maps	5,483
Google Gapps	4,472
Foursquare	3,527
Yahoo Weather	2,659
Devexpert Weather	2,567
Tile Game(Umoni)	2,239

Table 3: The applications making the most permission requests while running invisibly, per user/day.

2.2.1 Invisible Permission Requests

A total of 20.3M (75.1%) requests were made with no visual cues to the user. The breakdown of all permission requests, based on our defined visibility levels, is as follows:

- **Visible foreground applications** constituted 12.04% of requests.
- **Visible background services** constituted 12.86% of requests.
- **Invisible background services** constituted 14.40% of requests.
- **Invisible background applications** constituted 0.70% of requests.
- Activity while the **screen was off** constituted 60.0% of requests.

We analyzed the invisible permissions most frequently requested, and the applications that most frequently requested those permissions. ACCESS_NETWORK_STATE, the most popular permission, was requested roughly once every 3 seconds. Applications use this to check network connectivity. More concerning were the location data accesses, which can be requested via the ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, and ACCESS_WIFI_STATE (using WiFi SSIDs) permissions. Tables 2 and 3 show this data, normalized per user per day.

Location data accesses also strongly violated contextual integrity. Contextual integrity requires users to be aware of information flows to deem them appropriate, however less than 1% of location requests were made by visible applications or display a GPS icon in the notification bar. This is because the GPS icon only appears when an application accesses the GPS sensor. 66.1% of location requests used the `TelephonyManager` to determine location from cell tower information, 33.3% used WiFI SSIDs, and used a built-in location provider the remaining 0.6% of the time. Of this 0.6%, the GPS sensor was only accessed 6% of the time, because the majority of queries were to the cached location provider. Thus, in total, roughly 0.04% of location requests displayed a GPS icon.

Another interesting juxtaposition is comparing screen off requests with screen on requests. Harbach et al. posit that users have their phone screen off 94% of the time [17]. Since only 60% of permission requests occur during this time, it seems that permission request frequency drops when users are not actively using their phone. However, some applications such as Microsoft Sky Drive and Brave Frontier Service actually make more requests while the user is not interacting with their phone. These applications primarily request the ACCESS_WIFI_STATE or INTERNET permissions. We hypothesize that these applications were performing data transfer or synchronization tasks, but checking this would require examining the application source code.

Lastly, invisible applications also read stored SMS messages 125 times per user/day, read browser history 5 times per user/day, and accessed the camera once per user/day. While these may not all be privacy violations, they do violate contextual integrity as the user is likely unaware of them.

2.2.2 Request and Data Exposure Frequency

Felt et al. recommended allowing benign data accesses, and only prompting for higher risk irreversible data requests. They classify the prompt-worthy data requests into the following categories, which are encompassed by the 12 sensitive permissions:

- Reading location information
- Reading browser history
- Reading SMS messages
- Sending premium SMS messages (those that incur charges), or spamming the user's contact list with SMS messages

91 of 300 (30.3%) applications requested these permissions, and these requests happened an average of 213 times per user/hour (roughly every 20 seconds). However, not all of these requests resulted in sensitive data being read or modified. We could differentiate between the cases which exposed data and those that didn't based on the called function name in the logs. The differentiation with examples is discussed next. For example, `getWifiState()` only reveals whether WiFi is on, but `getScanResults()` returns a list of nearby SSIDs. The majority of location requests were to `getBestProvider()`, which returns the optimal location provider based on application needs, and not actually location data. Similarly, most requests for READ_SMS requested SMS store information rather than actual SMS messages (eg. `renewMmsConnectivity()`). However, every request to SEND_SMS actually sent an SMS. In browser history, functions reorganizing directories (eg. `addFolderToCurrent()`) did not expose data, while looking at visited URLs through `getAllVisitedUrls()` would.

Overall, 5,111 of 11,598 (44.3%) sensitive permission requests exposed user data. A breakdown of data accesses can be seen in Table 4. Limiting runtime prompts to only these cases is still infeasible; this would cause a prompt nearly once every 40 seconds. Further solutions are discussed in §2.3.

Resource	Visible		Invisible		Total	
	Data Exposed	Requests	Data Exposed	Requests	Data Exposed	Requests
Location	758	2,205	3,881	8,755	4,639	10,960
Read SMS data	378	486	72	125	450	611
Sending SMS	7	7	1	1	8	8
Browser History	12	14	2	5	14	19
Total	1,155	2,712	3,956	8,886	5,111	11,598

Table 4: The sensitive permission requests (per user/day) when requesting applications were visible/invisible to users. "Data exposed" refers to the subset of permission-protected requests that resulted in sensitive data being accessed.

2.2.3 User Expectations

Our exit survey collected 673 participant responses ($\approx$19 per participant) for $\{$application,permission$\}$ pairs. Of these, 423 were screenshot-based (*screen-on* requests) and 250 occurred while the screen was off (*screen-off* requests). From the screenshot-based requests, 243 screenshots were taken when the requesting application was also in the foreground, and the other 180 screenshots were from invisible applications. Our reservoir sampling algorithm also attempted to diversify by the $\{$application,permission$\}$ pairs as much as possible.

Of the 36 participants, 80% (30 participants) said they would have blocked at least one permission request. In all, participants wanted to block 149 (35%) of all 423 screen-on requests. When participants rated requests for how expected they were (5-point Likert scale with higher number meaning more expected), allowed requests averaged 3.2, while blocked requests averaged 2.3. This supports the notion of contextual integrity, since users tended to block requests which defied their expectations. Furthermore, when queried for their reasons for wanting to block those requests, two themes emerged: (1) the request did not pertain to application functionality in their eyes, (2) the request involved resources they were uncomfortable sharing. In fact irrelevance to application functionality was the reason for 79 (53%) of the 149 permissions users wanted to block. Privacy concerns, particularly sensitive information such as SMS, pictures, and conversations, were cited as the reason for denying 49 (32%) of the 149 "blocked" permission requests. Conversely for allowed requests, users cited convenience (10% of allowed requests) and a lack of sensitive data involved (21% of allowed requests) as the reason for requests to proceed.

We wanted to explore how certain factors related to user decisions to block or allow permission requests based on user responses. I personally performed many of the statistical tests, confirmed the ones I didn't perform, and worked heavily on the data analysis portion. Our results are summarized below.

- **Effect of Correctly Identifying Permissions on Blocking:** Of the 149 cases where participants wanted to block permission requests, they were only able to correctly state what permission was being requested 24% of the time; whereas when wanting a request to proceed, they correctly identified the requested permission 44% (120 of 274) of the time. However, Pearsons product-moment test on the average number of blocked requests per user and the average number of correct answers per user did not yield a statistically significant correlation (r=0.171, p<0.317).

- **Effect of Visibility on Expectations:** Looking only at the *screen-on* responses, we noted that users had an average expectation scale value of 3.4 for the 243 visible permission requests and 3.0 for the 180 invisible requests. Wilcoxon' signed-rank test with continuity correction revealed a statistically significant difference between the expectation values for the two groups (V=441.5, p<0.001). Thus, users expected visible requests much more than invisible requests.

- **Effect of Visibility on Blocking:** We calculated the percentage of request denials for each participant, for both visible and invisible requests. Wilxcoxon's signed-rank test with continuity correction resulted in a statistically significant difference (V=58, p<0.001). Users were much more likely to deny invisible requests than visible ones.

- **Effect of Privacy Preferences on Blocking:** We used Spearman's rank test to determine that there was no statistically significant correlation (ρ=0.156, p<0.364) between users' privacy scale (both PCS and IUIPC) scores and their desire to block permission requests. This result was not surprising, as previous studies have demonstrated a difference between stated privacy preferences and actual privacy behaviors [1].

- **Effect of Expectations on Blocking:** This directly related to evaluating contextual integrity as a good model to follow while analyzing user preferences. We calculated average Likhert scores for user expectations for a request and the percentage of requests they wanted to block. Pearson's product-moment test resulted in a statistically significant negative correlation (r=-0.39, p<0.018), indicating that unexpected requests were more likely to be denied.

2.3 Analysis

Based on our field study results, we have shown that prompting on every data exposure is infeasible. However, there is also a dire need for better permission control mechanisms, as 80% of users indicated at least one violation of contextual integrity. We evaluate ask-on-first-use prompts, recommend a new version of ask-on-first-use prompting, and provide a new direction for reducing runtime prompts below.

2.3.1 Feasibility of Runtime Prompts

We first wanted to evaluate the feasibility of runtime prompts, as Felt et al. recommended for the sensitive permissions in Table 1, both in terms of accuracy and number of prompts. We used this to evaluate ask-on-first-use using the traditional {application,permission} pair (in iOS, since Android M had not launched at the time of our study), and a slightly modified version of ask-on-first-use in which we use a {application,permission,visibility} triplet.

Based on our study data, users would see an average of 34 (ranging from 13 to 77, $\sigma = 11$) runtime prompts in one week on a new phone if they were queried on first use of the triplet. By filtering out requests that do not expose data, we can reduce this number to 29 prompts. 21 (72%) of the 29 prompts are for location data, which the iOS ask-on-first-use model already prompts for. There has been no evidence of user habituation or annoyance with the iOS model, indicating that the 8 extra prompts our system would induce would likely not introduce significantly more burden on the user. However, the accuracy of the policy increases greatly with the triplet. Looking at cases in our study where a user was prompted for the same {application,permission} pair twice, we evaluated whether the user's first decision to block agreed with their subsequent decisions. The ask-on-first-use pair policy had a 51.3% agreement rate, while the triplet policy had a 83.5% agreement rate.

2.3.2 User Modeling

We constructed several statistical models to examine whether users desire to block certain permission requests could be predicted using the contextual data that we collected. We wanted to evaluate if a classifier could be used to predict user preferences, where the only runtime prompts would be those where the classifier had low confidence in its decision. Thus, we constructed statistical models (mixed effect binary logistic regression models) to see if we could predict a user's desire to block a permission request. Our models set the response variable to be the users choice of whether to allow or block the permission request. Our predictive variables consisted of contextual information that would be available during runtime: a user ID variable representing the unique user (**userCode**), permission type (**perm**), requesting application (**app**), and visibility of the requesting application (**vis**).

We found that creating separate models for the *screen-on* data and *screen-off* data resulted in much better fits for each model. For each model, we built two classifiers and evaluated their accuracy using 5-fold cross-validation, where our exit survey data served as ground truth. We tested each possible subset of our predictive variables with the model, using Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) to judge goodness-of-fit, to determine which variables provided the greatest insight. We found that (**vis**, **app**, **userCode**) had the best fit for the *screen-on* model, while (**perm**, **app**, **userCode**) provided the best fit for the *screen-off* model. The predictive power of the **userCode** variable also revealed that each user's individual privacy preferences play a large role in their decision to allow or deny permission requests.

We also provide some preliminary analysis using receiver operating characteristic (ROC) plots, which evaluate the tradeoffs between true-positive and false-positive rates. We computed the area under the ROC curve (AUC) for each model, and compared it to the random baseline of 0.5, with a maximum possible AUC of 1, indicating perfect classification. The *screen-on* classifier AUC was 0.7 and the *screen-off* classifier AUC was 0.8. This led us to believe that a classifier had a good chance of being able to predict user decisions, which inspired our next work.

3 Android OS Instrumentation

The small set of independent variables in our *screen-on* and *screen-off* models did not provide enough information to create a highly accurate model to predict user permission decisions. Furthermore, variables such as **userCode**, the user IDs of our participants, were not useful in a real-world setting. For a predictive model for Android permissions to effectively predict user decisions, we needed a much more comprehensive feature set. Thus, we instrumented Android 5.1.1 with the goal of collecting behavioral and contextual data during runtime to generate features for the classifier, as well as be able to perform further analysis on user behavior. The University of Buffalo provided the PhoneLab testbed, which allowed us to push over-the-air (OTA) Android OS updates to a group of roughly 300 users [25]. We used this to deploy our instrumented OS and collect data and generate a dataset which we could use to train and test our proposed classifier.

We chose to use an event-based logging model for two reasons: it provided flexibility for feature generation because it produced unprocessed information, and integrated easily with PhoneLab's logging model. Our instrumentation had to ensure that we covered all access points for each feature we hoped to record, including application-usage features without the ability to instrument the application code itself. This required the use of some side channels to infer user actions within applications. Furthermore, the performance implications of each instrumentation was strongly considered, as we had to ensure that there was no perceptible lag to the user.

With the instrumentation, we hoped to record aspects of users' interaction with their smartphones that might be indicative of their attitudes towards privacy in general and predict their responses to permission prompts. For example, we hypothesized that users who tended to take many pictures or not have screen locks would be less privacy conscious, and thus more likely to allow permission requests. The data that our system collected would help us find and verify such trends, and use them in a classifier. We attempted to log any user interaction events that could lead to predictive features. I discuss these features in detail in §5.1. In total, we modified over 25 classes in the Android platform code, and added a class of our own. A full list of our instrumentation points can be found in the Appendix.

3.1 Ensuring Coverage

There were various cases where instrumenting a user action required deep analysis of platform codepaths and Android developer APIs. Oftentimes, there were multiple ways for an application to implement a functionality, and we had to ensure all such paths were accounted for to reduce false negatives or positives in our event logging. I will focus on two especially challenging cases that I implemented in this section: camera usage and recording initial user data.

3.1.1 Camera

As mentioned before, we expected picture-taking habits to relate to user privacy attitudes. Determining when a user took a picture through any third-party application required a combination of multiple code path instrumentation and timing correlations on event logs. Android 5.1.1 has two major Camera APIs: `Camera` and `Camera2`. While most third-party applications such as Facebook, Snapchat, and Instagram use `Camera`, some popular applications such as Google Camera application use `Camera2`. `Camera2` is advertised as the recommended API moving forward, so later versions of Android will likely deprecate `Camera`. However, in both Android 5.1.1 and 6.0, both APIs are available for use by applications to maintain backwards compatibility.

The Camera API has a recommended takePicture() method, which was my main instrumentation point. A call to takePicture() would trigger the generation of a log containing information about which Camera device (i.e., the front-facing or rear-facing camera) was used, the application calling the method, and timestamp data. However, some applications using the Camera API would not appear in these logs, such as Snapchat. After scouring the developer API endpoints, I noted that an application could also choose to receive a stream of *preview frames* instead of calling takePicture(). The application can then choose which frame to modify or save within its own logic, which was outside of our observable scope. Thus, I settled for logging the start of any preview frame stream in onPreviewFrame(), coupled with the same metadata as in the takePicture() logs. While an application could capture any number of pictures during a preview frame session, we could at least enumerate the number of preview frame sessions and camera device switches during sessions.

At the time of our instrumentation, Camera2 had limited documentation and even fewer applications using it. However, we could not ignore it since some users would inevitably install applications using that API. The capture() method functioned similarly to the takePicture() method in Camera, so that provided a straightforward port. The richness of features in Camera2 provided a challenge; picture modes like burst pictures and action shots were built in through different methods. I found that the setRepeatingRequest() method would be called to generate CaptureRequests for most special picture modes, and created an event log there with all of the metadata mentioned above.

The CaptureRequest logs required time correlation analysis because the setRepeatingRequest() would be called multiple times for special picture takes. For example, during a burst picture, setRepeatingRequest() is called twice within a 300 ms window, and three more times within a 700 ms window. I accounted for such patterns in the classifier's feature extraction code rather than within the event logging instrumentation to avoid performance issues.

3.1.2 Settings and Initialization

To be able to compare user privacy behaviors, we needed to record default system settings, as well as previous user settings for the majority of our instrumentation features. Since users were not using their phones for the first time in our experiment, we needed to be able to record any changes they had previously made to the system as well. We decided to only do this on phone start up, because all settings at once impacts performance. We were also guaranteed the initialization to happen at least once for all users, since Nexus 5 devices needed to be restarted after our OS version was installed on their devices.

A major difficulty involved integrating user and system space components in one log. The majority of our logging for initialization settings was in the ActivityManagerService, but we also had some logic in NfcService and PreferenceUtils to reach some data that was not accessible from ActivityManagerService. We recorded all modifiable security settings, NFC settings, location settings, developer mode configurations, and audio settings. Of these, NFC proved to be particularly difficult because the NfcService is not necessarily started by the time the `ActivityManagerService` is. To account for this, we instrumented the `NfcService` to asynchronously provide information to the initialization logging point in `ActivityManager-Service`.

Furthermore, while recording all system settings and changes, we had to consider the Settings application that comes pre-installed on Android devices. This is a user-space application that makes changes to system-space settings. Since I was instrumenting both the initialization in `ActivityManagerService` and the `SecuritySettings` UI component in the application, I balanced which logs to record in user-space and system-space. Most user-triggered change actions were logged as part of the user-space application, while the value changes to settings were recorded in system-space at the service level. Combining the information in those logs during the data analysis phase would help recreate the phone's system state during the permission request while maintaining all setting value changes.

3.2 Side Channels

We often had to record observable effects to infer what was going on within the system. Often, this was because we did not have access to application code but felt that the user interactions within an application would dictate their functionality expectations from the application. As mentioned before, user perceptions of functionality strongly correlate with their willingness to allow or deny permission requests. In other cases, we could not directly record an event because of performance reasons. For example, instrumenting the network interface or the `Intent` mechanism proved to create noticeable lag on phones. While we used side channels in determining many user actions, such as social media application interactions, I will focus on the most prominent example: the Chrome browser application. Similar ideas were extended onto other popular mobile browsers including Firefox and Opera.

Users' mobile browsing habits seemed likely to relate to their privacy preferences. In particular, we felt that understanding how often users used incognito tabs (also known as "private browsing" tabs), how often they visited websites supporting HTTPS, and how they interacted with SSL warnings would indicate their privacy consciousness. Without access to the Chrome browser code, we had no way to record this directly. Instead, we resorted to side channels to infer when these actions were occurring, without looking at URLs to maintain our participants' privacy.

To record incognito tabs, we used icons on the notification bar. The notification bar at the top of the screen is generated by the platform, so we instrumented it to see if an incognito icon appeared. This meant that the user was using incognito tabs. We confirmed this by monitoring the name of the `Activity` rendered by the platform; non-incognito and incognito tab `Activities` had different names. To determine if the site had HTTPS, we checked if the "locked" HTTPS icon was rendered by the platform, since we could observe every image file it rendered. Similarly. monitoring the images rendered allowed us to note SSL warnings displayed by the browser, and differentiate between different warnings. If the next page load rendered an "insecure" HTTPS icon, we knew that the user had chosen to proceed through the warning.

Side channels played a large role in allowing us to gain insight into application behavior and user

Action	Stock Android	Instrumented Android	Difference
Application Switch	530 ms	560 ms	30 ms (5.7%) slower
Capture Photo	1.1 ms	2.3 ms	1.2 ms (109%) slower
Get Location	9.3 ms	11.5 ms	2.2 ms (23.6 %) slower
Get Wifi SSID	2.3 ms	2.8 ms	0.5 ms (21.7 %) slower
Send SMS	16.2ms	18.4 ms	2.2 ms (13.4%) slower
Read Browser History (including Bookmarks)	84.6 ms	49.8 ms	34.8 ms (41.1%) faster
Read NFC Tag	6.9 ms	5.8 ms	1.1 ms (15.9%) faster

Table 5: Amoritized performance difference on benchmarked actions.

actions, while reducing performance overhead without losing information in our logs.

4 Performance Evaluation

The performance impact of our operating system instrumentation was evaluated on the LG Nexus 5 device. Our analysis examines two resources: latency and battery usage. In latency, we quantify the lag, slowdowns, and in some cases speedups, to the Android system that our instrumentation incurs. In the battery portion, we estimate the extra power usage that our instrumentation causes. I created the latency benchmark, and ran the battery consumption study.

4.1 Latency

It is critical for usability to ensure that our system does not introduce any perceptible lag into the user experience. Thus, we developed an application to identify and quantify the slowdown in user-facing tasks. We do not modify the instruction set or kernel logic, so existing microbenchmarks were not very useful in this process.

I chose a set of common user actions, shown in Table 5, that accesses the majority of the spectrum of instrumentation we have added to the Android platform. I created an application to perform these actions, which was installed on both stock Android 5.1.1 and our instrumented version. These actions were performed 50 times each and averaged to avoid noise from individual runs. In the case of actions that are normally cached, such as reading browser history, we ignored the first time the action was taken, and recorded the time it took post-caching instead.

In most cases, our instrumentation is slower, and the percentage slowdown varies from 5% (application switch) to 109% (capture photo). However, it is more important to look at the absolute difference for our purposes. If a user takes an action and the phone responds in 50*ms* or less, the user considers it an instantaneous action [27]. For all of our tested actions, the absolute slowdown is less than 30*ms*. To a user, this difference is completely imperceptible.

During our PhoneLab user study with over 200 users running our instrumented version, no users reported latency issues or unexpected application behavior. Our goal was to ensure that no user found our instrumentation intrusive, obstructive, or otherwise caused them to alter their usage patterns to compensate for lag introduced by our changes. Given that none of the study participants noticed any slowdown on the device, our instrumentation's performance costs appear to be imperceptible to the user.

4.2 Battery

Our impact on battery performance was quite difficult to measure. We tried a few methods of estimating it, each with a sizable error margin. To understand our estimates, it is important to note that our instrumentation spawns no new threads. This supports the concept that our battery consumption overhead is quite low, since spawning new threads is one of the main causes of battery usage increase.

To empirically measure battery usage, we had pilot study participants use stock Android 5.1 and record battery levels periodically throughout the day. During the recording periods, they would look at the battery monitor breakdown provided in the settings menu to determine what was using the most power. They then went through the same process with our instrumented version of Android 5.1. Unfortunately, this process is highly susceptible to variations in user activity on the device. We tried to remedy this by measuring the proportion of usage accredited to the Android system. We found that our instrumentation consumed at most 2% more battery.

Given our rough experimental data and the fact that we do not spawn any new threads, we believe this estimate to be roughly correct. We are looking into more rigorous ways of testing the effects of our instrumentation on phone battery life.

5 Classifier

Our results in §2.3 led us to believe that a predictive model could be effective in reducing runtime prompts by predicting user decisions for most data accesses. More specifically, our goal is to predict user responses to permission request prompts based on their past decisions and behavior using a classifier. We gathered data across many users through a user study on PhoneLab and used this data to train and evaluate the classifier, based on their answers to our runtime prompts. Ideally, this classifier will predict a user decision to a permission request accurately, which we can use to allow or deny a permission request instead of prompting the user. This results in a permission model that has less prompting, annoyance, and risk of habituation for the user while still adhering to their privacy preferences. Currently, we train a single global model across all users; we hope to create more personalized models in future work.

While we had some ideas for features to our classifier from our *screen-on* and *screen-off* model analysis, we needed to explore a much broader set of features while replacing some predictive variables. For example, the **userCode** variable caused a problem; training a classifier for each user would require large amounts of labeled data, generated through runtime prompts. To avoid the overwhelming habituation and user annoyance problem, we needed to determine metrics that

Feature	Setting Type
User accesses security settings	Security
User actively modifies security settings	Security
Type of lockscreen	Security
Length of lockscreen password	Security
Is lockscreen password hidden	Security
User changes type of lock	Security
Location tracking on/off	Location
Granularity of location	Location
NFC on/off	Misc
Developer options enabled/disabled	Misc

Table 6: User actions recorded in settings features.

could be measured at run-time to replace the **userCode** but still capture each user's uniqueness as determined by their individual privacy behaviors.

5.1 Features

The new features we proposed have two goals: help us infer personal user privacy preferences for each individual, and help understand context surrounding a user decision. The labeled dataset is provided through user responses to permission prompts during a large-scale user study on PhoneLab preceding the building of the classifier. We collect 33 feature groups, which sum to over 16,000 individual features, resulting in the most extensive instrumentation of the Android system to our knowledge. Due to space constraints, I only highlight the thought process behind feature generation and delve into some of the more interesting feature categories. The full list of instrumentation points feature groups can be found in the appendix.

5.1.1 Settings-Related Features

An excellent source of preference-indicative information lies in phone settings, since those are directly controlled by the user, barring default initialization values. We focused on two main subsets of settings: security settings and location settings. We gathered information about each user's settings and extracted features from this.

Table 6 lists the subset of the data we collect from the settings panel. Security settings indicate privacy preferences directly, since a user can control access to her phone through this menu. Location settings help us better understand the user's location preferences; our previous study demonstrated that whether users considered location private varied greatly with the user and the context, so we expected this feature to be helpful in predicting user preferences regarding application use of location permissions. We also record NFC and developer options settings, since they are a means of exporting or importing data from the phone. To account for default settings, we record both the default values and current values of each setting for each user, so we know if these settings have been changed. We can compare these with their current values to detect which ones users actively changed.

I personally instrumented all of the features mentioned in this section.

5.1.2 User Behavior Features

Many of our features directly focus on identifying user privacy-related behavioral habits of users; this subset of features is the most useful in creating a privacy preference "profile" for a user and clustering users by their similarities in privacy preferences. Many of these features are user actions taken on the device which do not directly relate to security, but help us profile the user's behavior patterns in general. These features fall into a few general categories:

- Screen Timeouts
- Ringer Preferences
- Phone Call Habits
- Mobile Browser Habits
- Camera/Picture Habits
- Notification Habits
- Two Factor Authentication

Our instrumentation indicates whether the screen turned off due to a timeout being reached, or if the user physically presses the lock screen button to turn off the screen. Our hypothesis is that a user physically locking their screen more often is more concerned about threats to their privacy. Ringer preferences and phone call habits imply how publicly a user carries on their communications. A user that keeps their phone ringer on loud essentially broadcasts incoming messages and phone calls to everyone in the vicinity, whereas a user keeping their phone on silent or vibrate does not. Similarly, speakerphone and headphone usage on phone calls can be used to infer the publicity of conversations.

Mobile browsing habits may be a strong indicator of user privacy preferences. Some features that seem promising are the percentage of SSL-secured links that users visit, how often they encounter SSL warnings, and the percentage of incognito tabs they open as opposed to non-incognito tabs. We hypothesize that a user that visits a higher percentage of SSL-secured links, uses incognito tabs often, and does not ignore SSL warnings often will tend to be more conservative with their data.

Picture-taking and uploading habits are associated with social media, thus we consider them as indicators of privacy preferences. Furthermore, this feature group easily demonstrates how one instrumentation point results in multiple features. Our instrumentation logs an event on each picture capture by the user, and when an application accesses the `MediaStore` to upload the picture. We transform these two types of event logs the following features: the number of pictures users take, the number of pictures they upload, the percentage of those pictures that are "selfies" (pictures of themselves), the percentage of pictures that are not selfies, etc. The feature generation for camera logs are based on our hypothesis is that the people that are more willing to share pictures, especially of themselves or family, are more likely to share other information as well.

We also measure how often users get notifications from their applications and how quickly they respond to them. We believe that users that have much more communication are likely to be more social. Also, by differentiating which applications are sending the notifications, we can further infer actions users take on social media applications. Lastly, the use of two-factor authentication is one of the most obvious indicators we have of how much a user cares about their security. We hypothesize that a strong desire for security is tied to a strong desire for privacy, so features related to this seem quite promising as well.

I personally instrumented all of the features highlighted in this section, except for mobile browser habits.

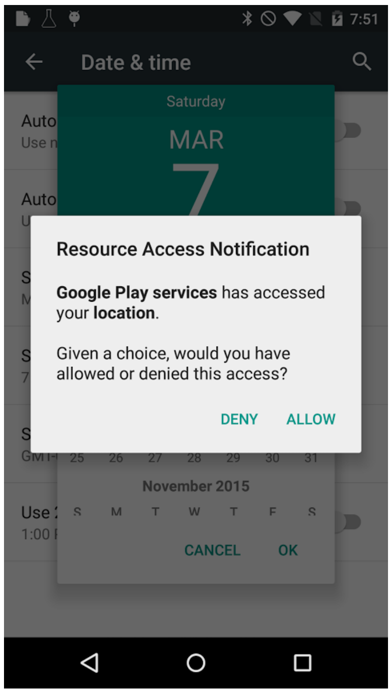
5.1.3 Runtime Features

Run-time features allow us to understand which activities the user is interacting with directly and create a timeline of the processes running on the system. These features provide us the greatest insight into exactly which user actions were taking place on the device. The three categories of run-time features that provide us with which applications and services were running on the device are memory prioritization, activity switches within an application, and sensitive permission requests. We also maintain a list of running processes and their visibility levels to the user.

Memory prioritization directly indicates which applications are running in the foreground, in the background as services, or invisibly to the user. The process with the highest priority in memory is always a foreground activity that the user is directly interacting with. As the priority is lowered, we can know if an application is a visible service (such as Spotify, which retains a notification in the notification bar when it is playing music in the background), or an invisible service (such as Google Play services). Visibility is essential to the inferring expectedness of an information flow, and thus is critical to the context of a permission request [35].

Activity switches within an application can be used to infer which actions a user is taking within the application itself. One of the fundamental limitations with instrumenting the Android operating system is that one cannot instrument third-party application code. Activity switches help us overcome this limitation by letting us infer which view of the application is in the foreground. For example, activity switching allows us to differentiate between when a user is browsing their Facebook news feed or viewing another Facebook user's profile. Using these switches, we can also compute how long a user spends in each activity on each application on her phone, letting us understand what the user tends to use her smartphone for. This helps us create a more accurate snapshot of the context under which a privacy decision was made.

We have also selected a set of sensitive permission requests that have irreversible results, which can potentially be harmful to the user. Sensitive permission requests are monitored to give us a list of data access requests from applications. We can identify how much data applications seek (and are granted) in real-world usage situations. Furthermore, the prompts that we generate for sensitive



Resource Access Notification

Google Play services has accessed your **location**.

Given a choice, would you have allowed or denied this access?

DENY ALLOW

Figure 2: Example of a prompt chosen by reservoir sampling that a user would see.

requests to provide ground truth for the classifier are generated based on this category of features.

5.2 Data Collection

Training a classifier requires a large dataset. In our case, no dataset existed, so we had to generate our own. To achieve this, we deployed our highly instrumented operating system on PhoneLab, which installs our operating system on the phones of roughly 300 users [25]. Our experiment collected data for 5 weeks on Phonelab, collecting feature data as well as generating ground truth through a system-UI user prompt we created, as seen in Figure 2.

Our runtime user prompt gained ground truth on which sensitive permission requests that users considered privacy invasive; our prompt is similar in appearance to the ask-on-first-use prompt in Android Marshmallow, with the caveat that it did not only appear on first use. As shown in Figure 2, we informed the user of a sensitive permission request and the requesting application, and asked if she would have denied it if given the opportunity. During the experiment, we made it clear to users that any answer would not affect their experience with the phone, and that their answer to the prompt would not actually affect information access patterns.

We use a weighted reservoir sampling algorithm to determine which permission requests generate a prompt. The weights in our algorithm are chosen inversely to {*permission request, requesting application*} combinations frequencies to ensure that we sample as diverse a set of combinations as possible. This ensures a ground truth sample for a large set of cases, allowing us to generalize and intialize our models well. We limit the phone to prompting the user once per day to minimize habituation.

PhoneLab has the option for a user to opt out of an experiment at any time. Thus, not all 300 users participated in our experiment. In total, we had 204 users participate in our experiment, with 133 of them providing more than 20 days of data. We recorded over 176M events and 96M permission requests (one every 6 seconds per user). Users answered 4,670 runtime prompts over the course of the entire experiment, which translated directly to labeled data points with which to train our classifier.

5.3 Offline Model

Using the 4,670-point labeled dataset we generated in our PhoneLab experiment, we first wanted to determine which features were effective in predicting user decisions. During the feature selection process, we also evaluated various models and attempted to increase accuracy as much as we could. I worked heavily on all parts of the feature selection and modeling pipeline, so I will discuss the process in its entirety.

5.3.1 Feature Selection

We used a combination of regression tests determining statistically significant correlations and common machine learning techniques such as information gain values and regularization methods to determine which features were most predictive. Through this process, we hoped to gain insight into which user behaviors could be used to infer privacy preferences. In particular, we utilized logistic mixed effect regression, random forests, and regularization to reduce overfitting while emphasizing important features.

We first confirmed if the results of our previous study, particularly pertaining to the predictive power of application visibility, still held. All models reported visibility as one of the most predictive factors, and logistic regression considered it statistically significant. We then analyzed how application usage correlated with users' decision rates. For each application used (not just installed) by at least 10 users, we measured usage time and checked to see if it significantly correlated with their denial rate for permission prompts through the Kendall rank correlation test. We found 6 applications for which a user's usage time significantly correlated with their denial rate across all permission prompts. The importance scores assigned by the random forest model confirmed the predictive power of the applications. The list of significant applications can be found in the appendix.

We also used the Kendall rank correlation test for a few other features, which served as independent variables, with denial rate as the dependent variable. Some interesting features that correlated significantly are listed below.

- how often a user allows their phone screen to time out instead of actively locking it (measured as a ratio of timeouts:locks)
- how often users access HTTPS links (measured as a ratio of HTTPS:non-HTTPS links)
- the number of downloads the user initiates

Feature	Feature Category
Number of websites visited	Behavorial
Ratio of HTTPS-secured to non-HTTPS-secured sites	Behavioral
Ratio of websites requesting user location to those that did not request it	Behavorial
Number of downloads initiated by user	Behavorial
Type of screen lock (password, PIN, or pattern)	Behavorial
Number of screen unlocks daily	Behavorial
Total time spent unlocking	Behavorial
Ratio of screen locks to due timeout or user action	Behavorial
Number of phone calls daily (both made and received)	Behavorial
Amount of time spent on phone calls	Behavorial
Amount of time spent on silent mode on phone	Behavorial
Requesting application visibility	Runtime
Permission requested	Runtime
Denial rate per user for an {app,perm,vis} combination	Aggregate
Denial rate per user for an {foreground app,perm,vis} combination	Aggregate

Table 7: Set of features used in SVM and RF classifier.

- the number of calls a user makes and receives per day (essentially the number of calls a user participates in daily)
- the average call duration of a user

These features were also confirmed by the models as having high importance, and each of them helped increase accuracy, precision, and recall of our models. Interestingly, the effect size of each correlation was quite small; nearly all of them were less than 0.2. We believe this is partially because of our large sample size, consisting of 4,670 of decisions across 204 users. Furthermore, each feature has a large standard deviation. We hypothesize that the large variance in user smartphone interactions due to time of day and day of the week contribute to the large standard deviation, further lowering effect sizes [7]. Specifically in the case of the application usage, we can also attribute the smaller effect size to the fact that none of the applications were used by all users.

5.3.2 Model Evaluation

Improving our predictive model is an ongoing process, which we will likely continue to iterate in our future work. Currently, we have tried a logistic mixed-effect regression model for feature exploration, a random forest model, and a support vector machine (SVM) model for prediction. All reported accuracy and AUC values are averaged over five-fold cross-validation, with folds generated over each decision. This does not take temporal ordering into account meaning later decisions could be used to predict earlier decisions; we have yet to assess how this affects the accuracy of the model. We also used grid search to find optimal hyperparameter values for each model. The example confusion matrices are chosen from a single random fold, but are representative of the other four folds.

	Actual Deny	Actual Allow
Predicted Deny	504	50
Predicted Allow	29	344

Table 8: The confusion matrix for logistic mixed-effect regression on a test set of 927.

	Actual Deny	Actual Allow
Predicted Deny	536	18
Predicted Allow	22	351

Table 9: The confusion matrix for our SVM classifier on a test set of 927.

	Actual Deny	Actual Allow
Predicted Deny	508	41
Predicted Allow	35	343

Table 10: The confusion matrix for our random forest classifier on a test set of 927.

The final set of features we used in our SVM and random forest models can be found in Table 7. We group the features into three major categories: behavioral, runtime, and aggregate features. The behavioral features listed were the most predictive tended to relate to mobile browsing, phone calling, and screen locking habits. The predictive runtime features were the permission requested and the visibility of the requesting application. Interestingly absent is the requesting application itself, which did not have much predictive power. We believed this was a result of the numerous possible applications creating sparsity and reducing predictive power. To remedy this, we tried other features that would provide similar information, such as a boolean indicating if the requesting application was popular or highly used, but these features were also not predictive. The aggregate features, which computes the denial rate for permission prompts for each {*application, permission, visibility*} combination, was inspired by the logistic mixed-effect regression model. We also compute the denial rate for different foreground applications, as it relates to decision context.

We first used the logistic mixed-effect regression model for binary classification (keeping in mind that we use logistic regression as a classification model, not regression as the name would imply). Mixed effects account for two kinds of effects, inter-user and intra-user, and model these as fixed and random effects respectively. Thus, in cases of repeated measurements, such as our data collection process, they tend to do very well. The aggregate features mentioned above were our attempt to introduce intra-user effects for the SVM classifier, which does not inherently account for them. Essentially, without our aggregate feature, fixed-effect models like SVMs treat a single user's decisions as independent of each other.

Our mixed-effects model had an accuracy of 92.3% with an AUC of 0.92. We also used the same feature set on an SVM and random forest classifier. The SVM (with an RBF kernel) achieved an accuracy of 95.7% with an AUC of 0.95, and the random forest had an accuracy of 91.8% with an AUC of 0.91. Tables 8, 9, and 10 contain the confusion matrices for each of our models. The key point to notice is that some models tend to default towards denying, such as the logistic mixed-effect regression model. This means that the majority of their errors are when they choose to deny instead of allow a permission. We prefer these models to those that allow more, because falsely allowing data is more costly than denying it. Granted data cannot be redacted, but denied data can always be granted at a later time.

Overall, the accuracy values we see are very encouraging, indicating that the machine learning approach could be promising in practice. Furthermore, through our behavioral features, it seems that passive behavioral data could be used to improve predictions and reduce prompting for the user. We have yet to evaluate these systems in full deployment, but these initial results indicate that this direction is one worth pursuing.

6 Related Work

Various other work has shown that applications access data beyond their expected functionality [9, 22, 29]. In fact, static analysis tools have demonstrated that around 35% of applications are over-privileged [10]. Many of these over-privileged applications collect highly personal information for advertising, customer profiling, and similar purposes which users do not expect. Some of the information they collect include location, stored data including images and text messages, contact list, unique device identifiers (such as IMEI numbers) and others [9, 8]. Researchers have also examined user privacy expectations in the context of application permissions, noting that they were quite surprised by how much data background applications collect [20, 34]. Users' reactions varied from mild annoyance to actively wanting to seek retribution for their privacy invasion after being shown the risks associated with specific permissions [12]. Our work confirms many of these results while adding more granularity to our understanding of user privacy expectations.

Many researchers have also attempted to improve the current install-time permission model by better explaining privacy risks associated with each permission more clearly, and creating recommendation systems to recommend more privacy-conscious alternatives to applications [6, 21, 16, 22]. However, privacy is rarely the primary objective in a user's decision to install an application, so recommended alternatives are often ignored [32]. Other researchers have added fine-grained access control to the Android permission model and successfully increased user privacy control [6, 19, 31, 18]. However, these methods pose a usability problem, as demonstrated by the fact that hardly any real-world users take advantage of such options [14, 3].

Other systems attempt to take the recommendation system idea further; they utilize other users' security concerns as ratings into a recommendation service for applications [2, 15, 36]. Some work focuses on clustering users on privacy preferences, based on how users react to different access requests [30, 23]. While these systems do achieve high accuracy, they do not operate on a real-world randomized dataset, require users to rate different permission types on Likert scales preemptively, and respond to 10% of permission requests as runtime prompts. This places a burden on the user, and requires them to answer a prompt nearly every three minutes [35], which results in a system with high habituation and user annoyance. We want to look into extending their work into a realistic setting with usability constraints, perhaps as part of our future work.

7 Future Work

The main focus of our ongoing and future work is to test the viability of the classifier in the wild. We will be looking into ways to improve the cross-validation accuracy of the classifier and to generate new features that better capture context from the rich instrumentation data we have. We also plan to run a user study in which users are given phones with our custom version of the Android OS containing our classifier to actually measure the classifier's accuracy in a realistic setting. They will use these phones for a period of time, while we record the decisions our classifier makes. Through a similar study as the one in §2, we hope to use a screenshot-based study to ask them if they agreed with the decisions our system made. We also want to conduct in-depth exit interviews with all participants to understand how well our system fits user expectations.

Much of our data supports the idea that users are unique when it comes to privacy preferences, and that these preferences change over time. We hope to encompass this by making an online learning model, meaning that it can be updated with new data points in real-time. We plan to use machine learning algorithms which support gradient descent and utilize a trusted server on which to train the models. User phones will collect features through our existing instrumentation, simplify it into a feature vector, and send it to the trusted server. The server will train the model associated with the phone, and return a weight vector to the phone, which it will use for classification. This ensures that phone battery and processing power is not spent on the expensive training phase, and that models do not need to be stored on the limited memory on the phone. Based on our preliminary calculations, each feature vector and weight vector exchange should be less than 5KB, maintaining low data transfer overhead. These exchanges would happen no more than a few times a day.

8 Conclusion

In both the field study and the PhoneLab experimental study, users chose to deny a significant portion of permission requests (ranging from 35% to 60% in the two studies), and 80% of users chose to deny at least one request. Under install-time permission prompts, all of these requests would have been granted, and the ask-on-first-use model's assumption of users not changing their decision for an {*application,permission*} pair is incorrect nearly 50% of the time [35]. This leads us to believe that user privacy expectations are not encompassed by current permission models. By defying user expectations, current models also violate Nissenbaum's theory of contextual integrity [28]. We thus propose a model which attempts to leverage the contextual nature of user privacy into determining which data accesses are privacy-invasive.

Given the significant effect of visibility on user decisions, we provide a method to improve the existing ask-on-first-use model by utilizing {application, permission, visibility} triplets rather than the current {application, permission} pairs. This change would increase the probability that subsequent decisions made by the ask-on-first -use model comply with the user's decision from 0.51 to 0.84. However, our work also demonstrates the infeasibility of runtime prompts for all sensitive data accesses because of the overwhelming frequency with which accesses occur. Thus, we strive to create a system that confronts the user minimally while being able to gain enough information to automatically infer user privacy preferences and make decisions as they would.

The data from our initial field study further provided an insight into the factors that differentiate users' decisions. Our models revealed that individual user characteristics greatly explained the variance between different user decisions, inspiring us to understand how to capture these characteristics automatically and unobtrusively to guide our automatic inference-based system. Thus, we created a highly instrumented version of the Android OS to capture behavioral patterns and contextual data aiming to measure individual user characteristics through metrics collectible at runtime.

As we continue to add behavioral features to our offline model, we continue to improve prediction accuracy. Our ability to infer user privacy preferences based on their behavioral features grows as well, leading us to believe that the individual user variance can be handled through behavioral metrics collectible at runtime. Our probability of conforming to user decisions already matches or exceeds that of both the install-time and ask-on-first-use models, illustrating the potential of an automated permission model.

In conclusion, we show the real-world circumstances under which Android permission requests are made. Our study demonstrates ways of improving contextual integrity in existing models, while highlighting many of their limitations. We also contribute a framework for an automated model that attempts to make the same contextual decisions that an Android user would make. Through this work, I hope to create a convenient process through which users are empowered with far more control over their sensitive data. As information becomes increasingly accessible through our mobile devices, permission models will likely become increasingly complex. Given usability constraints, automated mechanisms that make contextually-aware decisions may become one avenue towards empowering users without further complicating their lives.

References

[1] A. Acquisti and J. Grossklags. Privacy and rationality in individual decision making. *IEEE Security & Privacy*, pages 24–30, January/February 2005. `http://www.dtc.umn.edu/weis2004/acquisti.pdf`.

[2] H. M. Almohri, D. D. Yao, and D. Kafura. Droidbarrier: Know what is executing on your android. In *Proc. of the 4th ACM Conf. on Data and Application Security and Privacy*, CODASPY '14, pages 257–264, New York, NY, USA, 2014. ACM.

[3] H. Almuhimedi, F. Schaub, N. Sadeh, I. Adjerid, A. Acquisti, J. Gluck, L. F. Cranor, and Y. Agarwal. Your location has been shared 5,398 times!: A field study on mobile app privacy nudging. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 787–796. ACM, 2015.

[4] Android Developer, http://developer.android.com/guide/topics/security/permissions.html. *System Permissions*, 2015.

[5] T. Buchanan, C. Paine, A. N. Joinson, and U.-D. Reips. Development of measures of online privacy concern and protection for use on the internet. *Journal of the American Society for Information Science and Technology*, 58(2):157–165, 2007.

[6] E. K. Choe, J. Jung, B. Lee, and K. Fisher. Nudging people away from privacy-invasive mobile apps through visual framing. In *Human-Computer Interaction–INTERACT 2013*, pages 74–91. Springer, 2013.

[7] R. Coe. It's the effect size, stupid: what effect size is and why it is important. British Educational Research Association, 2002.

[8] F. T. Comission. Android flashlight app developer settles ftc charges it deceived consumers.

[9] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taint-droid: 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, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[11] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *Proceedings of the 7th USENIX conference on Hot Topics in Security*, HotSec'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[12] A. P. Felt, S. Egelman, and D. Wagner. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 33–44, New York, NY, USA, 2012. ACM.

[13] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: user attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[14] D. Fisher, L. Dorner, and D. Wagner. Short paper: location privacy: user behavior in the field. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 51–56. ACM, 2012.

[15] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proc. of the 5th Intl. Conf. on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[16] M. Harbach, M. Hettig, S. Weber, and M. Smith. Using personal examples to improve risk communication for security & privacy decisions. In *Proc. of the 32nd Annual ACM Conf. on Human Factors in Computing Systems*, CHI '14, pages 2647–2656, New York, NY, USA, 2014. ACM.

[17] M. Harbach, E. von Zezschwitz, A. Fichtner, A. De Luca, and M. Smith. It'sa hard lock life: A field study of smartphone (un) locking behavior and risk perception. In *Symposium on Usable Privacy and Security (SOUPS)*, 2014.

[18] 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 *Proc. of the ACM Conf. on Comp. and Comm. Sec.*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[19] J. Jeon, K. K. Micinski, J. A. Vaughan, N. Reddy, Y. Zhu, J. S. Foster, and T. Millstein. Dr. android and mr. hide: Fine-grained security policies on unmodified android. 2011.

[20] J. Jung, S. Han, and D. Wetherall. Short paper: Enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 45–50, New York, NY, USA, 2012. ACM.

[21] P. G. Kelley, L. F. Cranor, and N. Sadeh. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3393–3402, New York, NY, USA, 2013. ACM.

[22] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. 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, pages 501–510, New York, NY, USA, 2012. ACM.

[23] B. Liu, J. Lin, and N. Sadeh. Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help? In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 201–212, New York, NY, USA, 2014. ACM.

[24] N. K. Malhotra, S. S. Kim, and J. Agarwal. Internet Users' Information Privacy Concerns (IUIPC): The Construct, The Scale, and A Causal Model. *Information Systems Research*, 15(4):336–355, December 2004.

[25] 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 *Proceedings of First International Workshop on Sensing and Big Data Mining*, pages 1–6. ACM, 2013.

[26] L. H. Newman. Android users won't drop money on just any app. *Slate*, 2014.

[27] J. Nielsen. Response times: The 3 important limits. *Nielsen Norman Group*, 1993.

[28] H. Nissenbaum. Privacy as contextual integrity. *Washington Law Review*, 79:119, February 2004.

[29] Path. We are sorry. February 8 2012. Accessed: January 17, 2016.

[30] J. L. B. L. N. Sadeh and J. I. Hong. Modeling users mobile app privacy preferences: Restoring usability in a sea of permission settings. In *Symposium on Usable Privacy and Security (SOUPS)*, 2014.

[31] B. Shebaro, O. Oluwatimi, D. Midi, and E. Bertino. Identidroid: Android can finally wear its anonymous suit. *Trans. Data Privacy*, 7(1):27–50, Apr. 2014.

[32] I. Shklovski, S. D. Mainwaring, H. H. Skúladóttir, and H. Borgthorsson. Leakiness and creepiness in app space: Perceptions of privacy and mobile app use. In *Proc. of the 32nd Ann. ACM Conf. on Human Factors in Computing Systems*, CHI '14, pages 2347–2356, New York, NY, USA, 2014. ACM.

[33] G. Sims. Google play store vs the app store: by the numbers (2015). *Android Authority*, 2015.

[34] C. Thompson, M. Johnson, S. Egelman, D. Wagner, and J. King. When it's better to ask forgiveness than get permission: Attribution mechanisms for smartphone resources. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, SOUPS '13, pages 1:1–1:14, New York, NY, USA, 2013. ACM.

[35] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, Washington, D.C., Aug. 2015. USENIX Association.

[36] H. Zhu, H. Xiong, Y. Ge, and E. Chen. Mobile app recommendations with security and privacy awareness. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 951–960, New York, NY, USA, 2014. ACM.

A Appendix

Application / Permission	**Peak (ms)**	**Avg. (ms)**
com.facebook.katana	213.88	956.97
ACCESS_NETWORK_STATE		
com.facebook.orca	334.78	1146.05
ACCESS_NETWORK_STATE		
com.google.android.apps.maps	247.89	624.61
ACCESS_NETWORK_STATE		
com.google.process.gapps	315.31	315.31
AUTHENTICATE_ACCOUNTS		
com.google.process.gapps	898.94	1400.20
WAKE_LOCK		
com.google.process.location	176.11	991.46
WAKE_LOCK		
com.google.process.location	1387.26	1387.26
ACCESS_FINE_LOCATION		
com.google.process.location	373.41	1878.88
GET_ACCOUNTS		
com.google.process.location	1901.91	1901.91
ACCESS_WIFI_STATE		
com.king.farmheroessaga	284.02	731.27
ACCESS_NETWORK_STATE		
com.pandora.android	541.37	541.37
ACCESS_NETWORK_STATE		
com.taptu.streams	1746.36	1746.36
ACCESS_NETWORK_STATE		

Table 11: The application/permission combinations that needed to be rate limited during the study. The last two columns show the fastest interval recorded and the average of all the intervals recorded before rate-limiting.

Application / Permission	**Peak (ms)**	**Avg. (ms)**
com.facebook.katana ACCESS_NETWORK_STATE	213.88	956.97
com.facebook.orca ACCESS_NETWORK_STATE	334.78	1146.05
com.google.android.apps.maps ACCESS_NETWORK_STATE	247.89	624.61
com.google.process.gapps AUTHENTICATE_ACCOUNTS	315.31	315.31
com.google.process.gapps WAKE_LOCK	898.94	1400.20
com.google.process.location WAKE_LOCK	176.11	991.46
com.google.process.location ACCESS_FINE_LOCATION	1387.26	1387.26
com.google.process.location GET_ACCOUNTS	373.41	1878.88
com.google.process.location ACCESS_WIFI_STATE	1901.91	1901.91
com.king.farmheroessaga ACCESS_NETWORK_STATE	284.02	731.27
com.pandora.android ACCESS_NETWORK_STATE	541.37	541.37
com.taptu.streams ACCESS_NETWORK_STATE	1746.36	1746.36

Table 12: application/permission combinations that needed to be rate limited during the study. The last two columns show the fastest interval recorded and the average of all the intervals recorded before rate-limiting.

Application	**Effect Size**	**p-value**
Google Inbox	-0.142	<0.04
Weather	-0.189	<0.005
Pacprocessor (In-built Proxy Service)	0.142	<0.03
Twitter	-0.139	<0.04
Google Books	0.141	<0.04
Google Videos (In-built Video player)	0.153	<0.01

Table 13: Applications that had a statistically significant correlation with user denial rate, evaluated through logistic mixed-effect regression. A positive effect size indicates that more usage of the application led to a higher denial rate, while a negative effect size indicates that more usage led to a higher allow rate.

File Containing Change	**Line No.**	**Explanation**
PreferenceUtils.java	47	developer options enabled
SecuritySettings.java	569,588	security settings opened/closed
SecuritySettings.java	695	security settings changed
NfcEnabler.java	108,123	NFC is toggled by user
NfcEnabler.java	143	NFC state is changed by system
LocationSettingsBase.java	115	location mode changed
LocationSettings.java	114,150	location settings opened/closed
DevelopmentSettings.java	1569	developer options enabled
ChooseLockGeneric.java	455	record when user changed lock
NfcService.java	509	NFC initial on/off logged
InboundSmsHandler.java	486	Tell if they're getting 2FA SMS (eg. Gmail)
ActivityManagerService.java	11518	record initial account data
ActivityManagerService.java	6923	sensitive request made
ActivityManagerService.java	18181	application visibility
PowerManagerService.java	1079	screen times out
PowerManagerService.java	1100	user locks screen with power button
PowerManagerService.java	1121	app turns off screen
ReservoirSampler.java	211,324	how long people look at our prompt
ReservoirSampler.java	242,271	whether they allow or deny our prompt
ActivityRecord.java	997	time spent on activity
KeyguardPatternView.java	222	if pattern lock,security type, length of lock
KeyguardAbsView.java	112	pin/password lock, security type, length of lock
AudioService.java	1911,1933	ringer mode change (vibrate, silent, loud)
AudioService.java	2540,2562	speaker phone toggled
AudioService.java	3543,3566	headphones connected
AudioService.java	1829	microphone muted
ImageView.java	422	type of link visited in Chrome
CallLog.java	457	how long people talk on phone
CameraCaptureSessionImpl.java	163,204	picture taken using new camera API
Camera.java	1470	picture taken with old camera API
Camera.java	1122	preview frames streamed to some app
ContentResolver.java	634	picture media store accessed
NotificationManager.java	248,286,325	How long it took for them to click through notification
NotificationManager.java	168	When notification first appears

Table 14: This table contains all of the instrumentation we added. The files are all part of the Android platform, except for `ReservoirSampler.java`, which is a class we added. The explanation covers the main idea for each instrumentation point, but we collect hundreds of unique events, which are omitted from this table for space reasons.