Visual Configuration of Mobile Privacy Policies

Abdulbaki Aydin[1⋆], David Piorkowski[2], Omer Tripp[2⋆⋆], Pietro Ferrara[2⋆⋆⋆], and Marco Pistoia[2]

[1] University of California, Santa Barbara, USA
baki@cs.ucsb.edu,
[2] IBM T.J. Watson Research Center, New York, USA
pistoia@us.ibm.com

Abstract. Mobile applications often require access to private user information, such as the user or device ID, the location or the contact list. Usage of such data varies across different applications. A notable example is advertising. For contextual advertising, some applications release precise data, such as the user's exact address, while other applications release only the user's country. Another dimension is the user. Some users are more privacy demanding than others. Existing solutions for privacy enforcement are neither *app-* nor *user-* sensitive, instead performing general tracking of private data into release points like the Internet. The main contribution of this paper is in refining privacy enforcement by letting the user configure privacy preferences through a visual interface that captures the application's screens enriched with privacy-relevant information. We demonstrate the efficacy of our approach w.r.t. advertising and analytics, which are the main (third-party) consumers of private user information. We have implemented our approach for Android as the VisiDroid system. We demonstrate VisiDroid's efficacy via both quantitative and qualitative experiments involving top-popular Google Play apps. Our experiments include objective metrics, such as the average number of configuration actions per app, as well as a user study to validate the usability of VisiDroid.

1 Introduction

The mobile era is marked by contextual and user-sensitive functionality. Notable examples include location-aware apps and services (browsing, advertising, and more); social features (e.g. in gaming and navigation apps); as well as personalization capabilities (often based on analysis of text, audio or video content). These and other similar features are all based on access to, and use of, personal user information.

⋆ The author performed the research leading to this paper while working at IBM Research as an intern.

⋆⋆ The author performed the research leading to this paper while at IBM Research, but is currently affiliated with Google, Inc.

⋆⋆⋆ The author performed the research leading to this paper while at IBM Research, but is currently affiliated with Julia, S.R.L.

In some cases the user is conscious of the privacy/functionality tradeoff, and can sometimes even disable (or refrain from enabling) privacy-threatening functionality. Unfortunately, there are also common cases wherein users, and sometimes even developers, are in the dark regarding the contexts where, and the extent to which, their private information is exploited.

This worrisome situation links back to the design of mobile platforms. The two major platforms, Android and iOS, both mediate access to private information via a permission model. Permissions govern access to designated resources, such as the contact list or GPS system. In Android, prior to API level 23 permissions were managed at install time. Now both Android and iOS either grant or deny a given permission by seeking user approval upon first access to the respective resource. In both cases, permissions apply globally across all contexts and usage scenarios, and so for example usage of the user's location for navigation cannot be distinguished from its usage for contextual advertising.

Scope and Threat Model. This paper is informed by the need to create a *usable* interface for *end users* to understand and configure how their private information is utilized. Within this general scope, we take a first step in targeting the release of private data related to contextual advertising, as well as UI/UX (aka usability) and navigational analytics. Advertising is the most popular Android monetization model. Analytics is the gateway to user experience. As such, these are the most prominent third-party consumers of private data.

In line with past research on user privacy [25], our assumed threat model concerns authentic (rather than malicious) mobile applications. Such applications, in contrast with malware, are aimed at executing their declared functionality. At the same time, extraneous behaviors, such as advertising and analytics, may access private information and share it with third parties without explicit authorization. Given our focus on these clients, we define release of private user information to remote advertising/analytics websites as a potential *privacy threat*, and concentrate on this category in particular.

Blocking such releases completely is not an acceptable solution. First, for advertising, this would disrupt the Android monetization model that currently allows users to download apps for free. Also, certain users actually take interest in ad content.[3] Analytics data is of immense value to customize and improve user experience. Hence, our goal is to create a method to configure, rather than blindly suppress, usage of private information by analytics/advertising engines.

One challenge, which we later illustrate, is that private data is seldom embedded into ad/analytics web messages in clear form. For security as well as performance reasons, such messages often carry encoded and sometimes even encrypted data. This obviates naive attempts to detect (and anonymize) sensitive data in ad/analytics messages (e.g. via a man-in-the-middle proxy).

Our Approach. We address the gap between user experience, which is typically far removed from notions like code-level permissions, and the need to tailor privacy enforcement according to the preferences and sensitivities of the user at

[3] `http://www.adweek.com/socialtimes/study-kenshoo-mobile-app-advertising-trends/617293`

hand. We bridge this gap via a visual configuration interface. This critical aspect of the enforcement system is complementary to existing solutions, which typically enforce privacy via (user-insensitive) information-flow tracking [5,9,11,25,28], in specializing the system per the given user. In our approach, usages of private information are rendered onto the app UI, which resonates directly with user experience. The user can then visually configure privacy preferences atop the UI, which are subsequently enforced by instrumenting the app. This creates a separation of concerns: The user is provided with a visual interface to reason about — and express — privacy concerns, while the enforcement system benefits from a tight privacy policy, which contributes to both performance and accuracy.

In our concrete system, we instantiate the above workflow — as a first step — w.r.t. privacy threats due to contextual advertising and UI/UX analytics. The user is presented with annotated UI screenshots that reflect, atop advertising widgets and analytics-empowered screens, the private information. The user can then configure constraints on the context, which the system enforces by anonymizing respective private fields before these are sent to the remote server.

Contributions. This paper makes the following principal contributions:

1. Usable privacy enforcement: We take a first step toward a general solution to the challenge of usable configuration of privacy enforcement systems (Sections 2.2 and 3.2). In our solution, which focuses on advertising and analytics, usage of private data is reported atop the app UI, enabling seamless configuration of privacy preferences.
2. System design: We describe and illustrate the architecture and algorithms underlying our solution (Sections 3—4). These include offline analyses as well as per-user code rewriting, where — for portability and general applicability — we do not require a custom platform build.
3. Implementation and evaluation: We have implemented our solution as the VisiDroid system for Android. We report on quantitative measurements as well as a user study that we have conducted, which validate the viability of our approach and the usability of VisiDroid. Some of the experimental data is of wider applicability, characterizing which fields typically flow into advertising/analytics servers and how often (Section 5).

2 Overview

The VisiDroid workflow begins with offline analysis of the target application A; the user then performs interactive policy configuration; finally, an instrumented app that enforces the user privacy settings is emitted. We discuss these steps, which are visualized in Figure 1, in more detail below with reference to Listings 1.1 and 1.2 that are based on the popular photo-blogging theCHIVE app (package name: `com.thechive`). theCHIVE is a highly popular Android photo-blogging app.[4] It utilizes the location and device ID for advertising, where

[4] `https://play.google.com/store/apps/details?id=com.thechive`

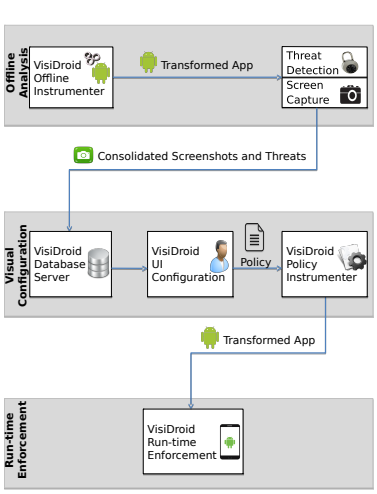

Fig. 1: VisiDroid architecture

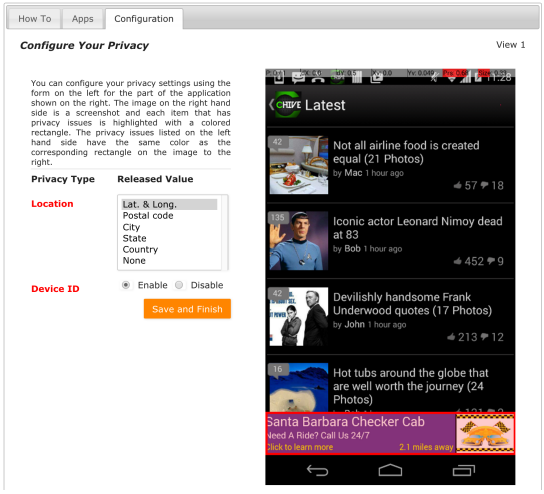

Fig. 2: The VisiDroid configuration interface

the recommended practice to instead use a coarsening of the precise location and the Android ID [23].

Listing 1.1: Transformed code for offline analysis

```
1  adv = findViewById(id);
2  did = getSysService("phone")
       .getDeviceId();
3  LOGSOURCE(DEVICE_ID, did);
4  adv.setKeywords("User:"+did);
5  LOGSINK(adv); //adv.loadAd();
6  // method declarations
7  LOGSINK(MoPubView adv) {
8   adReq = adv.getAdRequest();
9   LOG(adReq); CAPTURESCREEN();
10  adv.loadAd(); }
```

Listing 1.2: Transformed code for runtime enforcement

```
1  adv = findViewById(id);
2  did = getSysService("phone")
       .getDeviceId();
3  adv.setKeywords("User:"+did);
4  LOADAD(adv); //adv.loadAd();
5  // method declarations
6  LOADAD(MoPubView adv) {
7   Mdid = MOCK(DEVICE_ID, did);
8   adv.setKeywords("User:"+Mdid);
9   adv.loadAd();
10 }
```

2.1 Phase I: Offline Analysis

Offline analysis is conducted once per application A. It mines the contexts associated with UI objects in A. This is done in two steps. The first step is to track flow of private data starting at *privacy sources*, such as `getDeviceId()` and `getLastKnownLocation()`. The second step, if the data reaches an outgoing ad/analytics request, is to match the message against one or more UI objects.

For high accuracy, offline tracking is done dynamically. The app is instrumented to enable runtime tracking (lines 3, 5 in Listing 1.1). The instrumented

version is then exercised (either manually or automatically) on a designated device that is set up with mock user information. In the case of theCHIVE, the location and device ID are tracked into their respective `MoPubView` setters. Next, the request/response matching is established when `loadAd()` is invoked. Internally, this method serializes the `MoPubView` object as an ad request, sends the request, and renders the ad content received in response to the screen.

Another aspect of offline analysis is to persist programmatic and geometric information about the UI objects to enable specification and enforcement of privacy restrictions on the different screens/widgets across different runs. This is done via the *adb* library, which provides APIs to capture stateful screenshots with XML-structured data (line 9 in Listing 1.1).

Automated crawling is a challenging problem that has received considerable attention [4,12,18]. To simplify and focus the contributions of the current paper, we neither developed nor used any of the available solutions. Instead, we crawled the subject apps manually, which was neither difficult nor burdensome. We detail our methodology for manual crawling in Section 5.

2.2 Phase II: Visual Configuration

Offline analysis yields stateful app screenshots that visually link advertising/-analytics libraries to the private information they consume. These artifacts are uploaded to the VisiDroid server. A user can then perform visual configuration with the VisiDroid client application (i.e., website). The VisiDroid client application displays the screenshots from the server, as illustrated in Figure 2. The user is presented with the different privacy threats visually, with the available configuration options on the left and a screenshot of the view under configuration on the right. The user can then choose the granularity of data release (or prevent release altogether using mock data; *Disable*, *None*) on a per-widget basis. For example, for the location data flowing into `MoPubView`, the user may permit state and city information but not their exact address.

2.3 Phase III: Enforcement

The third and final step is to rewrite the app per the customized policy, as configured by the user. This is done via code instrumentation, which imposes anonymized values on the private fields constrained by the user. We illustrate the transformation in Listing 1.2 (the `MOCK` call at line 7). The actual values are substituted with their anonymized counterparts at the very last point before the ad/analytics request is discharged. This is to ensure that there are no side effects w.r.t. other functionality that consumes the values, which is confirmed to be a problem in practice [11].

VisiDroid imposes the anonymized values on the outgoing request via built-in interface methods. In our running example, these are the setters defined by the respective widget object. The `loadAd()` method is called inside `LOADAD` at line 9 in Listing 1.2, such that prior to the original code, setter methods exposed by `MoPubView` are invoked with the anonymized location and device-ID fields.

2.4 Scope and Limitations

In its current form, VisiDroid is designed to address privacy threats in contextual advertising as well as UI/UX, or navigational, analytics. Linking privacy threats in other categories to user experience is our goal in future research.

VisiDroid exploits commonalities across different libraries for efficiency. An example of this, noted above, is the use of built-in setter APIs to overwrite private fields with their anonymized counterparts. Leveraging knowledge of the different ad/analytics servers simplifies the VisiDroid architecture, as well as optimizes its performance and accuracy. At the same time, new servers need to be welded into VisiDroid, which is a limitation of our design. Though this process is facilitated by a declarative interface exposed by VisiDroid, it still needs to be performed for every new server.

3 Offline Analysis

Offline analysis is performed once per (all users of an) app. It takes as input the app, and produces as its output — following a crawling session that explores different app views and states — a visual representation of advertising -/analytics-related privacy threats. For this, VisiDroid integrates between privacy-related information flows and screen captures, which is the focus of this section.

3.1 Detection of Privacy Threats

A common method to track flow of private information across a mobile system is via taint tracking [5]. Doing so via app instrumentation is highly nontrivial, and leads to prohibitive slowdown, and so a common solution is to customize the platform. This limits the portability and maintainability of the detection system. Instead, inspired by the BayesDroid system [25], VisiDroid performs *value-based* threat detection. Intuitively, the idea is to record private fields as they originate, and compare them — using similarity metrics — against values about to be released. If the level of similarity is significant, then a potential threat has been detected. Specifically, VisiDroid utilizes the Levenshtein metric to assess value similarity. Informally, $\mathrm{lev}(|a|, |b|)$ is the minimum number of single-character edits — insertion, deletion or substitution — needed to transform string a into string b. The significance of using "fuzzy" similarity analysis, rather than precise matching, stems from cases where the value is reformatted by the app (e.g., accepting email address `john.doe@gmail.com` as input, then releasing `john` and `doe` separately as the first and last names). See the BayesDroid paper [25] for more discussion. Compared to statement-level information-flow (or taint) tracking, this approach is lightweight. It further enables characterization of the type and amount of information about to be released per the value at hand (e.g., if the value matches the street name out of the full address).

VisiDroid is equipped with a diversified set of privacy sources, including built-in platform APIs to obtain the user's location (e.g. the `getLastKnownLocation()` method in `LocationManager`), device identifiers (e.g. the `getSimSerialNumber()`

method in `TelephonyManager`), etc; APIs exposed by social apps (e.g. the Facebook graph interface); and user inputs received through the UI, which are obtained via platform APIs (e.g. `EditText.getText()`).

VisiDroid currently accounts for the five top-popular advertising libraries (in non-gaming apps),[5] as well as two of the most popular UI/UX analytics libraries.[6] These are Google AdMob, MoPub, Millenial Media, Amazon Ads and Ad Marvel, and Google Analytics and Adobe Analytics, respectively. Extending VisiDroid with additional libraries is straightforward via a declarative specification interface.

While informed by BayesDroid, the detection technique of VisiDroid is slightly different. First, rather than accounting for a single release flow, VisiDroid has to monitor — in the case of ad requests — the composition of (i) the outgoing flow from the app to the server (the sink being the request) and (ii) the incoming flow from the server back to the app (the sink being the widget). Treating both as a single composed flow — as in BayesDroid — is futile, as the ad content reaching the widget sink is rich media rather than text that can be searched for matches against the private field.

Instead, VisiDroid leverages a predefined specification of how different popular ad libraries load remote content. In the example in Listing 1.1, for instance, `loadAd` is the API provided by MoPub to send out the ad request and render the resulting content to the UI. Inspecting the values stored in the widget receiving the `loadAd` call as it is entered establishes which private fields (if any) are about to be released to the remote server. The widget is then annotated with these private fields, which accounts for the incoming flow.

An advantage of performing value similarity analysis against the (fields of the) widget object, rather than the HTTP request, is that at that point the values are still stored in clear form. Hashing, encryption and other transformations all take place downstream, as the widget's state is serialized into an HTTP message. This enables direct (and efficient) similarity analysis, without the need to account neither for standard nor for custom transformations (unlike BayesDroid).

Another important point is that our offline analysis detects threats at the subfield level, also accounting for transformations. If for instance the location is transformed into a street address, but only the city (sub)field is passed to the third-party library, then the analysis would report that subfield alone rather than the entire location/address.

To achieve this robustness, the VisiDroid analysis proactively computes for certain private fields, and in particular the user's location and date of birth, other common equivalent representations as well as derivative information. For birthday, these include the user's age as well as other date patterns including month/day, round trip, short time and long time. For the location, the corresponding address is computed. VisiDroid computes the transformations via standard APIs (like the `Geocoder.getFromLocation` method for location/address transformation).

[5] `http://www.appbrain.com/stats/libraries/ad`
[6] `http://www.appbrain.com/stats/libraries/dev`

Algorithm 1 Geometric representation of ad widgets

1: **procedure** GEOMETRICREP(s : app screen)
2: $ids \longleftarrow [\,]$, $A \longleftarrow$ GETFOCUSEDACTIVITY(S), $layout \longleftarrow$ GETLAYOUT(A)
3: **for all** node $n \in layout$ **do**
4: **if** n is an ad widget **then**
5: $C \longleftarrow$ extract bounds, $ids \longleftarrow ids \cdot [n \mapsto (A, C)]$
6: **return** ids

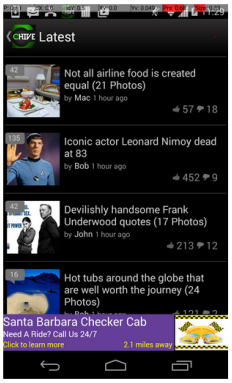

Fig. 3: UI capture with contextual ad

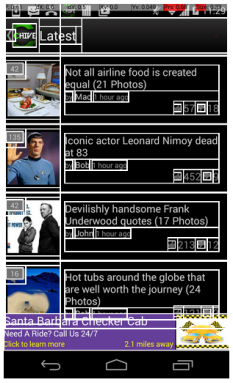

Fig. 4: UI structure of Fig. 3

3.2 Screen Capturing

Interleaved into threat detection during the crawling process is the collection of UI screens. This is done via built-in *adb* utilities, and in particular the *screencap*, *dumpsys* and *uiautomator* shell commands.

At a given instant in the crawling process, the steps described in Algorithm 1 are executed. First, the current screen is captured and the XML layout of the screen, providing rich hierarchical information regarding the UI elements and their geometrical representation, is derived with the help of the *adb* utilities (line 2). Next, for each XML node n, representing a distinct UI element, we read its coordinates as the `bounds` attribute (line 5), and compute an identity for it as a pair (A, C) consisting of the the parent activity A and coordinates C (line 5). Figures 3 and 4 presents example screen capture from theCHIVE and its corresponding decomposition into UI elements, as given by the XML layout. Note in particular the precise boundaries surrounding the ad widget, which is disambiguated from the other content on the screen.

The geometric representation of a UI object is sensitive to the display properties, varying across users and devices, which would seemingly pose a challenge to VISIDROID. Observe, however, that VISIDROID utilizes geometric information only for visual configuration. This is done atop offline screen captures, and not the user's device, which offsets the risk of distorting the bounds for a UI object.

Yet there is another aspect of bounds extraction that is subtle. Computing the boundaries of a UI element is straightforward as long as the element is known to occupy a fixed region on the screen. Sometimes, however, UI widgets — in particular ads and sometimes also views enriched with analytics — are embedded

into scrolling banners. In such cases, we fix the boundaries of the widget to be those of a non-scrolling parent element.

4 Configuration and Enforcement

Configuration. Before installing an app A, the user performs visual configuration of the privacy policy for A via the VisiDroid client. The resulting policy is then sent to the server as input to the VisiDroid instrumentation agent, which transforms A accordingly, and makes the new version available for the user to download and install. We have chosen this design, rather than downloading all the artifacts to the device and performing the configuration locally, to lower network and (device) storage costs.

As illustrated in Figure 2, the user is presented with the screen captures due to offline analysis one by one. For each screen, all elements linked to privacy threats are highlighted, such that the user can authorize which, if any, of the private fields to release. As discussed in Section 3.1, the user is shown the subfields that are actually released (e.g., city rather full address or location). This is crucial, as otherwise the user may form a wrong impression of the app's behavior and/or perform redundant configuration (e.g., constraining location release even though only the city is released).

Code Instrumentation. Upon completion of the policy configuration, the resultant constraints are discharged to the VisiDroid instrumentation agent (hosted on the VisiDroid server). The agent rewrites the app accordingly, as illustrated in Figure 1.2. Code instrumentation aims to mock values only for third parties without any crashes and side effects.

We have found that the idealized assumption regarding app/library insulation often holds in practice. Advertising and analytics libraries often consume the context from the app, and then operate on their own as an independent third party in obtaining and acting upon the contextual content. Knowing that, VisiDroid synthesizes the mock value immediately before setting it on the library, as shown at line 9 in Listing 1.2. The original value is preserved and propagated beyond the setter call, such that the mock value only flows into the library. VisiDroid is equipped with specification of formatting and logical constraints on privacy related data. It synthesizes a mock data that is consistent with the original. Hence, there would not be any side effects due to enforcement beyond potential changes to ad content and/or analytics data.

VisiDroid ensures that no private field is released without user authorization. It (statically) scans through the entire app code base for occurrences of advertising/analytics APIs (like `loadAd()`). For any such call that has not already been handled, and also for private fields not configured by the user w.r.t. handled calls (since flow of such fields has not been observed during offline analysis), VisiDroid inserts instrumentation code immediately prior to the call to trigger a warning at runtime. This ensures that if an unhandled privacy-relevant call is made, then the user has the opportunity to either authorize or reject the communication with the respective library.

5 Experimental Evaluation

We demonstrate VisiDroid's efficacy via both quantitative and qualitative experiments involving top-popular Google Play apps. We explain how we designed our experiments in the following.

Benchmarks. Our benchmark selection methodology consisted of several steps. We started from the 60 top-popular apps in every Google Play category, except games (as we explain shortly), for a total of 25 categories and 1,462 apps. The ideal number of apps should have been 1,500, but we experienced some technical issues downloading certain apps.

We skipped the games category for two reasons: First, the monetization strategy of gaming apps is mostly in-app purchases, where ad content (if present) mostly refers to other games. Second, from a technical standpoint, gaming apps are often equipped with large media files. Since the games category has 18 subcategories, the download process would have become very heavy with the addition of 1,200 big apps. Gaming apps are left for future research.

We pruned the initial set of apps via a script that searches through the app code for usage of ad and analytics libraries (supported by VisiDroid), and in particular initialization code for the libraries. This resulted in 364 apps (25% of all apps), which we further filtered using a similar script that scans for contextual advertising (i.e. occurrence of setter APIs like `setAge(...)`, `setBirthday(...)`, etc). 126 apps survived this filter.

Experimental methodology. To uncover UI screens and releases of private information, we exercised each of the 126 applications manually, spending approximately 5 minutes per app. For exhaustive crawling, as well as authentication via social apps, we created mock Facebook accounts and user profiles.

For unbiased and consistent overhead measurement, we automated — using the *a*db toolset — each of the manual interaction sequences. For statistical significance, we repeated overhead measurements 20 times per app. Running times, divided into user and system times, were derived from the device via built-in platform APIs.

Hardware setup. We executed all the apps on a Google Nexus 4 phone, running version 4.4.4 of the Android platform, with a Qualcomm Snapdragon S4 Pro 1.5GHz processor and 2 GB of memory. We used a single physical device per all apps and all experiments.

Experimental Results. VisiDroid's offline analysis identifies 31 apps that release private data to either advertising parties or analytics engines or both. These are listed in Table 1, which also specifies the involved libraries (second column), the released private fields (third column), as well as the source(s) of the private fields (fourth column). The privacy threats identified by VisiDroid are distributed as follows: full location – 35%; postal code – 12%; other location fields – 25%; birthday and/or age – 9%; gender – 11%; and other threats – 9%. Of special interest is that theCHIVE releases the device ID, and Job Search the build serial. These identifiers are unique, enabling identification of the device and thus also the user, though utilizing such identifiers is considered a bad practice. These statistics, and the data underlying them in Table 1, shows that different

apps release different private fields, and at different granularities, to third-party websites. This, combined with the fact that different users have different privacy concerns, emphasizes the need for a configurable privacy policy.

For unbiased measurement of configuration cost, we first fixed the privacy restrictions prior to the offline runs. We restrict location to city, personal info to only age and gender, and device info to only operator name. These restrictions reflect our own perspective, whereby it would be legitimate to use coarsened private information for advertising/analytics but not fully precise data. Equipped with the restrictions defined (i.e., policy), we configured each of the 31 applications from Table 1 via the VisiDroid client. We quantify the involved effort by counting (i) *configuration forms*, where each such form is linked to one or more ad widgets or view on some screen; (ii) *configuration items*, where we refer to the total number of selections that the user can potentially make across all screens and forms (there are e.g. 2 configuration items in the form in Figure 2); and (iii) *configuration actions*, where an action is an actual selection we made guided by the restrictions. The results are summarized in the last 3 columns of Table 1. In total, across all apps, 6.61 configuration items need to be reviewed and 4.94 configuration actions are required on average. Aside from myHomework and MeetMe, which define 52 and 21 configuration items, respectively, all the rest of the apps require a total of 0-16 configuration item reviews and 0-16 configuration actions. The user performs these actions once (modulo changes to policy), and so we conclude that the user effort demanded by VisiDroid is tolerable if not low.

Although much of the VisiDroid workflow is automated, determining what privacy permissions to set requires users to interact with the web-application interface of VisiDroid. Prior work has already shown that mobile device users are aware of privacy concerns when installing mobile applications [7,8,10,14,15,17,22]. Therefore, for the usability study, our goal was not to confirm users' concerns, but rather to validate that users can change privacy settings easily and correctly in the user-facing client application. To this end, we ran a usability study based on standard approaches.

Usability Study Design. The usability study consisted of 7 male and 3 female participants whose ages ranged from 30 to 69 and had 1 to 8 years of experience (mean: 3.0 years) using mobile devices. Each participant was randomly assigned to three of five possible app configuration tasks. In each configuration task, they were asked to modify the privacy data released by the app such that it matched the permissions given to them with the task, such as the following example: "*Set the granularity of the location released by the advertising to None.*" We gave participants the configuration instead to test all the possible granularities of configuration options.

Prior to each session, participants filled out a brief background questionnaire. Then, participants were asked to complete each of the three tasks until they stated that they were done. No instructions were given on how to complete the configuration task, but help was available within the client application if needed.

Table 1: Applications with privacy threats ('D'=device; 'inp'=user input; 'FB'=Facebook; 'UIE'=UI element; 'pcode'=postal code; 'GAn'=Google Analytics; 'GAds'=Google Ads;'AA'=Adobe Analytics; 'MM'=Millennial Media)

Application	Ad libs	Private fields	Sources	Config. forms	Config. items	Action items
1Weather	GAn	main views, settings	UIE	10	10	10
AccuWeather	GAn	main views	UIE	4	4	4
AroundMe	MoPub	long/lat	D	1	1	1
Brightest Flashlight	MM	long/lat	D	1	1	1
Cars.com	GAds	city, sub state, pcode	D, inp	6	6	6
Checkout 51	GAn	login (+FB), browsing, account	UIE	8	8	8
com.idemfactor	MoPub	long/lat	D	1	1	1
Daily Workouts	MoPub	long/lat	D	3	3	3
Endomondo	GAds	long/lat, birthday, gender	D, inp, FB	8	16	16
Final Countdown	MoPub Amazon Ads	first/last name, gender	FB	2	4	2
FlightStats	GAds	country, mcc	D	2	4	0
Fox News	AA	browsing	UIE	6	7	7
GasBuddy	GAds	long/lat	D	2	2	2
GrubHub	GAn	main views	UIE	6	6	6
Horoscope	GAds	long/lat, birthday, gender	D, inp, FB	5	15	10
JackThreads	GAn	main views	UIE	5	5	5
KAYAK	GAds	long/lat	Dice	3	3	3
KBB	GAds	pcode	D	6	8	8
MeetMe	AdMarvel	long/lat, pcode, country, mcc, age, gender	D, inp, FB	7	21	7
myHomework	GAds Amazon Ads MM	long/lat, birthday, age, gender	D, inp, FB	13	52	26
Photobucket	AdMarvel	long/lat, pcode, gender	D, inp, FB	1	2	1
Radar Express	Amazon Ads	long/lat	D	1	1	1
Scanner Radio	MoPub MM	long/lat	D	1	1	1
Snagajob	GAds	long/lat, build serial	D	1	2	2
theCHIVE	MoPub MM	D ID, long/lat	D	2	4	4
theScore	MoPub	long/lat	D	5	5	5
Urbanspoon	GAds	long/lat	D	1	1	1
WeatherBug	AdMarvel GAn	long/lat, pcode, city, state, operator name, main view	D, UIE	4	5	5
Weather Kitty	GAds	long/lat	D	1	1	1
Whitepages	AdMarvel	long/lat, pcode, address #, city, sub state, state	D	5	5	5
Yelp	GAds	pcode, city, country	D	1	1	1

After the three tasks, each participant was asked to select and rank words that best represented their experience with VisiDroid. The words were taken from the Microsoft Desirability Toolkit [1]. After all, we asked the participant: "Could you provide additional details about why you selected <word>?".

We measured participants' ability to finish the task via three metrics. *Completeness* is the measure of how many participants were able to finish the task and generate any configuration. *Correctness* is a measure of how many participants were able to complete each task correctly. *Errors* occur each time a setting was misconfigured, the wrong application was selected, or an action unrelated to the task at hand was performed. A participant can recover from errors and still complete a given task correctly.

Usability Study Results. All 30 tasks were evaluated to be complete and correct, and 26 of those tasks were completed error-free. We observed six errors over four tasks. Participants 6 and 10 each made errors in their first task, where they configured the wrong app (and saved the configuration). Participant 6 did

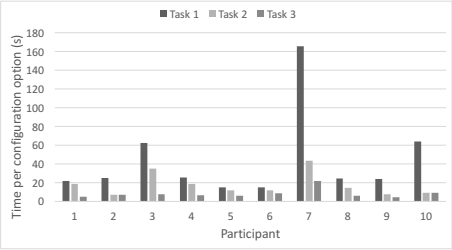

Fig. 5: Participants' average time per configuration

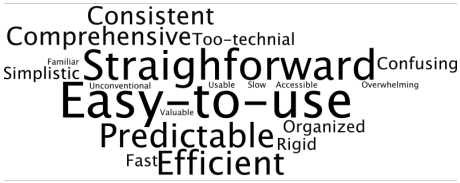

Fig. 6: Responses for the desirability evaluation

Table 2: Responses to the words selected from the desirability toolkit.

Selected words	**Participant Responses**
Easy-to-use	P1: "It seemed to be fairly obvious what to select from the given tasks." P3: "It was very easily understandable what the controls meant and what it was that I was doing." P4: "There is no confusion about what's happening. It's pretty straightforward and easy to use." P6: "[It's] similar to the existing applications. So I don't need to think a lot."
Straightforward	P2: "There's just a small number of things that you can select over on the left." P4: "It's pretty clear that it's about the [privacy threats] coming out of the featured views." P6: "Because it's highly predictable, so I know what I'm doing."
Efficient	P1: "It seemed to be pretty quick to go in there an select them [the configuration options]." P3: "It was very obvious what I selecting and what I was revealing." P6: "I guess the setting options are not so many, so I can know what I'm doing so easily."
Predictable	P6: "After finishing the first task, I know what I was expecting for the others." P8: "I can predict what's going to happen on the next page. So I think the learning path is just very short." P9: "I knew what to expect when I was going into the different apps and how to use the different settings."

it three times. Participant 10 did it once. Participants 7 and 8 each had one error, wherein they clicked on an item that had a broken link. Despite these errors, all participants were able to return to the correct configuration page and successfully finish all the configuration tasks.

Efficiency. As participants became more familiar with the interface, they were able to complete their tasks more quickly. Figure 5 indicates, across all participants, that the average time to configure a single option decreased during Task 2 and yet again during Task 3. The mean time to configure one option decreased from 44.2 seconds in Task 1 to 8.1 seconds in Task 3, or approximately a five-fold improvement. This suggests that the learning curve for understanding the interface and how to configure privacy options was sufficiently low to qualify as practical and feasible in real life. Participants' statements from the interview corroborate this hypothesis:

- P5: '...Once a workflow pattern is established, it's the same for all screens.'

- P7: 'After doing a couple of pages of examples, I really didn't have to think much about what was going on there.'

- P9: 'Once I got into the task, especially after the first task, I found the experience to be very consistent.'

The outlier in Figure 5 was Participant 7 for Task 1. Participant 7 spent time in the beginning of Task 1 switching back and forth between reading the instructions available within the application and experimenting with configura-

tions options for other apps before reading the task and completing it. After Task 1, he did not return to the instructions and finished the tasks faster.

User Sentiment. We also collected participants' sentiments about configuring privacy via the desirability toolkit. The word cloud visualization in Figure 6 depicts all the words that participants selected as part of their top-five words during that part of the usability study. The more participants that select a word, the larger it appears in the visualization. The most frequently selected words and the number of times they were selected were: Easy-to-use (6), Straightforward (5), Efficient (4) and Predictable (4). Table 2 provides excerpts from the participants' responses for selecting words in the visualization.

Along with the positive responses, Figure 6 also shows that there is room for improvement. Participants 1 and 8 found VisiDroid *confusing*, expressing that they did not understand the relationship between the released data the the screenshots provided. Participants 8 and 9 selected *too technical*, referring to the instructions in the tool. Participants 9 and 10 found VisiDroid to be *rigid* because there was no way to set permissions once and apply the settings to multiple screens. Participant 4's selection of *overwhelming* referred to the large number of screens to configure for some applications. We shall address this feedback in a future revision.

Enforcement. Lastly, we discuss VisiDroid's runtime enforcement. We focus performance (or overhead) and correctness (i.e. no missed leaks and no side effects). First, for performance, we measured the overhead per the methodology given in Section 5. On average, VisiDroid is responsible for an increase of 2.16% in user time and 0.96% in system time. For 9 out of the 31 apps we recorded a modest decrease in user and/or system times following the enforcement transformation (i.e., negligible if any overhead). The app exhibiting the highest overhead is Car Buying - KBB, which uses an old version of Google Ads. Our support for this version is based heavily on reflective calls as well as serialization/deserialization of objects, which cause the high overhead.

Next, to validate absence of side effects, we executed the automated crawling sessions for each of the apps with and without enforcement. We monitored the sessions to validate that intermediate as well as final app states are the same across both runs (modulo ad content). This was indeed the case for all apps. As an illustration, we detail our experience with GasBuddy. This app utilizes the user's precise location to find nearby gas stations, but at the same time, it also sends the location to ad servers. VisiDroid only mocks the location in the latter scenario. Indeed, relevant gas stations are still displayed, and alongside them, ad content that is not immediately related to the given location.

Finally, to confirm that there are no misses, we created another instrumentation scheme that — beyond applying anonymization transformations — intercepts the internal state of ad widgets and analytics engines. This step, shared in common with offline instrumentation, is to check whether any of the fields classified as sensitive (according to Section 3.1) has unexpectedly reached the remote library. As expected, we were unable to find a single instance of sensitive data reaching a third-party library in its original form. As additional anecdo-

tal evidence, we further interacted with approximately one third of the apps beyond the automated crawling scripts to check whether previously unseen advertising/analytics behaviors would be discovered. Encountering fresh behaviors would have resulted in a runtime warning by VisiDroid, as explained in Section 4. We could not, however, trigger any such warning.

6 Related Work

Tools like MockDroid [2] and LP-Guardian [6] replace actual user data with mock data at the cost of functionality degradation. VisiDroid does the same, though the user is provided with rich context to decide where and how to enable mocking. Also, unlike MockDroid and LP-Guardian, which rely on OS-level customizations, VisiDroid is portable.

Another approach is runtime privacy enforcement. Roesner, *et al.* [20] do so via *access control gadgets*, which are custom gadgets embedded into the app. TaintDroid [5], and other solutions built on top of it [13,21], perform runtime taint tracking. Additional solutions for online monitoring and control of release of private information are AppFence [11] and AppAudit [26], which are similar in spirit to TaintDroid; BayesDroid [25], which VisiDroid utilizes as its core tracking engine; Aurasium [27], which relies on sandboxing; as well as the approach of packet-padding-like mitigation [29] (cf. [3,24]). Unlike all of these solutions, which enforce a general privacy policy, VisiDroid allows privacy enforcement to be customized by the end user through a user-friendly visual interface.

Yet another category of tools is turned towards developers and analysts. The Aquifer framework [19] lets the developer define secrecy restrictions that protect the entire UI workflow defining a user task. AppIntent [28] outputs a sequence of GUI interactions that lead to transmission of sensitive data, thus helping an analyst determine whether that behavior was intended. VisiDroid targets end users rather than developers or analysts.

Finally, there are tools to derive privacy specifications. Lin, *et al.* [16] cluster privacy preferences into a small set of profiles to help users manage their private information. Their approach relies on static code analysis to determine why an app requests a given permission (e.g. advertising vs core functionality). Unlike Lin, *et al.*, VisiDroid provides a specification interface, and not inference capabilities, with the goal of serving the exact preferences of the specific user at hand.

7 Conclusion and Future Work

We have presented VisiDroid, a privacy enforcement system that allows the user to understand and manage usage of private information for advertising/analytics via a accessible interface. In the future, inspired by Lin, *et al.* [16], we intend to explore possibilities for crowd-sourced policies. We also plan to optimize runtime enforcement, e.g. by coarsening low-relevance privacy tags.

References

1. Benedek, J., Miner, T.: Measuring desirability: New methods for evaluating desirability in a usability lab setting. Proceedings of Usability Professionals Association 2003, 8–12 (2002)
2. Beresford, A.R., Rice, A., Skehin, N., Sohan, R.: Mockdroid: Trading privacy for application functionality on smartphones. In: HotMobile '11 (2011)
3. Chen, S., Wang, R., Wang, X., Zhang, K.: Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In: S&P (2010)
4. Choi, W., Necula, G., Sen, K.: Guided gui testing of android apps with minimal restart and approximate learning. In: OOPSLA (2013)
5. Enck, W., Gilbert, P., Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-flow Tracking System for Real-time Privacy Monitoring on Smartphones. In: OSDI (2010)
6. Fawaz, K., Shin, K.G.: Location Privacy Protection for Smartphone Users. In: CCS (2014)
7. Felt, A.P., Egelman, S., Wagner, D.: 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. pp. 33–44. ACM (2012)
8. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android permissions: User attention, comprehension, and behavior. In: Proceedings of the Eighth Symposium on Usable Privacy and Security. p. 3. ACM (2012)
9. Ferrara, P., Tripp, O., Pistoia, M.: MorphDroid: Fine-grained Privacy Verification. In: ACSAC (2015)
10. Fu, B., Lin, J., Li, L., Faloutsos, C., Hong, J., Sadeh, N.: Why people hate your app: Making sense of user feedback in a mobile app store. In: Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 1276–1284. ACM (2013)
11. Hornyack, P., Han, S., Jung, J., Schechter, S.E., Wetherall, D.: These Aren't the Droids You're Looking for: Retrofitting Android to Protect Data from Imperious Applications. In: CCS (2011)
12. Jensen, C., Prasad, M., A. Møller, A.: Automated testing with targeted event sequence generation. In: ISSTA (2013)
13. Jung, J., Han, S., Wetherall, D.: Short Paper: Enhancing Mobile Application Permissions with Run-time Feedback and Constraints. In: SPSM (2012)
14. Kelley, P.G., Cranor, L.F., Sadeh, N.: Privacy as part of the app decision-making process. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 3393–3402. ACM (2013)
15. Khalid, H., Shihab, E., Nagappan, M., Hassan, A.E.: What do mobile app users complain about? IEEE Software 32(3), 70–77 (2015)
16. Lin, J., Liu, B., Sadeh, N.M., Hong, J.I.: Modeling Users' Mobile App Privacy Preferences: Restoring Usability in a Sea of Permission Settings. In: SOUPS (2014)
17. Lin, J., Amini, S., Hong, J.I., Sadeh, N., Lindqvist, J., Zhang, J.: Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In: Proceedings of the 2012 ACM Conference on Ubiquitous Computing. pp. 501–510. ACM (2012)
18. Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: An input generation system for android apps. In: FSE (2013)

19. Nadkarni, A., Enck, W.: Preventing Accidental Data Disclosure in Modern Operating Systems. In: CCS (2013)
20. Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H.J., Cowan, C.: User-driven access control: Rethinking permission granting in modern operating systems. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. pp. 224–238. SP '12 (2012)
21. Schreckling, D., Posegga, J., Köstler, J., Schaff, M.: Kynoid: Real-Time Enforcement of Fine-grained, User-defined, and Data-centric Security Policies for Android. In: WISTP (2012)
22. Shklovski, I., Mainwaring, S.D., Skúladóttir, H.H., Borgthorsson, H.: Leakiness and creepiness in app space: Perceptions of privacy and mobile app use. In: Proceedings of the 32nd annual ACM conference on Human factors in computing systems. pp. 2347–2356. ACM (2014)
23. Stevens, R., Gibler, C., Crussell, J., Erickson, J., Chen, H.: Investigating user privacy in android ad libraries. In: W2SP (2012)
24. Sun, Q., Simon, D.R., Wang, Y., Russell, W., Padmanabhan, V.N., Qiu, L.: Statistical Identification of Encrypted Web Browsing Traffic. In: S&P (2002)
25. Tripp, O., Rubin, J.: A Bayesian Approach to Privacy Enforcement in Smartphones. In: USENIX Security (2014)
26. Xia, M., Gong, L., Lyu, Y., Qi, Z., Liu, X.: Effective real-time android application auditing. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. pp. 899–914 (2015)
27. Xu, R., Saïdi, H., Anderson, R.: Aurasium: Practical Policy Enforcement for Android Applications. In: USENIX Security (2012)
28. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., Wang, X.S.: AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In: CCS (2013)
29. Zhou, X., Demetriou, S., He, D., Naveed, M., Pan, X., Wang, X., Gunter, C.A., Nahrstedt, K.: Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources. In: CCS (2013)