“These Aren’t the Droids You’re Looking For”

Retrofitting Android to Protect Data from Imperious Applications

Peter Hornyack*
pjh@cs.washington.edu

Seungyeop Han*
syhan@cs.washington.edu

Jaeyeon Jung*
jyjung@gmail.com

Stuart Schechter†
stuart.schechter@microsoft.com

David Wetherall*
djw@cs.washington.edu

University of Washington* Microsoft Research†

ABSTRACT

In order to install an Android application, users are commonly required to grant these application both the permission to access information on the device, some of which users may consider private, as well as access the network, which could be used to leak this information. We present two privacy controls to empower users to protect their data from exfiltration by permission-hungry applications:

(1) covertly substituting shadow data in place of data that the user wants to keep private, and

(2) blocking network transmissions that contain data the user made available to the application for on-device use only.

We retrofit the Android operating system to implement these two controls for use with unmodified applications. A key challenge of imposing shadowing and exfiltration blocking on existing applications is that these controls could cause *side effects* that interfere with user-desired functionality. To measure the impact of side effects we develop an automated testing methodology that records the visual output of application executions both with and without privacy controls, then automatically highlights the visual differences between the different executions. We evaluate our privacy controls on 50 applications from the Android marketplace, selected from those that were both popular and permission-hungry. We find that our privacy controls can successfully reduce the effective permissions of the application without causing side effects for 66% of the tested applications. The remaining 34% of applications implemented user-desired functionality that required violating the privacy requirements our controls were designed to enforce; there was an unavoidable choice between privacy and user-desired functionality.

1. INTRODUCTION

When a user prepares to install an application on the increasingly-popular Android platform, she will be presented with an ultimatum: either grant the application every permission demanded in its manifest or abandon the installation entirely. Even when a user agrees that an application should have access to sensitive data to provide functionality she desires, once the application has access to these data it may misappropriate them and exfiltrate them off the device. Despite the permission disclosures provided by the Android platform, the power of permission ultimatums and the inevitability that data will be misappropriated for exfiltration have created an application ecosystem in which privacy-invasive applications are commonplace [6]. U.S. federal prosecutors have even taken notice, initiating a criminal investigation into the misappropriation of users’ data by mobile applications [1].

We have developed a system, called AppFence, that retrofits the Android runtime environment to impose privacy controls on existing (unmodified) Android applications. AppFence lets users withhold data from imperious applications that demand information that is unnecessary to perform their advertised functionality and, for data that are required for user-desired functions, block communications by the application that would exfiltrate these data off the device.

When an application demands access to sensitive data a user doesn’t want it to have, AppFence substitutes innocuous *shadow data* in its place. For example, an application that demands a user’s contacts may receive shadow data that contains no contact entries, contains only those genuine entries not considered sensitive by the user, or that contains shadow entries that are entirely fictional. Similarly, an application that demands the unique device ID (IMEI number), which is frequently used to profile users across applications, may instead receive the hash of the device ID salted with a device secret and the application name. This shadow data provides the illusion of a consistent device ID within the application, but is different from the ID given to other applications on the same device. Presenting a different device ID to each application thwarts the use of this ID for cross-application profiling. In other words, when an application demands the device ID for the purpose of linking the user to a cross-application profile, shadowing the device ID empowers users to reply that their device is “not the droid you’re looking for.”

Shadowing prevents the ingress of sensitive data into applications, breaking applications that truly require the correct data to provide functionality the user wants. For example, a user cannot examine or search her contacts if the application only has access to an empty shadow contact list. For data that *is* allowed to enter the application, we introduce a complementary data-egress control to prevent information from being misappropriated and sent off the device: *exfiltration blocking*. We extend the TaintDroid information-flow tracking system to track data derived from information the user considers private, then block unwanted transmissions derived from these data. For each sensitive data type in the system, AppFence can be configured to block messages containing data of that particular type.

In this paper, we first measure how 110 popular permission-hungry applications use the private information that they have access to. We expose the prevalence of third-party analytics libraries packaged within applications and reveal that applications now send sensitive data over encrypted channels (Section 2). This investigation of existing applications' behavior guided us to design two privacy controls, shadowing and exfiltration blocking, that protect against undesired uses of the user's sensitive data by applications (Section 3). We then study the potential side effects of these two privacy controls on the user experience of 50 applications. We develop a novel testing methodology for efficiently and reliably repeating experiments to investigate the user-discernable side effects that result when privacy controls are imposed on applications. The testing process records applications' screenshots and highlights the differences between executions so that they can be easily analyzed visually (Section 4). The evaluation that we performed using this methodology shows that, by combining shadowing and exfiltration blocking, it is possible to eliminate all side effects in the applications we studied except for those that represent a direct conflict between user-desired functionality and the privacy policy our controls enforce—that private data must not leave the device (Section 5). We discuss future and related work in Sections 6 and 7, and we conclude in Section 8.

We make the following three contributions. First, we provide the deepest study to date (to our knowledge) of information exposure by Android applications in terms of types of information investigated, forms of exposure including encryption, and exposure patterns to advertising and analytics servers. Second, we present two privacy controls for reducing exposure and show experimentally that the controls are promising: the privacy controls reduced the effective permissions of 66% of the 50 applications in our testbed without side effects. Last, we develop a novel testing methodology to detect side effects by combining automated GUI testing and by visually highlighting differences between application screenshots. This methodology allows us to characterize the side effects of tested applications, revealing some common functionalities of Android applications that require the exposure of the user's sensitive data and are thus unavoidably in conflict with the goal of privacy controls.

2. PRIVACY RISKS ON ANDROID

To inform the design of our privacy controls, we performed several initial measurements and analyses of today's Android applications. As an application cannot misappropriate data it does not have access to, we first measured the prevalence with which applications request access to each type of potentially sensitive data. We then determined the prevalence with which applications exfiltrate data of each type and where they send the data to.

2.1 Sources of sensitive information

We examined 1100 popular free Android applications, sampling the 50 most popular applications from each of 22 categories listed by the Android marketplace as of November 2010. We identified 11 permissions that could result in the disclosure of 12 types of sensitive information: `location`, `phone_state` (granting access to phone number & unique device ID information types as well as call state), `contacts`, user `account` information, `camera`, `microphone`, browser `history & bookmarks`, `logs`, `SMS messages`, `calendar`, and `subscribed_feeds`. We measured the prevalence with which applications demanded each permission by parsing the applications' manifests using the publicly available Android `aapt` tool [11]. We find that 605 applications (55%) require access to at least one of these resources and access to the Internet, resulting in the potential for unwanted disclosure. We present these results broken down by resource type in Table 1.

Resource type	Applications
phone_state	374 (34.0%)
location	368 (33.5%)
contacts	105 (9.5%)
camera	84 (7.6%)
account	43 (3.9%)
logs	38 (3.5%)
microphone	32 (2.9%)
SMS messages	24 (2.2%)
history & bookmarks	19 (1.7%)
calendar	9 (0.8%)
subscribed_feeds	2 (0.2%)

Table 1: Of the 1100 popular Android applications we examined, those that required both access to a resource containing sensitive data and access to the Internet (through which data might be exfiltrated).

2.2 Misappropriation

Prior work has revealed that some Android applications do exploit user data for purposes that may not be expected or desired by users. Enck *et al.*, who developed the TaintDroid information-flow tracking system extended in our work, used this system to analyze 30 Android applications that required access to the Internet and either users' location, camera, or microphone [6]. They found that half of these applications shared users' locations with advertisement servers. The problem is not unique to Android. Egele *et al.* used static analysis to track information flow in popular iPhone applications and discovered that many contained code to send out the unique device ID [5]. Smith captured network traffic to observe iPhone applications transmitting device IDs [17]. The Wall Street Journal commissioned its own study of 50 iPhone applications and 50 Android applications, also using a network-observation approach [22, 21]. The article suspects that these unique IDs are so commonly transmitted because they can be used to profile users' behaviors across applications.

2.3 A profile of the profilers

Given the existing concerns over cross-application profiling of user behavior, we examined our sample of 1100 applications to identify third-party analytics & advertising (A&A) libraries that might build such profiles. We used the Android `apktool` [23] to disassemble and inspect application modules to identify the most commonly used libraries. We found eight A&A packages, listed in Table 2. AdMob was the most popular A&A package, employed by a third of our sample applications, followed by Google Ads. Google acquired AdMob in 2010; the combined application market share of AdMob and existing Google Ads and Analytics packages was 535 of our 1100 applications (49%). We also found that 591 applications (54%) have one or more A&A packages included in their code. Moreover, 361 of these applications (33%) demand access to the Internet and at least one of the resource types identified in Table 1, enabling the potential for disclosure of sensitive information to these third party servers.

A&A Module	Applications all 1100		sensitive 605	
admob.android.ads	360	(33%)	225	(37%)
google.ads	242	(22%)	140	(23%)
flurry.android	110	(10%)	88	(15%)
google.android.apps.analytics	91	(8%)	66	(11%)
adwhirl	79	(7%)	67	(11%)
mobclix.android.sdk	58	(5%)	46	(8%)
millennialmedia.android	48	(4%)	47	(8%)
qwapi.adclient.android	39	(3%)	37	(6%)

Table 2: The prevalence of third-party analytics and advertisement modules in our sample of 1100 Android applications, and a subset of 605 applications that demand access to at least one resource containing potentially sensitive information.

A&A destination	Any	IMEI	Loc
*.admob.com	57	0	11
*.doubleclick.net	36	0	0
data.flurry.com	27	2	15
*.googlesyndication.com	24	0	0
*.mydas.mobi	23	0	0
*.adwhirl.com	21	0	0
*.mobclix.com	17	10	6
*.google-analytics.com	17	0	0
tapad.jumptap.com	6	0	0
droidvertising.appspot.com	5	0	0
*.mojiva.com	4	0	0
ad.qwapi.com	2	0	0
*.greystripe.com	2	2	0
*.inmobi.com	1	0	1

Table 3: The number of applications (from our 110 application sample) that sent any communication to the A&A server, number that sent the unique device ID (IMEI), and number that sent the user's location.

2.4 Where sensitive information goes

Not all applications that request permission to access sensitive information will exfiltrate it. We ran an experiment to identify the prevalence with which applications transmit each type of sensitive information off the user's device and where they send it to. Performing this preliminary study required us to enhance TaintDroid, as it had previously only tracked five of the 12 data types examined in our study, and it did not track traffic sent through SSL. We also added instrumentation to record the identity of communicating parties and the traffic going to, and coming from, these parties.

Given our resource constraints, we limited our remaining analysis to a 110-application subsample of the 1100 applications we had started with. For each permission, we included in the subsample at least 10 applications that used this permission, drawing first from those applications that contained an A&A package and, if more applications were needed to reach our goal, next drawing from the set of applications without A&A packages but that still required Internet access.[1] Our sample is intentionally biased in favor of *permission-hungry* applications: those that require the most permissions. This bias toward permission-hungry applications only increases the likelihood that we will identify side effects when imposing privacy controls.

To perform this analysis, we manually executed each of the 110 applications for about five minutes, exercising the application's main features and any features we thought might require the use or exfiltration of sensitive data (the same methodology is used in [21, 22]).

We augmenting the list of A&A domain names previously obtained through static analysis by observing traffic from these 110 and manually inspecting the sites they contacted to verify which third-parties were A&A servers. The resulting list of domain names of A&A servers can be found in Table 3.

For each sensitive resource, Table 4 shows the number of applications in our 110-application subsample that demanded access to it, and the fraction that we observed transmitting messages tainted by data from this resource out to the Internet. The only data types we see transmitted are device ID (IMEI), phone number, location, contacts, camera, account, and microphone. Some applications may send more information than we observed as we could not guarantee that all code paths were executed. Table 3 shows the breakdown of A&A destinations that collected tainted data from applications. We observed that location was sent to Admob, Flurry, Mobclix, and Inmobi, and IMEI was sent to Flurry, Mobclix, and Greystripe.

Resource		Demanded	Sent to Anywhere		A&A	
phone_state	IMEI	83	31	37%	14	17%
	Phone#	83	5	6%	0	0%
location		73	45	62%	30	41%
contacts		29	7	24%	0	0%
camera		12	1	8%	0	0%
account		11	4	36%	0	0%
logs		10	0	0%	0	0%
microphone		10	1	10%	0	0%
SMS/MMS messages		10	0	0%	0	0%
history&bookmarks		10	0	0%	0	0%
calendar		8	0	0%	0	0%
subscribed_feeds		1	0	0%	0	0%

Table 4: The prevalence of permissions demanded by applications in the sample used for our initial information flow experiments. Note that the sum of the application counts is greater than 110 as many applications require access to multiple data types. For each data type, we tracked applications that demanded access to that data type and measured the fraction that transmitted messages tainted by that data type.

Phone number. Five applications transmitted phone numbers. Two applications required users to register a phone number, so they filled in the device's phone number by default when the user completed the registration form (but the user could then modify the phone number if desired). The third application used the phone number to create a custom unique device identifier, so the phone number was not disclosed directly in the payload. However, two applications–Dilbert comic viewer and Mabilo ringtones downloader–sent the device's phone number with no discernable legitimate purpose!

Contacts. Seven applications transmitted contacts. Two did so to perform contact-specific searches, and three sent contacts as requested by the user. One, a reverse phone number lookup application (Mr. Number), sent contact entries to its own servers; it asks the user to opt in, but only after it has already sent the data to

[1]Fewer than 10 applications requested access to the subscribed_feeds and calendar permissions.

its servers. An instant messaging application (KakaoTalk) sent the phone numbers collected from the user's entire address book to its servers to automatically add other users of the application. The transmission took place without any notice and this feature is turned on by default. Additionally, six of the seven applications sent the IMEI along with the contacts, making it easy for applications to link contacts with other information that is commonly collected as described below.

IMEI. 31 applications transmitted the IMEI(device ID). As reported by previous studies, the use of the device ID by applications is prevalent. 11 applications employed SSL secure connections when they transmitted the device ID to application servers. We find that these encrypted transmissions of the device ID sometimes accompany other sensitive data such as contacts and phone number. We find seven game applications that send the device ID over SSL along with a score to store high scores using a third-party company.

Location. 45 applications transmitted location data. Third-party servers are the most common destinations for location data; 30 applications shared location data with A&A servers. All but two of these 30 shared location data with A&A servers exclusively. Half (15) employ the Flurry analytics package, which uses a binary (non-human readable) data format when sending out location data to the Flurry server. Prior investigations that observed network traffic alone would not have detected the transmission of this information.

Camera & Microphone data. We observed that one application sent a photo and another application sent a voice memo. Both cases are triggered by explicit user requests.

Account. The account resource is used to store profile and authentication information for online accounts that the user has access to. Four applications transmitted data tainted by the account resource; all uses appear legitimate. One security application used account data to send email to the user's Gmail account. One multimedia application used account data to allow the user to register her Facebook account for creating personal profiles. One music sharing application used account data to authenticate the user with its server. One application used account data to access the Android market for providing enhanced services.

2.5 Informing privacy controls

Our preliminary analysis can guide the selection of privacy control mechanisms for protecting sensitive data. One simple approach would be to block *all* access to the Internet by the application. While this obviously would impede user-desired functionality in some cases, we wondered if it might be sufficient in others. Having intercepted and observed all Internet communications to and from these applications, we show the fraction of each application's Internet traffic that is used for advertising and anlytics (A&A) in Figure 1. Of the 97 applications in our 110 application sample that accessed A&A servers, 23 (24%) communicated exclusively with A&A servers during our observations. While these could presumably provide the same functionality if one simply denied all access to the network, the rest would likely exhibit side effects.

Given the variation in the types of sensitive data, ways of using this data for user-desired features, and ways to misuse this data, it may simply not be possible to apply a single, one-size-fits-all policy that can protect data while minimizing side effects. Thus, we set out to explore a choice of privacy controls that could be customized to balance the needs of applications with the privacy requirements of their users.

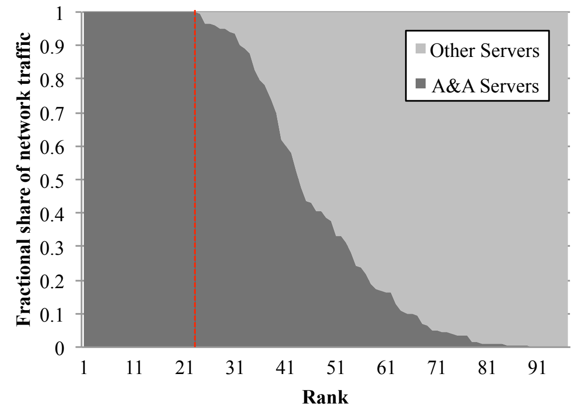


Figure 1: The fraction of network traffic (bytes inbound + bytes outbound) sent to A&A servers.

3. PRIVACY CONTROLS

AppFence implements data *shadowing*, to prevent applications from accessing sensitive information that is not required to provide user-desired functionality, and *exfiltration blocking*, to block outgoing communications tainted by sensitive data. Either (or even both) of these controls may be applied to limit an application's access to a sensitive data type.

3.1 Data shadowing

Since today's applications do not suspect the use of shadowing, we opt for simple shadow data rather than developing more elaborate ruses to fool applications that might attempt to detect shadowing. However, our implementation can be easily extended to support more sophisticated shadow data than what is presented below if it becomes necessary to do so.

Android applications use the file system to access the camera, microphone, and logs. When applications try to open these resources, we provide the illusion of opening an empty file. Similarly, we shadowed browser metadata (history and bookmarks), SMS/MMS messages, subscribed feeds, contacts, accounts, and calendar entries by returning an empty set of data.

When applications request the device's location, we return the coordinates 37.421265, -122.084026.

When applications request the device's phone state, we construct phone state with a fixed phone number (1 650 623 4000) and an application-specific device ID. The shadow device ID (IMEI) is generated by hashing a three-tuple containing the device ID, application name, and a secret salt randomly generated for the device. The salt ensures that an application that is granted access to the device ID cannot be linked to an application that is granted access to the shadow ID. The result of the hash is a string containing 15 decimal digits—the proper format for a GSM IMEI number.

The Android phone state permission also grants access the software version number (IMEI/SV), SIM serial number, voice mail number, and subscriber ID (IMSI). We did not observe any applications use these data, and thus did not test any shadowing strategies for them.

Implementation

The Android architecture isolates each application within its own runtime consisting of a Dalvik virtual machine and the Android core libraries. Each runtime is isolated within its own process. Because Android sandboxes applications from each other, the only way to impose privacy controls on unmodified applications is to modify the operating system itself, which includes the core libraries as well as the Android framework, a set of services and managers that reside outside of the application VMs. Figure 2 shows the components of the Android architecture that we modified for shadowing.

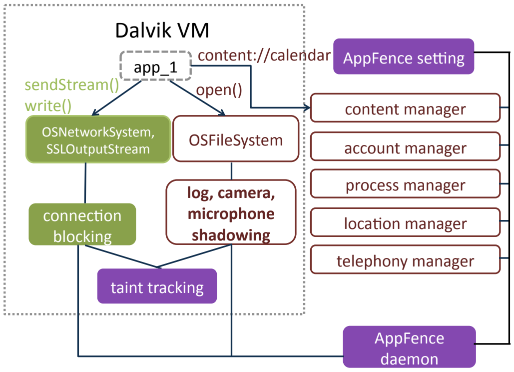


Figure 2: AppFence system architecture: the Dalvik VM contains the application and the Android core libraries, while resource managers reside in the Android framework. Existing resource manager and file system components are modified for shadowing, while exfiltration blocking introduces new components for connection blocking and taint tracking. The AppFence daemon runs as a native library, and is controlled by the AppFence settings application.

For simple resources such as the IMEI, phone number, and location, we return shadow values directly from the managers in the Android framework code. More complex resources, such as the user's calendar and contact list, are accessed through Android's `content provider` framework [9]. Applications identify the resource they wish to access via a URI. For example, the calendar may be queried with the string `content://calendar`. For these content provider resources, we replace the cursor that would normally be returned by the content manager with a shadow database cursor. For our experiments we return an empty database cursor, though one could instead create a shadow database and return a cursor to it.

3.2 Exfiltration blocking

To block exfiltration of data, we intercept calls to the network stack to (1) associate domain names with open sockets and (2) detect when tainted data is written to a socket. When an output buffer contains tainted data, we drop the buffer and choose one of two actions: we may drop the offending message *covertly*, misleading the application by indicating that the buffer has been sent, or *overtly*, emulating the OS behavior an application would encounter if the buffer were dropped as a result of the device entering airplane mode (all wireless connections disabled).

Implementation

To monitor and block network traffic, we modify both the Java code and native code in the Android networking stack. We rely on the Dalvik virtual machine to protect the integrity of our code running within the VM. Android's core native libraries are loaded on demand as applications require them. We prevent applications from directly loading their own native libraries as this could compromise the integrity of our taint tracking and privacy controls. At the time of testing, the use of native libraries was exceptionally rare; none of the 110 applications in our test set relied upon their own native libraries to run. Figure 2 shows key modules that we instrumented or created for exfiltration blocking.

When an application writes to a socket's output stream, the buffer is sent to the `sendStream()` method within the `OSNetworkSystem` core library. We modified `sendStream` so that if the buffer is tainted by data that should not be sent to its intended destination, we drop the buffer. When SSL sockets are used, we capture `write` calls to the `SSLOutputStream` class.

To emulate airplane mode, we first return error code `SOCKERR_TIMEOUT`, then block the next send with error code `SOCKERR_EPIPE`. If the application tries to open a new socket (via a `socket.connect()` call), we finally return a `SocketException` with error code `SOCKERR_ENETUNREACH`. Subsequent attempts to open sockets or send data will be allowed until we next encounter tainted data bound for a forbidden destination.

In order to facilitate user configuration and testing, we separate the policy specification mechanism into a service (daemon) that can be configured automatically or by users. Our privacy controls obtain their policies from this daemon. The privacy controls can be enabled globally or on a per-application basis.

AppFence relies on the open-source TaintDroid platform which, at the time of our testing, did not yet fully support just-in-time (JIT) compilation.[2] We have thus initially implemented AppFence for Android version 2.1, which does not use JIT compilation. Android 2.1 represented 24% of the Android installations accessing the Android Marketplace as of May, 2011 [12]. We did not encounter any compatibility issues running applications on Android 2.1.

Our combined implementation of shadowing and exfiltration blocking required introducing or modifying roughly $5,000$ lines of the Android platform code.

3.3 Limitations

One of the known limitations of our implementation is that the TaintDroid information flow tracking system, on which we built AppFence's exfiltration blocking feature, does not track information leaked through *control flow* operations. Applications intent on circumventing exfiltration blocking could move data using control flow operations. Tracking control flow may have reasonable overhead, especially if the code to do so is only activated when a tainted variable is loaded into the register space, but could raise the rate of false positives.

Still, actively circumventing AppFence would not be without consequences for software developers. Static analysis could be used to identify sections of code that appear to be designed to transfer data using control flow, exposing applications that actively attempt to subvert users' desired privacy policies. If application developers are found to be actively targeting and circumventing AppFence's exfiltration blocking controls, they may undermine their ability to employ the traditional excuse used to defend developers of privacy-invasive applications—that they operate openly with the implicit consent of a user base that is happy to reveal information.

An application that is aware of AppFence can detect the presence of exfiltration blocking. For example, an application could open two independent sockets, transmit tainted data over only one of those sockets and untainted data over the other socket, and have the server report back what it received. Similarly, shadow data may also not be convincing enough to fool an application. Applications that detect the presence of privacy controls could inform users that

[2]Collaborators at Duke University are adding JIT support to TaintDroid.

they refuse to provide user-desired functionality until these controls have been deactivated.

4. TEST METHODOLOGY

The primary cost of imposing privacy controls on applications is the introduction of side effects that negatively impact the user's experience. To enable the evaluation of our AppFence system, we developed a novel test methodology that allows us to automate the execution of applications and easily measure and characterize side effects introduced by the privacy controls. Our methodology overcomes the two main obstacles to systematic testing of the interaction between AppFence's privacy controls and applications: the ability to reproduce program executions (reproducibility), and the ability to detect side effects (detection). We describe how we use automated GUI testing and screenshot comparisons to tackle these issues in the next subsections.

We focus on *user-visible* side effects as the metric for evaluating AppFence because shadowing and exfiltration blocking have equivalent benefits when applied to the applications in our test bed; given that AppFence-unaware applications do not (at least to our knowledge) deliberately circumvent the information flow tracking used to block exfiltration, both privacy controls are equally effective on today's applications. We do not measure the performance impact of our privacy controls; the underlying information flow tracking provided by TaintDroid is fast enough to run applications in real-time with modest slowdown (worst case increase in CPU utilization of 14%), and beyond this we witnessed no discernable impact as applications with and without our privacy controls enabled ran side by side.

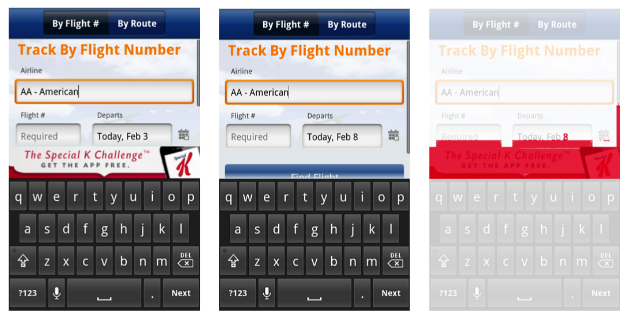
4.1 Automated application runs

Reproducibility is difficult because different runs of the same application may exercise different code paths. Furthermore, variations in user inputs, their timing, system state, and other factors may cause results to change. To minimize these variations, we built a test infrastructure that automates human usage patterns to remove variations in users' choices of actions and their timing. To this end we used the Android GUI testing system provided by the TEMA project [14, 19], which leverages the Android `monkey` event generator. The test system supports a scripting language in which user actions are expressed via high-level commands such as *TapObject*, *PressKey* and *SelectFromMenu*. Commands were sent from our PC running the GUI testing system to our Nexus One devices via a USB cable.

Given the labor-intensive nature of generating test scripts, we were able to script 50 of the 110 applications we examined in Section 2 (for a full list of these applications, see Appendix B). We scripted each application to perform its main tasks as we expected users to perform them. Our scripts are not guaranteed to cover all code paths, and so our results may not detect all uses of sensitive data by an application or all of the side effects of our privacy controls. The average time to execute each test script – excluding installation, uninstallation and cleanup – was 3.5 minutes, with an average of 24 script commands. We created a master test script that configures an Android device, enables the AppFence privacy controls for experimental configurations or disables them for the baseline configuration, and then tests all applications. For each application, the script installs and launches the application, executes the GUI test adapter to provide inputs, uninstalls the application, and then removes any changes to the device state caused by the application; we refer to these steps as an application *execution*.

4.2 Detecting changes in behavior

Detecting whether side effects impact *user*-desired functionality is a determination that eventually requires consultation of a user. However, placing a human in the loop can introduce bias and slow the process down, running counter to our goal of systematic, automated testing. To reduce the scalability constraints and bias caused by human evaluation, we leverage the insight that side effects are likely easy to detect and confirm if the visual outputs of the baseline and experimental executions can be compared side by side. We employed a feature of the GUI testing system to capture a screenshot from the Android device after every command in the test script. We first ran each test script with our baseline configuration—no resources were replaced with shadow resources and no attempts to exfiltrate data were blocked. We then ran each test script with our experimental configurations, in which either data shadowing or exfiltration blocking was activated. For each experimental execution, we automatically generated a web page with side-by-side screenshots from the baseline execution and the experimental execution, along with a visual `diff` of the two images. We found that these outputs could be scanned quickly and reliably, with little ambiguity as to whether a side effect had been captured in the image logs, as shown in Figure 3.



(a) Baseline execution (b) With exfiltration blocking (c) Visual diff between (a) and (b)

Figure 3: Detecting side effects using visual diff: The red shaded region in (c) highlights the advertising banner missing from (b).

We also monitored the tainted data exposure across test runs and found that it is not deterministic: it is possible for applications to transmit tainted data in some test runs but not others. We took steps to mitigate the underlying sources of variation during our testing. For example, we discovered that many applications request the most recent calculated location, without asking for the phone to access the GPS; they may do this to avoid the latency required to obtain updated location data, or to avoid the battery drain of activating the GPS unit. If a null location is returned, or if the last known location is stale (e.g. more than 60 minutes old), applications will often proceed without location data. To avoid inconsistencies during our testing, we modified the Android framework to always return a fixed default location, rather than null, when no last known location is available. To account for remaining variations in our testing, we examined the results of at least two test executions for every experimental configuration, and used additional executions and manual log inspection to resolve inconsistent application behavior.

5. EXPERIMENTS

This section shows the experimental results of testing AppFence's privacy controls on the 50 applications for which we generated test scripts (see Appendix B). We discuss the side effects resulting from the privacy controls and evaluate their impact on the user experience.

5.1 Experimental configurations

We executed applications over eight different experimental configurations. The control configuration, which did not have any privacy controls activated, represents how users run applications on Android today. In the *shadowing* configuration, sensitive data was replaced by shadow data, as described in Section 3.1. The remaining six configurations implemented some form of message blocking, three of which used overt blocking (simulating airplane mode) and three of which used covert blocking (pretending that blocked messages were actually sent). One pair of *exfiltration blocking* configurations (one covert, one overt) blocked messages tainted by sensitive data regardless of the server to which they were destined. Like data shadowing, these configurations are destination-agnostic. A pair of *destination-specific exfiltration blocking* configurations only blocked tainted messages if they were destined to known advertising & analytics (A&A) servers. Finally, to examine the benefits of exfiltration blocking over more naïve approaches, a *destination blacklisting* pair blocked all traffic to known A&A servers, regardless of whether it was tainted by sensitive data or not. (The list of known A&A servers can be found in Table 3.)

We divided the possible side effects impacting the user experience into four categories based on severity: the privacy controls had no side effect (*none*); advertisements no longer appeared (*ads absent*); the application still performed its primary purpose but failed to perform a less-important secondary function, or was otherwise *less functional*; or the application no longer fulfilled its primary purpose or crashed (*broken*). We then classified each application into one of these categories, based on the most severe side effect we observed in the entire execution of the application under our test script.

The definition of less functional (as opposed to broken) is somewhat subjective, and will vary according to the individual user. When classifying applications, we carefully considered the primary purposes for which a user would run a particular application, and when judgment calls were necessary, we made them in favor of more severe impacts. A detailed explanation of when we considered each application to be less functional is presented in Appendix A. Because we are concerned with evaluating the potential negative impact of our privacy controls on the *user's* experience, we do not consider the absence of advertisements to be a side effect, nor do we study the impact on application developers or their advertising and analytics partners.

5.2 Coarse-grained controls

Our first experiment examines the side effects of imposing privacy controls on all 12 data types simultaneously. We begin with such a coarse-grained analysis because it allows us to identify the best applications for further examination; those that are not impacted by coarse-grained privacy controls will not require more detailed analysis. Our results are summarized in Table 5. Advertising & analytics (A&A) servers don't generally provide user-desired functionality, so it is not surprising that the naïve approach of blocking tainted messages sent to known A&A servers has fewer side effects than approaches that block messages to other servers as well. However, even blocking just tainted messages to known A&A servers can cause disruption to the user experience if applications fail to handle blocking gracefully. For example, after a connection to an A&A server failed, one application assumed that the network was unavailable and abandoned all network access. Blocking all messages sent to A&A servers, rather than just those messages tainted by sensitive data, caused slightly more applications to break. Closer inspection revealed that these applications send untainted communications to A&A servers upon launch, which may cause them to wait indefinitely for a response (covert mode) or receive a socket exception that is interpreted as network unavailability (overt mode). For all exfiltration blocking configurations, we found negligible differences in the occurence of side effects caused by overt blocking versus covert blocking.

Alas, blocking only A&A servers only defends against behavioral advertising which, despite its popularity, is likely the least pernicious threat to sensitive data. More nefarious applications can circumvent such blacklist approaches, for example by proxying communications to A&A servers through their own (first party) servers. Preventing exfiltration of data through non-A&A servers requires one of our destination-agnostic approaches, i.e. using shadowing or using exfiltration blocking of tainted messages to all destinations. Table 5 shows that overall, shadowing causes fewer and less severe side effects than exfiltration blocking; a more detailed analysis is presented in the following section.

5.3 Fine-grained controls

We ran a second experiment to determine which resources were causing side effects when destination-agnostic privacy controls were applied. This required us to re-run our tests, applying privacy controls individually to each type of sensitive information. However, we only had to do so for those applications that were less functional or were broken when privacy controls had been applied to all types of information. For each resource (row) and privacy control (column) in Table 6, the corresponding entry shows the number of applications that experienced side effects as a result of imposing the privacy control on that resource.

Our results reflect that data types that are rarely directly presented to the user – device ID (IMEI), location, and phone number – are best protected by shadowing. Shadowing did not break any applications that attempted to send the device ID or phone number to their servers. Six applications did become slightly less functional when the device ID was shadowed—all were games that could still track their high scores, but not build cross-application high-score profiles. In contrast, eight applications that access the device ID broke when overt exfiltration blocking controls were imposed, and another seven were less functional. Many of these applications send data upon launch, waiting for confirmation before continuing, and thus break when exfiltration blocking is imposed. Others included the device ID in login information sent over an encrypted (SSL) socket, which we blocked. Because applications use the device ID in a way that is not directly visible to the user, shadowing the device ID is usually less disruptive to the user experience than actively blocking the communication.

When controlling access to the user's location, shadowing also had slightly fewer side effects than exfiltration blocking. Like the device ID, location coordinates are rarely presented to the user directly; rather, they are usually used to download information about a given location. Thus, exfiltration blocking will prevent any information from being retrieved, whereas shadowing will result in data being retrieved for the shadow location instead of the actual location. For some applications, data for the shadow location was not better than no data at all (as with exfiltration blocking), so these applications (14%) were classified as broken. However, the difference between the number of applications that were broken or less useful with location shadowing (28%) versus those broken or less useful with exfiltration blocking (39%) shows that some applications exfiltrated location data for purposes (such as analytics) that

	Shadowing	Exfiltration blocking of *tainted* messages to... *all* destinations Covert	Overt	*only* A&A servers Covert	Overt	Blocking *all* messages to A&A servers Covert	Overt
None	28 (56%)	16 (32%)	16 (32%)	45 (90%)	45 (90%)	19 (38%)	18 (36%)
Ads absent	0 (0%)	11 (22%)	11 (22%)	4 (8%)	4 (8%)	29 (58%)	26 (52%)
Less functional	14 (28%)	10 (20%)	10 (20%)	0 (0%)	0 (0%)	0 (0%)	1 (2%)
Broken	8 (16%)	13 (26%)	13 (26%)	1 (2%)	1 (2%)	2 (4%)	5 (10%)

Table 5: The side effects of imposing privacy controls on all 12 categories of sensitive data for 50 test applications.

	Breaks *or* less functional: Shadowing	Exfiltration blocking: Covert	Overt	Breaks (*only*): Shadowing	Exfiltration blocking: Covert	Overt
`device ID`	6/43 (14%)	16/43 (37%)	15/43 (35%)	0/43 (0%)	9/43 (21%)	8/43 (19%)
`location`	10/36 (28%)	14/36 (39%)	14/36 (39%)	5/36 (14%)	8/36 (22%)	8/36 (22%)
`contacts`	4/14 (29%)	2/14 (14%)	2/14 (14%)	2/14 (14%)	1/14 (7%)	1/14 (7%)
`history&bookmarks`	1/3 (33%)	0/3 (0%)	0/3 (0%)	0/3 (0%)	0/3 (0%)	0/3 (0%)
`phone number`	0/43 (0%)	3/43 (7%)	3/43 (7%)	0/43 (0%)	3/43 (7%)	3/43 (7%)
`SMS`	1/2 (50%)	0/2 (0%)	0/2 (0%)	1/2 (50%)	0/2 (0%)	0/2 (0%)
`calendar`	1/4 (25%)	0/4 (0%)	0/4 (0%)	0/4 (0%)	0/4 (0%)	0/4 (0%)

Table 6: For each type of sensitive information, the fraction of applications that require this information that *either* break or are less functional as a result of imposing a destination-agnostic privacy control (first three data columns), followed by the subset of *only* those applications that break – rather than just become less functional – as a result of these controls (the last three data columns). *Data types not represented by rows in this table did not cause our privacy controls to induce side effects.*

did not cause user-visible side effects when the location was shadowed. For these applications that use location data in a way that is not visible to the user, shadowing is a more appropriate privacy control than exfiltration blocking.

The results demonstrate that exfiltration blocking is best used for data that applications display to the user or allow the user to navigate. For example, whereas data shadowing causes four applications that use contacts to break or become less functional, only one of these applications is impacted by exfiltration blocking. Similar results are seen in Table 6 for bookmarks, SMS messages, and calendar entries.

Shadowing and exfiltration blocking are complementary, and when used together can produce fewer side effects than either can alone. While 28 of the 50 applications in our sample (56%) run side effect-free with just shadowing and merely 16 applications (32%) are side effect-free with exfiltration blocking, 33 (66%) could run side effect-free if the most appropriate privacy control (i.e. as determined by an oracle) could be applied to each application. Section 6.1 describes how we might determine appropriate privacy settings in the future.

The benefits of having two privacy controls to choose from are also apparent from Table 7, which presents another view of the data from our fine-grained analysis. This table characterizes the types of application functionality that were impacted by our privacy controls, and shows which data types led to side effects for shadowing, exfiltration blocking, or both. Many of the rows in this table show that for particular functionalities and data types, one control exhibits the side effect but the other does not, indicating that AppFence can avoid impacting this type of functionality if the appropriate privacy control is used.

Table 7 also offers further insight into the behavior of the tested applications. For example, returning to the previous discussion of applications that use location data in ways that are not visible to users, these applications are precisely those listed in the rows of the table for which exfiltration blocking of the location data type made applications broken or less functional while shadowing had no side effects.

Finally, Table 7 provides insight into the third of applications that were not side effect-free. For some types of functionality in the table (GameProfile, GeoSearch, GeoBroadcast, and FindOthers), applications experienced side effects when either privacy control was imposed. Revealingly, these functionalities are in direct conflict with the goal of the privacy control—to keep information from leaving the device. Cross-application game high-score profiles are in direct conflict with privacy controls designed for the very purpose of preventing cross-application profiles. Geographic database searches are implemented such that they cannot work unless the user reveals her location and allows the application to exfiltrate it. If the `mocospace` application is to identify which of your contacts are also using `mocospace`, it will need to send your contacts to its servers. In these cases, the user cannot have her functionality and privacy too. Fortunately, most applications in our evaluation ran without any side effects, even though the sample was heavily biased towards permission-intensive applications.

In summary, our in-depth analysis of side effects shows that AppFence's two privacy controls can block unwanted exposure of sensitive data by 66% of the applications that we tested without compromising any functionality. The remaining applications, however, do require transmission of sensitive data off the device for certain functionality, leaving a choice to the user between privacy and functionality. We have characterized the types of functionality that require the exposure of sensitive data, and the side effects that result if the data is shadowed or this exposure is blocked, to provide users with some guidance for making an informed decision.

6. FUTURE WORK

This section discusses promising avenues to explore in order to further strengthen AppFence. In particular, we discuss how to address the problems of determining which privacy controls to apply to which applications and data types, and preventing applications

Impacted functionality	Sh	EB	Data type	Applications impacted
Launch:	*Application can't launch because required network transaction contains sensitive data*			
	-	⊗	Phone #	`dilbert`, `yearbook`
	-	⊗	Device ID	`dex`, `docstogo`, `kayak`, `moron`, `yearbook`
	-	⊗	Location	`dex`, `docstogo`, `moron`
Login:	*User can't login because login request contains sensitive data*			
	-	⊗	Device ID	`assistant`, `tunewiki`
Query:	*User can't receive response to a query because query contains sensitive data*			
	-	⊗	Device ID	`wnypages`, `yellowpages`
	-	⊗	Location	`manga`
	-	⊗	Phone #	`callerid`
	-	⊗	Contacts	`callerid`
	-	⊖	Device ID	`iheartradio`
GameProfile:	*Can't access cross-application high-score profile associated with device ID*			
	⊖	⊖	Device ID	`droidjump`, `mario`, `papertoss`, `simon`, `smiley_pops`, `trism`
	-	⊖	Location	`papertoss`
GeoSearch:	*Can't perform geographic search*			
	⊗	⊗	Location	`compass`, `dex`, `starbucks`, `wnypages`, `yellowpages`
	⊖	⊖	Location	`apartments`, `iheartradio`, `npr`, `yearbook`
GeoBroadcast:	*Can't broadcast geographic location to others*			
	⊖	⊖	Location	`heytell`
FindOthers:	*Can't learn which contacts are also using this application*			
	⊖	⊖	Contacts	`mocospace`
SelectRecipient:	*Can't select contacts with whom to call, message, or share*			
	⊗	-	Contacts	`callerid`, `heytell`
	⊖	-	Contacts	`quickmark`
DeviceData:	*Can't access bookmarks, SMS messages, calendar reminders, or other device data*			
	⊖	-	Bookmarks	`skyfire`
	⊗	-	SMS	`sqd`
	⊖	-	Calendar	`tvguide`

'-': no side effect, '⊖': application less functional, '⊗': primary application functionality breaks.

Table 7: The types of application functionality that were impacted by AppFence's privacy controls. The symbols in the shadowing (Sh) and exfiltration blocking (EB) columns indicate the severity of the side effects observed when privacy controls were applied to the given data types. Applications may be listed multiple times if they exhibited side effects for multiple functionalities or for different data types.

from circumventing exfiltration blocking.

6.1 Determining privacy settings

While a user's privacy goals can be met by choosing the right privacy controls, the responsibility for making the correct choice must fall somewhere. To allow for more informed choices, we envision that AppFence instances could report application behaviors to a server and that users could report side effects. This data would reveal how applications use data and whether they will exhibit side effects if privacy controls are applied. Open problems to achieve this goal include finding ways to crowdsource the construction of application profiles while respecting users' privacy, detecting attempts by application developers to compromise the integrity of this system to the advantage of their applications, and finding the right set of choices to present to users based on the data available.

6.2 Hampering evasion

As we discussed in Section 3.3, applications may be able to exploit limitations of AppFence's information flow tracking, which only monitors data flow operations, to circumvent exfiltration blocking.

Tracking information flow through control dependencies may broaden the set of data that is marked as tainted and result in false positives, which would in turn result in the unwarranted blocking of messages from an application. One promising option is to continue information flow tracking that is less likely to overtaint, and simultaneously use a more aggressive tracking that may overtaint. When AppFence detects a message that is tainted only by the more aggressive flow tracking it would allow the message. However, it would also report the event and the conditions that led up to it, to our servers for further analysis. We would then perform more comprehensive offline analysis (e.g. influence analysis [15]) to detect the cause of the difference between more and less aggressive tainting.

Alas, we cannot prevent applications from exploiting side channels (e.g., cache latency) to cleanse data of taint and circumvent exfiltration blocking. As shadowing prevents applications from ever accessing private data, it may always be the safest way to protect data from truly malicious applications. Data shadowing can be extended to offer finer-granularity controls such as shadowing location with a nearby but less private place, e.g. the city center. However, this kind of context-dependent control would require more configuration, warranting more research to make such controls practical and useful.

7. RELATED WORK

The use of shadow resources dates back at least as far as 1979, when the `chroot` operation was introduced to run UNIX processes with a virtualized view of the file system hierarchy. Shadow password files allow system components that once accessed the real password files to get some of the information in that file without exposing the password hashes. Honeypots and Honeynets [20, 18, 16] have popularized the use of shadow resources to run malware while studying its behavior and limiting its potential to do damage. The prefix *honey* is frequently used for shadow resources created for the purpose of attracting an adversary and/or monitoring the adversary's behavior.

Felt and Evans propose a data shadowing scheme, called privacy-by-proxy [8]. Their mechanism is similar to our data shadowing as it provides a fake placeholder to third-party Facebook applications rather than the user's real information but the privacy-by-proxy is only effective to applications that access the user's information for the sole purpose of displaying the exact information back to the user. A recent paper by Beresford *et al.* also argues for replacing sensitive user data with "mock" (shadow) information. They apply data shadowing for a limited number of data types to 23 applications selected from those that were previously examined by Enck *et al.* using TaintDroid. However, they only tested to determine if shadowing could be applied to applications without causing them to crash–they did not measure user-discernable side effects [3].

There is also a wealth of prior work on the use of information-flow tracking to protect data confidentiality and integrity. Yin *et al.*'s Panorama uses dynamic information-flow tracking (DIFT) to perform offline analysis of data exfiltration by malware [27]. Chow *et al.*'s TaintBochs [4] uses DIFT to analyze the lifetime of security-critical data in memory, finding vulnerabilities when applications free memory containing encryption keys without first deleting them. Wang *et al.*'s PRECIP [25] tracks sensitive data (e.g., clipboard and user keystrokes) in Windows at the system-call level – tainting system objects – to prevent malicious processes from gaining access to them. However, it does not track taint propagation within applications and so the taint is lost when data is copied between objects. Perhaps most relevant is Vachharajani *et al.*'s RIFLE [24], which enforces security policies at runtime by translating programs into a custom instruction set architecture enhanced to track information flow.

Others have have worked to detect potential abuses of permissions and data by Android applications. Enck *et al.* [7] have developed a lightweight security checker, called Kirin, that analyzes manifest files to identify permissions that are dangerous when combined.

Android applications obtain user consent for all the permissions they will require at the time they are installed [10]. An alternative approach, to obtain consent for access to a resource at the time it is requested, is used for certain resources on Apple's iOS platform (e.g. location [2]). Requiring consent at time of access gives users more granular control over the time at which applications can access sensitive resources, and likely reduces the success rate of ultimatums. It does so at a cost of more frequent user interruptions. The Android team argues that the usability cost of time-of-access consents work "to the detriment of security" [10]. Regardless of when permissions are granted, neither the time-of-install nor the time-of-access consent model can prevent applications from misappropriating them.

We have argued that structuring time-of-installation consents as ultimatums, requiring users to either accept all permissions or abandon installation, gives developers strong and unfair power over users, especially when compared to our model which allows users to effectively opt out of granting access to sensitive data. To understand why these ultimatums are so powerful, it is useful to consider the classic *ultimatum game* [13, 26] from the field of economics. Like the ultimatum game, installation consent ultimatums leave no opportunity for negotiation. Moreover, only the most optimistic user would believe that her individual behavior would change the permissions offered by a developer in the future.

8. CONCLUSION

AppFence offers two different approaches for protecting sensitive data from today's Android applications: shadowing sensitive data and blocking sensitive data from being exfiltrated off the device. We find that these privacy controls are complementary: when both privacy controls are available and the appropriate control can be chosen for each application and data type, all of the potentially-avoidable side effects in the applications we studied could be avoided. The only side effects that remain are those that represent a direct conflict between user-desired functionality and the privacy requirement that these controls are designed to enforce: that sensitive data not be allowed to leave the user's device. The testing methodology that we have developed for assessing side effects proves valuable for characterizing the types of application functionality that may be impacted by privacy controls.

Acknowledgments

We would like to thank Intel Labs for supporting this work, William Enck for sharing Android application binaries and Byung-Gon Chun, Peter Gilbert, Daniel Halperin, Patrick Gage Kelley, Robert Reeder, and Anmol Sheth for providing valuable feedback. This work was supported by National Science Foundation award CNS-0917341.

9. REFERENCES

[1] S. T. Amir Efrati and D. Searcey. Mobile-app makers face U.S. privacy investigation. `http://online.wsj.com/article/SB10001424052748703806304576242923804770968.html`, Apr. 5, 2011.

[2] Apple Inc. iPhone and iPod touch: Understanding location services. `http://support.apple.com/kb/HT1975`, Oct. 22, 2010.

[3] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.

[4] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, 2004.

[5] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *NDSS*, 2011.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[7] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *CCS*, 2009.

[8] A. Felt and D. Evans. Privacy protection for social networking apis. In *Proceedings of Web 2.0 Security And Privacy (W2SP)*, 2008.

[9] Google Inc. Android developers: Content providers. `http://developer.android.com/guide/topics/providers/content-providers.html`.

[10] Google Inc. Android developers: Security and permissions. http://developer.android.com/guide/topics/security/security.html.

[11] Google Inc. Android developers: Using aapt. http://developer.android.com/guide/developing/tools/aapt.html.

[12] Google Inc. Android developers: Platform versions. http://developer.android.com/resources/dashboard/platform-versions.html, May 2011.

[13] W. Güth, R. Schmittberger, and B. Schwarz. An experimental analysis of ultimatum bargaining. *Journal of Economic Behavior & Organization*, 3, Dec. 1982.

[14] A. Jääskeläinen. *Design, Implementation and Use of a Test Model Library for GUI Testing of Smartphone Applications*. Doctoral dissertation, Tampere University of Technology, Tampere, Finland, Jan. 2011.

[15] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, June 15, 2009.

[16] N. Provos. A virtual honeypot framework. In *USENIX Security Symposium*, 2004.

[17] E. Smith. iPhone applications & privacy issues: An analysis of application transmission of iPhone unique device identifiers (UDIDs). In *Technical Report*, 2010.

[18] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, Boston, MA, Sept. 10, 2002.

[19] Tampere University of Technology. Introduction: Model-based testing and glossary. http://tema.cs.tut.fi/intro.html.

[20] The Honeynet Project. *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison-Wesley, 2001.

[21] S. Thurm and Y. I. Kane. The Journal's cellphone testing methodology. The Wall Street Journal. Dec. 18, 2010. http://online.wsj.com/article/SB10001424052748704034804576025951767626460.html.

[22] S. Thurm and Y. I. Kane. Your apps are watching you. The Wall Street Journal. Dec. 18, 2010. online.wsj.com/article/SB10001424052748704694004576020083703574602.html.

[23] *Unknown*. android-apktool: Tool for reengineering android apk files. http://code.google.com/p/android-apktool/.

[24] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *MICRO*, 2004.

[25] X. Wang, Z. Li, N. Li, and J. Y. Choi. PRECIP: Practical and Retrofittable Confidential Information Protection. In *NDSS*, Feb. 2008.

[26] Wikipedia. Ultimatum game. http://en.wikipedia.org/wiki/Ultimatum_game.

[27] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS*, 2007.

APPENDIX

A. WHEN APPLICATIONS ARE "LESS FUNCTIONAL"

When evaluating the impact of privacy controls on user experience, we consider certain side effects to render an application "less functional" when the application is able to perform its primary purpose but cannot perform some secondary function. In this appendix we explain the precise circumstances that led us to classify applications as less functional.

device ID (IMEI): We classified as less functional games that could not load a cross-application high-score profile because the profile is associated with the true device ID. Additionally, we classified the `iheartradio` application as less functional because its searches for nearby radio stations failed due to the inclusion of the device ID with the search request.

location: We included those applications where location proximity would have provided enhanced, but not core, functionality. For example, the `npr` radio application enhances its primary service by identifying the user's local stations, `yearbook` offers local chat in addition to its other chat options, and `heytell` allows users to optionally include their current location along with sent messages. We also included some applications that could no longer automatically capture the user's location, but offered users the option of manually entering their location (e.g. the `apartments` apartment-hunting application). Finally, the `papertoss` application became less functional when its high-score profile failed to load because it sends the user's location along with the request.

contacts: We included one chat application, `mocospace`, that could no longer add users' local contacts to the server-side chat contacts database. We also classified as less functional a barcode scanning application, `quickmark`, that offers the ability to send a bar code image to someone in the contacts book, but was not be able to do so if contacts were protected by our privacy controls.

bookmarks: We included a browser, `skyfire`, that could still browse the web but was not be able to read or save bookmarks if they were protected.

calendar: We classified as less functional the `tvguide` application that cannot add reminders to the user's calendar if the calendar has been replaced by a shadow calendar.

B. APPLICATIONS SCRIPTED FOR AUTOMATED TESTING

#	application	package name
1	antivirus	com.antivirus
2	apartments	com.cellit.forrent
3	assistant	com.netgate
4	astrid	com.timsu.astrid
5	autorun	com.rs.autorun
6	avril	com.ringtone.avrillavigne
7	basketball	com.droidhen.basketball
8	bible	com.faithcomesbyhearing.android.bibleis
9	callerid	net.bsdtelecom.calleridfaker
10	christmas	com.maxdroid.christmas
11	chuck_norris	com.bakes.chucknorrisfacts
12	compass	com.a0soft.gphone.aCompass
13	dex	com.mportal.dexknows.ui
14	dilbert	com.tarsin.android.dilbert
15	docstogo	com.dataviz.docstogo
16	droidjump	com.electricsheep.edj
17	espn	com.espnsport
18	flightview	com.flightview.flightview_free
19	fmlife	fmlife.activities
20	heytell	com.heytell
21	howtotie	com.artelplus.howtotie
22	iheartradio	com.clearchannel.iheartradio.controller2
23	kayak	com.kayak.android
24	manga	com.ceen.mangaviewer
25	mario	de.joergjahnke.mario.android.free
26	minesweeper	artfulbits.aiMinesweeper
27	mocospace	com.jnj.mocospace.android
28	moron	com.distinctdev.tmtlite
29	mp3_ringtone	net.lucky.star.mrtm
30	musicbox	com.dreamstep.musicbox
31	npr	org.npr.android.news
32	papertoss	com.bfs.papertoss
33	princesses	com.socialin.android.puzzle.princess
34	quickmark	tw.com.quickmark
35	simon	com.neilneil.android.games.simonclassic
36	simpsons	us.sourcio.android.puzzle.simpson
37	skyfire	com.skyfire.browser
38	slotmachine	com.slot.slotmachine
39	smarttactoe	com.dynamix.mobile.SmartTacToe
40	smiley_pops	com.boolbalabs.smileypops
41	sqd	com.superdroid.sqd
42	starbucks	com.brennasoft.findastarbucks
43	taskos	com.taskos
44	trism	com.feasy.tris2.colorblocks
45	tunewiki	com.tunewiki.lyricplayer.android
46	tvguide	com.roundbox.android.tvguide.presentation.activity
47	videopoker	com.infimosoft.videopoker
48	wnypages	com.avantar.wny
49	yearbook	com.myyearbook.m
50	yellowpages	com.avantar.yp