Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications

Jinseong Jeon* jsjeon@cs.umd.edu
Kristopher K. Micinski* micinski@cs.umd.edu
Jeffrey A. Vaughan† jeff.vaughan@logicblox.com
Ari Fogel‡ arifogel@cs.ucla.edu
Nikhilesh Reddy‡ nreddy@nreddy.com
Jeffrey S. Foster* jfoster@cs.umd.edu
Todd Millstein‡ todd@cs.ucla.edu

ABSTRACT

Google's Android platform includes a permission model that protects access to sensitive capabilities, such as Internet access, GPS use, and telephony. While permissions provide an important level of security, for many applications they allow broader access than actually required. In this paper, we introduce a novel framework that addresses this issue by adding finer-grained permissions to Android. Underlying our framework is a taxonomy of four major groups of Android permissions, each of which admits some common strategies for deriving sub-permissions. We used these strategies to investigate fine-grained versions of five of the most common Android permissions, including access to the Internet, user contacts, and system settings. We then developed a suite of tools that allow these fine-grained permissions to be inferred on existing apps; to be enforced by developers on their own apps; and to be retrofitted by users on existing apps. We evaluated our tools on a set of top apps from Google Play, and found that fine-grained permissions are applicable to a wide variety of apps and that they can be retrofitted to increase security of existing apps without affecting functionality.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Design, Languages, Security

*University of Maryland, College Park
†LogicBlox; work performed while the author was at UCLA
‡University of California, Los Angeles

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPSM'12, October 19, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1666-8/12/10 ...$15.00.

Keywords

Android, binary transformation, fine-grained permissions

1. INTRODUCTION

Google's Android is the most popular smartphone platform, running on 52.5% of all smartphones [16], and with more than 10 billion apps downloaded from the Market [18]. To help address security concerns, Android uses *permissions* to protect access to sensitive resources, including the Internet, GPS, and telephony.

While Android permissions provide an important level of security, the available permissions are often much more powerful than necessary. For example, the Amazon shopping app must acquire full Internet permission, enabling the app to send and receive data from *any* site on the Internet, not just `amazon.com`. Other Android permissions are similarly broad, granting arbitrary access to a particular resource (e.g., the user's contacts) or granting several unrelated privileges with a single permission (e.g., the ability to modify system settings). The result is that many apps violate the principle of least privilege [29], and the excessive privileges held by apps may be used to violate user privacy and security, e.g., by directly accessing sensitive resources or exploiting privilege escalation vulnerabilities. This problem is not merely theoretical; for example, a recently publicized security hole on HTC Android phones made detailed system logs available from a local socket. Access to local sockets is granted by the Internet permission, so any app with Internet permission could access the logs [2].

We believe that coarse-grained permissions impose costs on Android users, app developers, and platform owners (e.g., Google). Individual users are harmed when a malicious app uses an overly broad permission to perform an unexpected and undesired activity. Developers are harmed when permissions are insufficient to certify than an app will only interact appropriately with phone resources. Finally, the platform owner is harmed when users lose confidence or developers lose sales.

In this paper, we propose to address this issue in a natural way: by splitting existing Android permissions into a larger number of fine-grained permissions, as others have suggested [3, 15]. Our goals are to help app developers identify their own permission errors and improve their apps'

robustness against security exploits, and to enable users to gain stronger assurances about how apps use permissions. Thus, rather than focus on particular Android security exploits, our work addresses the coarse-grained models that potentially allow vulnerabilities to hide in plain sight.

While conceptually simple, supporting fine-grained permissions in practice presents several technical challenges, which we address by developing several tools.

RefineDroid. First, the new permissions should naturally and succinctly capture the most common uses of Android permissions; otherwise the finer granularity risks overwhelming both developers and users. We have developed a taxonomy that classifies Android permissions into four broad categories based on the kind of resource being protected (Section 2). We observe that the permissions in each category share a few general strategies for producing practical fine-grained variants. For example, one category contains permissions that control access to structured personal information, such as contacts and calendar information, and finer-grained permissions can leverage this structure to restrict accessibility (e.g., to a contact's email address but not phone numbers, or only to particular calendars).

To validate the practical utility of our taxonomy we have built RefineDroid, a static analysis tool that infers fine-grained permission usage in existing apps (Section 3). Currently RefineDroid infers fine-grained variants of five common Android permissions, including access to the Internet, user contacts, and system settings. For example, the tool infers permissions of the form *InternetURL(d)*, which grants access only to network domain *d*. We ran RefineDroid on 750 of the most popular apps from the Google Play app store. We found that in most cases a small number of fine-grained permissions from our taxonomy can replace the original Android permissions to provide the desired access, significantly reducing over-privilege and providing a clear security policy for developers and users.

Mr. Hide. Second, fine-grained permissions should be enforceable without modifying the Android platform or jailbreaking of devices; otherwise the benefits of the approach would be out of reach for most Android users. We have developed Mr. Hide (the Hide interface to the droid environment), a set of Android services that wraps several privileged Android APIs and dynamically enforces a specified set of fine-grained permissions from our taxonomy (Section 4). Mr. Hide runs in its own processes, and all fine-grained permission checking is done within these processes. Thus, Android's process separation ensures that client apps cannot subvert Mr. Hide's narrow API. We also provide a client-side library called hidelib that handles all communication with Mr. Hide and is a drop-in replacement for sensitive Android APIs, making it as easy for developers to use fine-grained permissions as the original Android ones.

Dr. Android. Finally, apps should be able to be retrofitted with fine-grained permissions without requiring source code access or recompilation; otherwise the approach cannot help secure the large number of existing apps. The ability to retrofit apps also allows power users and enterprises to customize the apps they use to conform to desired security policies. We introduce Dr. Android (Dalvik Rewriter for Android), a tool that removes Android permissions in existing application package files (apks) and replaces them with a specified set of fine-grained versions that are accessed through Mr. Hide (Section 5). Dr. Android includes a small number of high-level transformations that together enable expressive and well-formed security rewritings. Experiments on 14 apps validate the correctness and practicality of the approach (Section 6).

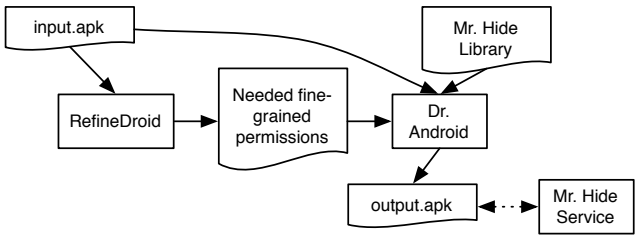


Figure 1: Overview of fine-grained permissions framework

In summary, we have produced a rich toolchain—RefineDroid, Mr. Hide, and Dr. Android—that allows app developers and their users to understand, infer, and enforce fine-grained permissions on stock Android devices and apps. Figure 1 overviews this toolchain and the relationships among its components. Different clients can use our tools in different ways that fit their needs. For example, a developer writing a new app can simply employ Mr. Hide and hidelib directly in lieu of the standard Android permissions and APIs to reduce over-privilege and give more confidence to users. A developer who wants to upgrade his existing app with Mr. Hide permissions can do the same but additionally employ RefineDroid to first identify the required fine-grained permissions. Finally, a sophisticated user can follow the full process shown in the figure, and employ Dr. Android to retrofit a downloaded app with Mr. Hide permissions in order to enforce a desired security policy, employing RefineDroid to understand the requirements of the application.

2. FINE-GRAINED PERMISSIONS

The Android platform has a wide range of permissions, which provide access to many different kinds of data and functionality. Thus, there is no one-size-fits-all approach to developing finer-grained permissions. On the other hand, we would like to avoid needing a different strategy for creating fine-grained variants of every single permission.

We have developed a taxonomy that partitions Android permissions into broad categories based on the kind of resource being protected. We observe that a few, general strategies for developing practical fine-grained variants can be shared by all permissions in the same category. Figure 2 gives a complete mapping from all dangerous and normal permissions on Android platform level 10 (versions 2.3.4–2.3.7) into our taxonomy. Within each category, we picked one or two exemplars and developed fine-grained implementations (named in italics in the figure) in Mr. Hide, support for them in Dr. Android, and, where applicable, permission inference for them in RefineDroid.

We next discuss each category in our taxonomy in more detail, along with associated strategies for refinement.

Category 1: Access to Outside Resources. Our first category contains 11 Android permissions that enable access to external resources, including Internet access, sending and receiving text messages, and reading and writing external storage. These permissions are naturally parameterized by

1. Outside Resources	*InternetURL(d)*	2. Structured User Info.	*ContactCol(c)*
INTERNET, RECEIVE_(MMS | SMS | WAP_PUSH), SEND_SMS, WRITE_EXTERNAL_STORAGE, BLUETOOTH, NFC, CALL_PHONE, USE_SIP, PROCESS_OUTGOING_CALLS		(READ | WRITE)_(CALENDAR | CONTACTS | SMS), READ_LOGS, (GET | MANAGE)_ACCOUNTS, AUTHENTICATE_ACCOUNTS, USE_CREDENTIALS, SET_ALARM, SUBSCRIBED_FEEDS_(READ | WRITE)	
3. Sensors	*LocationBlock*	4. System State/Settings	*ReadPhoneState(p)* *WriteSettings(s)*
ACCESS_(FINE | COARSE)_LOCATION, CAMERA, RECORD_AUDIO, ACCESS_LOCATION_EXTRA_COMMANDS		(READ | WRITE)_HISTORY_BOOKMARKS, WRITE_SETTINGS, (READ | MODIFY)_PHONE_STATE, GET_TASKS, WAKE_LOCK, (ACCESS | CHANGE)_(NETWORK | WIFI)_STATE, BATTERY_STATS, BLUETOOTH_ADMIN, MODIFY_AUDIO_SETTINGS, CHANGE_CONFIGURATION	
Other permissions	(READ | WRITE)_SYNC_SETTINGS, READ_SYNC_STATS, RECEIVE_BOOT_COMPLETED, GET_PACKAGE_SIZE, ACCESS_MOCK_LOCATION, SET_TIME_ZONE, REORDER_TASKS, KILL_BACKGROUD_PROCESSES, SET_WALLPAPER(_HINTS), WRITE_APN_SETTINGS, DISABLE_KEYGUARD, EXPAND_STATUS_BAR,BROADCAST_STICKY, CLEAR_APP_CACHE, SET_DEBUG_APP, SET_ALWAYS_FINISH, SET_PROCESS_LIMIT, SET_ANIMATION_SCALE, SIGNAL_PERSISTENT_PROCESSES, MOUNT_(UNMOUNT | FORMAT)_FILESYSTEMS, VIBRATE, FLASHLIGHT		

Figure 2: Taxonomy of all dangerous and normal permissions on Android platform level 10

the particular external resource being accessed. For example, Internet access involves specifying an Internet domain, text messages are sent to a phone number in a certain area code, and SD card access involves specifying a directory or file name. Thus, there are two natural strategies for fine-grained variants of this category: using a whitelist of allowed resources (domains, area codes, directories, etc.), or a blacklist of forbidden resources.

In our framework, we have chosen to focus on Internet access, which is pervasive across applications and may be particularly dangerous. We developed a fine-grained, whitelisting permission *InternetURL(d)*, which allows network connections only to domain d and its subdomains. Notice that this permission does not remove the need to trust the target domain d to some extent. However, it does help ensure that app vulnerabilities cannot be exploited to contact malicious web sites, and it limits Internet access for suspicious apps.

Category 2: Access to Structured User Information. Our second category contains 14 Android permissions that access structured user data, such as a user's calendar, contact list, and account information. For these permissions, we can introduce fine-grained variants that leverage the structure to grant access to a subset of the information. For example, calendar information could be restricted to entries on a particular calendar (e.g., my public calendar), contact information could be restricted to a particular set of fields (e.g., name and phone number), and apps could be granted access to a subset of available accounts.

We developed a fine-grained variant of Android's READ_CONTACTS permission. Our new permission, called *ContactCol(c)*, allows access only to a particular field (or *column*) c of an accessed contact in the user's address book.

Category 3: Access to Sensors. Our third category contains 7 Android permissions that protect access to various sensors on the phone, including the camera, GPS receiver, and microphone. We can derive fine-grained variants of these by reducing the fidelity of the signal received from these resources. For example, we could truncate low-order bits of the GPS location, suppress background noise on the audio channel, or redact a portion of a camera image.

We developed a fine-grained permission *LocationBlock* that blurs location information by truncating low-order bits to provide approximately 150m resolution (about one city block). This allows a user to gain utility from many apps without disclosing her exact location.

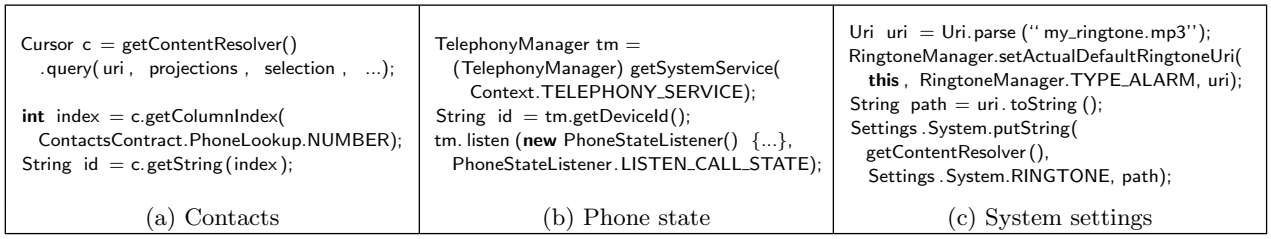
Category 4: Access to System State and Settings. Our fourth category contains 15 Android permissions that provide access to state and settings information on the phone. Typically each such permission provides access to read or update several unrelated pieces of information. For example, Android's READ_PHONE_STATE, among other things, allows an application to determine if a phone call is occurring and read the phone's unique IMEI number. Similarly, the WRITE_SETTINGS permission allows an app to update many distinct phone settings, including the default ringtone and network preferences. Thus, for these permissions, we can introduce fine-grained variants that separately grant access to each state or setting of interest.

In our framework, we propose the permissions *ReadPhoneState(p)* and *WriteSettings(s)*, which specialize the Android permissions of the same name as described above. In Mr. Hide and Dr. Android, we implemented instantiations of these permissions for the two most common uses in our app test set: *ReadPhoneState(*UniqueID*)*, which grants access to the device IMEI number, and *WriteSettings(*Ringtone*)*, which allows apps to update the phone's default ringtone.

Other permissions. Finally, there are several Android permissions that do not fit into the four categories above, either because they already are sufficiently fine-grained or because they would not benefit from finer granularity. For example, *RECEIVE_BOOT_COMPLETED* has only one purpose that does not seem useful to subdivide, and while *KILL_BACKGROUND_PROCESSES* could potentially be fragmented (e.g., by restricting the processes that could be killed), doing so seems unlikely to add much practical security.

3. FINE-GRAINED PERMS. IN THE WILD

In the previous section we introduced several new fine-grained permissions. In this section, we investigate how often such permissions can be applied to a suite of the most



```
Cursor c = getContentResolver()
  .query(uri, projections, selection, ...);

int index = c.getColumnIndex(
  ContactsContract.PhoneLookup.NUMBER);
String id = c.getString(index);
```

(a) Contacts

```
TelephonyManager tm =
  (TelephonyManager) getSystemService(
    Context.TELEPHONY_SERVICE);
String id = tm.getDeviceId();
tm.listen(new PhoneStateListener() {...},
  PhoneStateListener.LISTEN_CALL_STATE);
```

(b) Phone state

```
Uri uri = Uri.parse("my_ringtone.mp3");
RingtoneManager.setActualDefaultRingtoneUri(
  this, RingtoneManager.TYPE_ALARM, uri);
String path = uri.toString();
Settings.System.putString(
  getContentResolver(),
  Settings.System.RINGTONE, path);
```

(c) System settings

Figure 3: Typical idioms detected by RefineDroid

popular apps from Google Play, and how many fine-grained permission are needed to replace each standard permission in these apps.

3.1 Inferring Fine-grained Permissions

To answer these questions, we developed RefineDroid, a static analysis tool that analyzes an app's Dalvik bytecode to infer which of our fine-grained permissions are used by that app. In particular, RefineDroid infers which network domains are accessed, i.e., *InternetURL(d)* permissions (category 1); which contact columns are used, i.e., *ContactCol(c)* permissions (category 2); which phone states are read, e.g., for *ReadPhoneState(p)* permissions (category 4); and which system settings are written, e.g., for *WriteSettings(s)* permissions (also category 4). We believe RefineDroid is extensible to other category 1, 2, and 4 permissions.

RefineDroid does not infer usage of *LocationBlock*, since it and other category 3 permissions require an understanding of the sensor fidelity necessary for proper functioning of an app. For example, it would not be useful to employ *LocationBlock* in a navigation app, whereas it may be useful for an app that locates the nearest 5 post offices. In Section 6, we find that *LocationBlock* provides acceptable results for several popular apps.

RefineDroid's analysis contains three major components. First, it includes a simple string analysis that determines what strings matching a particular pattern occur in the string pool in the bytecode, e.g., what URL-like strings are mentioned. Second, RefineDroid includes a mapping from Android API calls to their necessary privileges, taken from the publicly available data of the Stowaway project [15]. Third, RefineDroid implements constant propagation for integers and strings [1] so that we can determine what strings may be passed to key parameters of privileged methods, e.g., what name might be passed to Settings.System.putString(). Our constant propagation analysis is a standard dataflow analysis, augmented to precisely track the effects of StringBuilder.append, String.concat and Uri.<init>, as many apps use these methods to manipulate the strings we must track.

Technical Details. RefineDroid infers the four permission groups mentioned above as follows. Note that below we describe code patterns for each permission at the source level, but RefineDroid actually operates on Dalvik bytecode.

InternetURL(d). RefineDroid collects all URL-like strings appearing in the apk's string pool and assumes those correspond to domains used by the app. Here by URL-like, we simply mean that a standard URL parser accepts it as valid.

ContactCol(c). Figure 3(a) gives some sample code that queries the contacts database. First the code derives a *cursor* from one of several URIs for the contacts database. Then specific columns are retrieved from the cursor by calling its getString() method with an index number, which is typically determined by a call to getColumnIndex(), as shown in the figure. Thus, to find which contact columns are accessed, RefineDroid uses constant propagation to find indices passed to getColumnIndex().

ReadPhoneState(p). Figure 3(b) shows sample code that accesses phone state information. The code begins by getting a TelephonyManager instance tm. That instance is used in two different ways to access phone state. First, the call tm.getDeviceId() retrieves the phone's IMEI number. Thus, in this and similar cases, RefineDroid determines a permission is needed because a particular method is invoked. Second, the call to tm.listen() registers a callback that will be invoked when there are state changes specified by the second argument to listen(). Thus, for this case, RefineDroid uses constant propagation to find the values of the second arguments to listen() and thereby determine needed permissions.

WriteSettings(s). Figure 3(c) gives sample code that writes to system settings, again in two different ways. First, the code uses a RingtoneManager method to change a ringtone. For these cases, RefineDroid relies on method names to determine permissions. Next, the code uses Settings.System.putString() to change the ringtone. For this case, RefineDroid once again uses constant propagation to find the arguments to putString() to determine permissions.

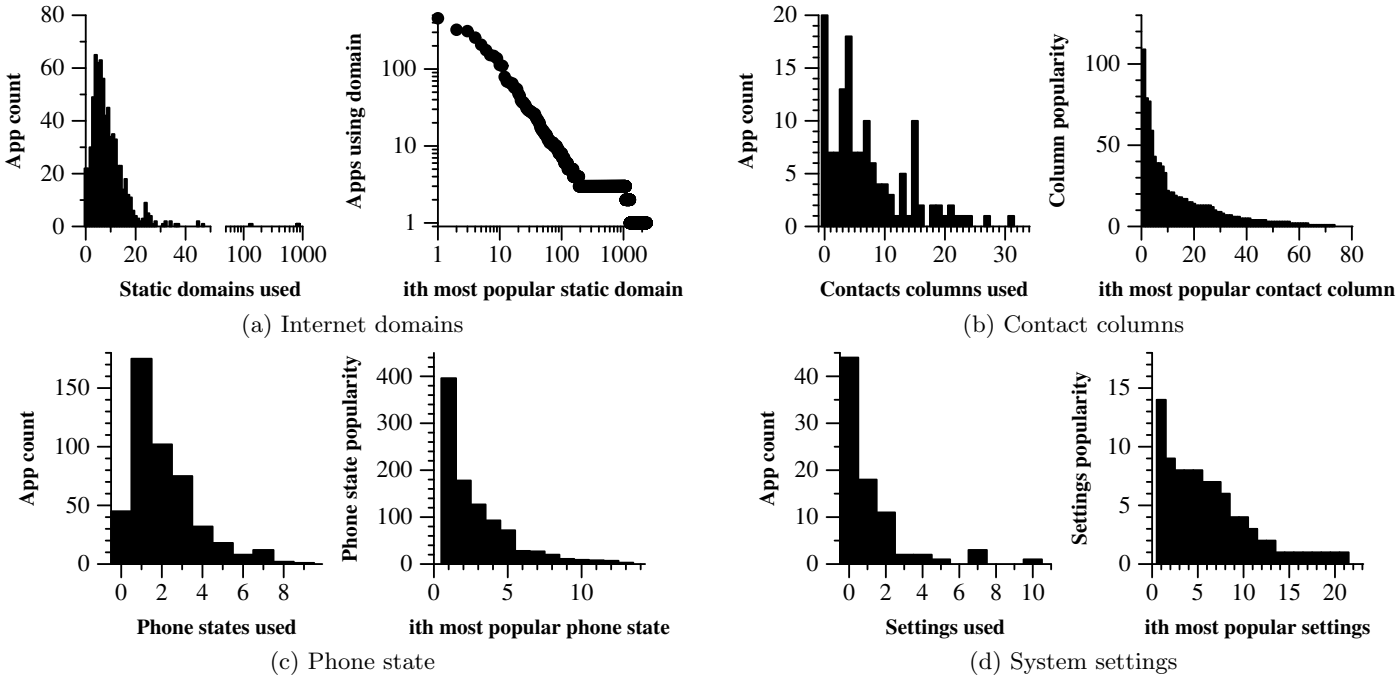
Limitations. RefineDroid deliberately performs a best-effort static analysis, rather than attempting to be conservative. For example, if RefineDroid cannot determine which contact column index is passed to a particular getString(), then it ignores that call, rather than conservatively assuming that all indices are possible. As another example, RefineDroid does not track invocations of the methods in some deprecated APIs, and it does not track invocations that occur via reflection or in native code. Thus, RefineDroid may have false negatives.

As is typical with static analysis, RefineDroid may also have false positives, e.g., RefineDroid will report a statically detected call to a privileged method even if it is dynamically unreachable, and it will report a URL-like string even if it is never used to access the network.

In Section 6, we report on the use of RefineDroid to infer Internet domains and contact columns on a test suite of apps from Google Play. On these apps we find that RefineDroid produces useful results with a low rate of false negatives. (Full details in Section 6.)

3.2 Study Results

We ran RefineDroid on the top 24 free apps in each Google Play category as of April 11, 2012, yielding 750 apps in to-



(a) Internet domains (b) Contact columns

(c) Phone state (d) System settings

Figure 4: Results of running RefineDroid analysis over 750 popular apps.

Domain	N	Domain	N
google.com	457	flurry.com	176
admob.com	324	gmail.com	152
gstatic.com	311	google-analytics.com	148
facebook.com	256	twitter.com	138
android.com	207	mydas.mobi	113

Figure 5: The 10 most popular static domains. Each domain occurs in N-many apps.

tal. (Some apps appear in multiple categories and are only counted once.) We ran a particular RefineDroid analysis only if the target app had the appropriate standard permission, e.g., we only inferred contact columns used on apps that had the READ_CONTACTS permission.

Our results show that in terms of the permissions RefineDroid analyzes, apps are often significantly *over-privileged*, in the sense that they use far narrower capabilities than are actually available from the full platform permissions. Our results also show that in the majority of cases, an Android permission can be replaced by just a handful of fine-grained permissions, providing stronger guarantees and significantly better documentation for developers and users.

Figure 4(a) summarizes the Internet domains found by RefineDroid. The chart on the left shows how many static domains are used in each app. Despite the unlimited possibilities, apps access a median of only 8 domains. This result illustrates the extent of over-privileging inherent in the Internet permission. Moreover, it suggests that for many apps, listing per-domain permissions is plausible. We expect that in practice, we would pick a cutoff (perhaps 10); any apps under the cutoff would list individual domain permissions, and apps above the cutoff would request full Internet access.

Across all our apps, a total of 2,358 unique domains are used. In the chart on the right, we sort domains in order of popularity across our set of apps, and plot the number of apps per domain. We can see that a handful of domains are very commonly used across all apps, with a long tail of other domains. Figure 5 lists the most popular domains found by RefineDroid. These domains provide a variety of services to apps, including advertising, analytics, and social network access. It could be useful to tailor specific fined-grained permissions for groups of popular domains, e.g., advertising; we leave this for future work.

Figure 4(b) summarizes contact columns access, in the same format as part (a). In the chart on the left, we see that many apps access only a few columns, suggesting *ContactCol(c)* permissions are practical. We investigated the spike of apps using 15 columns and found that nine of these apps are actually various versions of Verizon Navigator that target different devices, and thus the spike is mostly an artifact of the data set. Note that RefineDroid finds no use of contacts permissions in 2 apps; this number can include both apps that do not use the declared permission and apps that access contacts through a deprecated API.

Android provides 79 possible column names in the contacts database. Of these, RefineDroid finds that only 73 are accessed by apps in the test set. From the right chart in Figure 4(b), we see that a few columns are very popular, and then there is a long tail. The two most popular columns are *_id* and *type*, which store metadata. Other popular columns names include the generic *data1* and *data2*, which can contain different kinds of data depending on the URI used to create the cursor, as well as *display_name* and *title*. RefineDroid resolves many of the *dataN* occurrences to more specific column names by tracking the URI values through constant propagation, but the URI is often dynamically generated, for example when contacts are selected by the user through the Android contact picker.

Figure 4(c) shows the results of RefineDroid for phone state information. RefineDroid separately tracks 16 pieces

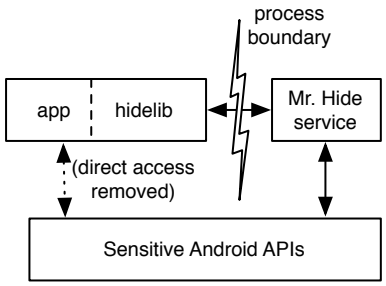


Figure 6: Mr. Hide architecture

of phone state that can be accessed via the READ_PHONE_STATE permission. We can see (left chart) that apps tend to access a very small amount of this information. We can also see (right chart) that just a few items of phone state dominate in popularity. The most commonly accessed phone state is the device's unique ID, and this occurs in twice as many apps as access to any other phone state information. Other popular pieces of information are the phone number and the current phone-call state.

Finally, Figure 4(d) indicates that (left chart) apps typically write to two or fewer settings. We also found that 44 apps are over-privileged, acquiring the WRITE_SETTINGS permission but never using it. The chart on the right shows the relative popularity of system settings. The most popular one is for the ringtone, followed by settings for screen brightness and airplane mode.

4. ENFORCING FINE-GRAINED PERMS.

One way to incorporate the fine-grained permissions we introduced in Section 2 into Android would be to modify the platform, as has been done in related prior work [4, 23, 32, 21]. However, this would require users to jailbreak their phones, putting the approach out of reach for many in practice. Moreover, it reduces flexibility, as adding new permissions requires updating the phone OS.

Instead, we decided to take a different approach that uses existing, app-level permission mechanisms. We developed Mr. Hide, a set of Android services that enforces fine-grained permissions by interposing a new API between underlying resources and client apps. Mr. Hide leverages Android's support for creating new permissions, which are then dynamically enforced in the Mr. Hide API. Mr. Hide runs in its own process, and Android's process separation ensures that client apps cannot subvert Mr. Hide's narrow API.

4.1 Mr. Hide overview

Figure 6 overviews the Mr. Hide architecture, which contains two main components: a set of services that interact with a client app and hardware resources, and a client-side library called hidelib that provides a clean interface to each service. hidelib is a drop-in replacement for sensitive Android APIs and makes usage of Mr. Hide transparent to clients apps. Mr. Hide is implemented as a group of process-isolated Android services each accessing only a single Android permission; this architecture simplifies the implementation of each component and is intended to facilitate audit for correctness and trustworthiness. Currently Mr. Hide is designed to run on Android version 2.3.6.

Permissions. Mr. Hide uses Android's permission framework to define its own set of permissions, which apps then request in the same way as system permissions. These Mr. Hide permissions behave just like platform permissions—they are displayed to the user at installation time, and no additional permissions can be acquired at run time. When responding to requests, a Mr. Hide service checks the calling app's permissions and throws standard security exceptions as needed.

Apps may also include native code, although in our experience this is uncommon (except for rendering in games, which is not security-sensitive). Mr. Hide provides no special interfaces for native code, but note that native code must still have permission to access sensitive resources. Thus if we remove a platform permission and replace it with a Mr. Hide permission, we can be certain the native code no longer has the access granted by the platform permission.

Binding to Mr. Hide services. Clients interact with Mr. Hide services using an asynchronous message-passing protocol to establish a *binding* that then enables synchronous remote procedure calls. Most hidelib methods are implemented on top of the synchronous interface to maintain compatibility with built-in Android APIs. hidelib provides code to establish necessary service bindings on app startup, to display a splash screen while waiting for connections to be completed, and to subsequently pass control to the app's distinguished "launcher" activity. Currently, hidelib does not support entry points other than the launcher activity, though this could be added if needed.

Parameterized permissions. Android does not directly support parameterized permissions such as *InternetURL(d)*, but we can encode these using a *permission tree*, which is a family of permissions whose names share a common prefix. For example, *InternetURL(*google.com*)* is represented in Mr. Hide as an instance of class *hidelib.permission.net.google_com*, which is part of the *hidelib.permission.net* tree. Note that the permissions provided by a service such as Mr. Hide need to be defined by the time that the service's clients are installed. Mr. Hide contains a GUI for adding permissions needed by new clients, as well as a predefined list of useful *InternetURL(d)* permissions.

4.2 Permission implementations

Mr. Hide includes implementations of the five fine-grained permissions introduced in Section 2; we discuss some of the more interesting implementation details next.

InternetURL(d). Android apps access the Internet using libraries that wrap low-level native code that interfaces with Linux via system calls and with cryptographic libraries used for SSL sockets. hidelib virtualizes these low-level components, implementing native calls declared in the classes InternetAddress, OSNetworkSystem, and NativeCrypto and forwarding them to a Mr. Hide service. Structures that cannot be marshaled for RPC, such as file handles and SSL contexts, are represented using unique proxy values, which the Mr. Hide service maps to, e.g., Linux file handles. Mr. Hide performs appropriate access control checks on the calling app before establishing socket connections or performing DNS lookups.

Most apps do not directly access low-level code, but in-

stead rely on high-level, built-in libraries for network access. hidelib includes work-alike libraries we built by recompiling the original sources after making suitable source-level changes both automatically (e.g., renaming classes and methods) and manually (e.g., calling our native code wrappers). In total, hidelib replaces large parts of the functionality of java.net, org.apache, and javax.net. Because these interfaces are large, we do not have complete coverage of all methods, and this occasionally leads to observable differences in app behavior. We also do not currently support android.webkit. (Newer versions of Android include hooks for controlling webkit's use of sockets, so in future work we plan to update our Android version and use these hooks to support webkit.)

ContactCol(c). Contacts are implemented on Android as a content provider database. Apps address content providers using well known URIs that are defined as library constants. For instance, ContactsContract.Contacts.CONTENT_URI maps to content://contacts, the database of all contacts.

Mr. Hide implements a new content provider that exposes the same interface as the built-in contacts provider, but is identified by a different URI. This content provider filters query results to ensure the only returned fields are those client apps have permission for. Finally, to maintain compatibility with legacy code, hidelib provides work-alike classes where, for instance, hidelib.ContactsContract.CONTENT_URI maps to Mr. Hide's URI for contacts.

LocationBlock. Apps typically access location data via a LocationManager object that allows clients to request asynchronous callbacks with current location information. To implement *LocationBlock*, hidelib provides a replacement LocationManager that passes callback registration requests to a Mr. Hide service. Mr. Hide polls location information data and uses remote procedure calls to invoke client callbacks. Location coordinates returned by Mr. Hide are truncated to provide approximately 150m resolution. Mr. Hide and hidelib also support a few alternative ways to read location, including via the TelephonyManager class and the method LocationManager.getLastKnownLocation().

ReadPhoneState(p). Currently Mr. Hide implements a single instantiation of the *ReadPhoneState(p)* permission, for the case when the accessed state is the phone's unique ID. As described in Section 3.2, access to the unique ID is by far the most popular usage for Android's READ_PHONE_STATE permission. The Mr. Hide implementation could simply wrap a call to the Android method TelephonyManager.getDeviceId(). However, for added security Mr. Hide instead returns a fixed value that is not the device's actual id. This could easily be generalized to a range of policies, such as returning a random value each time, returning a per-app randomized value, returning a per-app-author randomized value, etc.

WriteSettings(s). Currently Mr. Hide implements a single instantiation of the *WriteSettings(s)* permission, for the case when the device's ringtones are changed. As described in Section 3.2, this is the most popular usage for Android's WRITE_SETTINGS permission. Device ringtones can be set in two ways on Android: via a RingtoneManager object and by calling the method Settings.System.putString; the former is actually implemented on top of the latter. Thus, hidelib provides replacement RingtoneManager and Settings.System classes, each of which contacts a Mr. Hide service to change a ringtone when requested. The Mr. Hide service is a thin wrapper that checks for the necessary permission and forwards ringtone updates to Settings.System.putString(). It would be straightforward to add the ability to write other settings through this Mr. Hide wrapper.

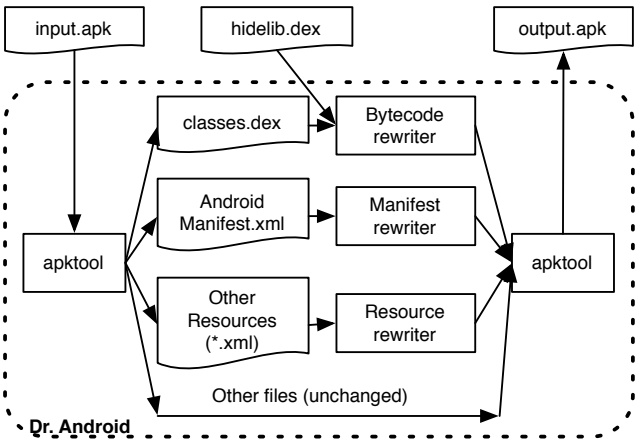


Figure 7: Dr. Android architecture

5. FINE-GRAINED PERMS. FOR USERS

Mr. Hide makes it easy for developers to employ fine-grained permissions in their apps, and hidelib allows them to port existing app source code with minimal effort. In this section, we introduce Dr. Android[1], a tool that allows users to update app binaries to use Mr. Hide in place of the built-in Android API. Because Dr. Android retrofits apps without needing source, it allows app users to more directly control the security of the apps they run. For example, an app user could replace the default Android permissions in a downloaded app with fine-grained variants, obtaining stronger security guarantees without being beholden to the app developer. Users could also easily experiment with multiple fine-grained policies to explore the tradeoff between security and utility, and different users could make different choices within this spectrum. For purposes of this paper, we use Dr. Android in combination with RefineDroid, using the latter to infer fine-grained permissions that we add with the former (full details in Section 6).

Figure 7 illustrates Dr. Android's architecture. In addition to bundling and unbundling app components, Dr. Android performs three kinds of transformations. First, it modifies the app's Dalvik bytecode (classes.dex) to use Mr. Hide and injects hidelib code into the app. Second, Dr. Android modifies the app's *manifest* to remove Android permissions, add hidelib permissions, and modify declared program entry points. Third, Dr. Android modifies resource files that define user interface layouts so that hidelib classes with UI elements are referenced as needed. The rest of this section describes Dr. Android in more detail.

[1]Redexer, the general-purpose Dalvik transformation tool Dr. Android is built on top of, can be downloaded at http://www.cs.umd.edu/projects/PL/redexer/.

```
1  type perm = InternetUrl | LocationBlock | ...
2  let (perm_map : perm → (string * string) list) = ...
3  let (perm_manager : perm → (string * string) list) = ...
4
5  let rewrite (dx : dex) (ps : perm list) : unit =
6    merge_hidelib dx;
7    replace_classes dx (List.concat (List.map perm_map ps))
8    replace_managers dx ps perm_managers;
9    if List.mem ContactCol ps then
10     replace_contact_strings dx;
11   insert_service_binding dx
```

Figure 8: Bytecode rewriter pseudocode

Reading, writing, and signing apps. Dr. Android relies on apktool [17] to read and write apk files. Android apps are digitally signed and modified binaries must be re-signed to run on a phone. Dr. Android signs apps with fresh cryptographic keys which inevitably differ from the original keys. Fortunately, app signatures are mostly used to establish trust relationships between different apps produced by a single vendor and signed by the same key.[2] We can preserve these relationships by consistently signing apps from the same original authors with the same new key but have not implemented this feature as it was not necessary for any of our case studies.

Rewriting manifest and resource files. Rewriting the XML-formatted manifest and resource files is straightforward. Existing permissions appear as <permission> elements in the manifest, and Dr. Android removes specified Android permission elements and replaces them with appropriate Mr. Hide permissions. Mr. Hide also has to modify some resource files that contain XML declarations used to instantiate Java classes at runtime. These are commonly used to declare user interface elements. As with manifest rewriting, Dr. Android replaces XML elements referencing Android components with Mr. Hide equivalents.

Rewriting bytecode. The core of Dr. Android is the bytecode transformer, which contains approximately 10K lines of OCaml code that parses a bytecode file into an in-memory data structure, modifies that data structure, and unparses it to produce an output bytecode file.

Dalvik bytecode files are structured as a series of indexed "pools," each of which contain a single kind of element: strings, types, method signatures, method definitions, etc. Elements in one pool can refer to elements in another. For instance, instructions in the method definition pool refer to type names stored in the string pool.

Figure 8 shows (a simplified version of) the core OCaml rewrite function that applies Mr. Hide rewrites to bytecode argument dx. Rewriting is guided by list ps of fine-grained permissions desired for the rewritten app. There are several key steps that apply to rewriting any app for any fine-grained permission. First, Line 6 merges hidelib with dx.

[2]As far as we are aware, changing app signatures does not affect ad revenue. The ad libraries we have looked at track ads shown based on a developer-specific ID that is part of the app's source code and is passed directly to the ad library. This ID is not modified by Dr. Android.

Task	Orig (s)	Transformed (s)	Slowdown
Internet	16.241	20.252	25%
Contacts	0.634	0.953	50%
Location	15.004	19.407	29%
Ringtone	1.257	1.382	10%

Figure 9: Microbenchmark performance results

Here it is necessary to maintain the invariant that output pools are both duplicate-free and sorted. Duplicates can arise when (e.g.) both dx and hidelib refer to the string representing type void. Next, Line 7 replaces references to Android API classes (e.g. android...LocationManager) with corresponding hidelib classes (e.g. hidelib...LocationManager). Because individual Android APIs may provide access to both privileged and unprivileged functionality, the scope of rewriting can be adjusted to avoid creating unnecessary IPC calls. Lastly, Line 11 inserts code to establish a synchronous binding between the client app and Mr. Hide (see Section 4.1). To this end Dr. Android creates an app entry point in library class hidelib.App that connects to Mr. Hide services. Dr. Android also modifies hidelib.App so that it inherits from the client's App class (if any), ensuring that client code runs as needed. This is simpler and more robust than modifying a client app's instruction stream directly.

There are also some rewriting steps that are only used for particular permissions. First, most apps connect to Android's contacts content provider using URI names declared as constants in library classes. In these instances Line 7 performs class replacement to swap URIs and redirect contacts access to Mr. Hide. Other apps construct URIs out of hardcoded strings, and Line 10 rewrites such contact-related strings in the string pool to redirect contacts queries to Mr. Hide.

Second, manager classes, such as LocationManager, are used to read system data and issue callbacks. Clients obtain instances of manager classes by calling Context.getSystemService() and downcasting. Line 8 uses heuristics to detect this pattern and substitute hidelib's managers for the platform's. These heuristics are necessary for permissions *LocationBlock* and *ReadPhoneState(p)*.

6. EXPERIMENTS

We evaluated RefineDroid, Mr. Hide, and Dr. Android in several ways. First, we performed informal testing on Mr. Hide to ensure it implements its permissions correctly. For example, we verified that only permitted domains can be accessed with *InternetURL(·)*. Second, we used microbenchmarks to measure the overhead of using Dr. Android and Mr. Hide compared to using direct system calls. Third, we ran RefineDroid on a set of apps to evaluate its false positive and negative rate. Finally, we ran Dr. Android on the same set of apps and evaluated the correctness and usability of apps transformed with Dr. Android to use Mr. Hide.

6.1 Microbenchmark performance

To measure the overhead of the interprocess communication entailed by Mr. Hide, we developed an app that can measure the time required to retrieve data from a sequence of 100 distinct web pages on the local network; make 100 queries to the contact manager; request 10 location updates; and change ringtone paths 1,000 times.

Figure 9 shows the running times of these microbench-

marks before and after applying Dr. Android and Mr. Hide, and the slowdown ratio. These results are the average of 5 runs on a Google Nexus S phone. As could be expected, the slowdowns are fairly significant, as the interprocess communication required by Mr. Hide is quite an expensive operation. Nevertheless, the cost of this overhead is incurred only at relatively infrequent calls into system-level code, and it is rarely an issue in practice, as discussed below.

6.2 RefineDroid, Dr. Android, and Mr. Hide on real apps

To determine how well our tools work in practice, we selected 14 case study Android apps from a variety of Google Play categories, with a range of different functionality. We deliberately picked apps that exercise the five fine-grained permissions we implemented in a variety of ways, that were amenable to high-coverage manual testing, and that fit within the limitations of the Dr. Android and Mr. Hide implementation.

Rewriting apps to use hidelib. When a user wishes to run Dr. Android on an app, he first uses RefineDroid to determine the set of permissions that are applicable for the app. After ascertaining which fine grained permissions are applicable for an app, a user will run Dr. Android with the appropriate arguments.

For example, when run on the Amazon app, RefineDroid reports that it is possible to replace the location, Internet, and phone state permissions with the finer grained fuzzed location, restricted Internet access, and unique ID permissions. To transform the app, we then invoke Dr. Android:

```
1 ./drAndroid --cmd rewrite
2   com.amazon.mShop.android-23.apk
3   --perms internet loc uniqid
```

Dr. Android produces an output file in the form of an APK file, which can be then installed on the device and run as normal.

Scope of rewriting. Figure 10 summarizes the the rewrites performed with Dr. Android. Out of 35 occurrences of the Android permissions that we study, 31 can be replaced by fine-grained permissions with minimal degradation to user experience. As an example of why such a replacement is not always appropriate, consider the ASTRO file manager, which requires *INTERNET* permission so that users can connect to arbitrary hosts and upload or download files. This particular behavior is not captured by any of our fine-grained permissions, but is captured by the platform *INTERNET* permission.

Correctness of rewriting. Our experiments show Dr. Android is capable of processing commercially published apps with acceptable performance, yielding correct Dalvik binaries. Columns 2–4 in Figure 11 describe the size of apps before rewriting, including the apk size, the size of classes.dex after the apk is unpacked, and the number of Dalvik bytecode instructions. The next two columns report the number of changes applied by Dr. Android and the running time of Dr. Android. A single change comprises either modifying an instruction to refer to a different class; replacement of a manager class, or modification of a string in the string table. Reported performance is based on one run on a 2.8 GHz Intel i7 860 with 16 GB RAM, running Ubuntu 11.04 64-bit. From these results, we see that Dr. Android is easily fast enough for offline use.

The Android platform performs lightweight bytecode verification when apps are installed on a phone, and more thorough verification during application runtime. These verification phases test various well-formedness constraints on Dalvik files. All apps in Figure 11 install and run without verification errors, giving confidence that Dr. Android's transformations are structurally correct.

Behavior of rewritten apps. Each Android Activity displays its own user interface screen to the user and so represents a distinct piece of functionality supported by the app. We manually experimented with each app to understand the behavior of each of its activities. We then conducted the same test in a rewritten app to verify that the expected buttons, menus, and other displays appear on an activity's UI and that clicking on these items produces the expected behavior (e.g., displaying a different Activity, quitting the application, etc.).

We manually ran the apps, and for each app we attempted to visit all accessible activities and use every feature we could reasonably access. Note that we did not explore such features that would be costly to test (for example, completing a purchase), but completed all steps leading to these actions (for example, loading a set of items into a shopping cart). During testing, we found that almost all activities of applications function normally, with no observable changes.

We did find small differences in certain activities that use the Internet, due to limitations of Mr. Hide. Google Sky Map and Shazam use a WebView widget, which we do not support; these views show placeholder text after rewriting. This issue also affects a number of apps which use libraries to display ads, as ads are displayed within a WebView container. We found that Angry Birds, Angry Birds Rio, Brightest Flashlight, Ultimate Flashlight, and Gasbuddy had this issue. Later updates to the Android OS platform include hooks for intercepting and handling WebView network requests; we plan WebView support for updated platforms as future work.

Transformed apps may experience a noticeable delay on startup while the app connects to Mr. Hide services. After the initial connection is established, application performance, measured by informal observation, is similar for transformed and untransformed applications. We speculate that this is because the cost of using Mr. Hide is amortized over the cost of other operations, and because the test apps are designed to be interactive, spending a large share of time waiting for user input.

RefineDroid precision and recall. We collected log data during the experiments described above, and used it to compare RefineDroid's static analysis results with the runtime behavior of apps under test.

The rightmost columns in Figure 11 summarize these results. Under heading Domains, column # reports the number of *InternetURL(d)* permissions found by RefineDroid for each app, and columns FP and FN report the number of false positives and negatives in RefineDroid's output. Here a false positive indicates that RefineDroid reports that an app may use a fine-grained permission, but that permission was not used during testing. A false negative is recorded

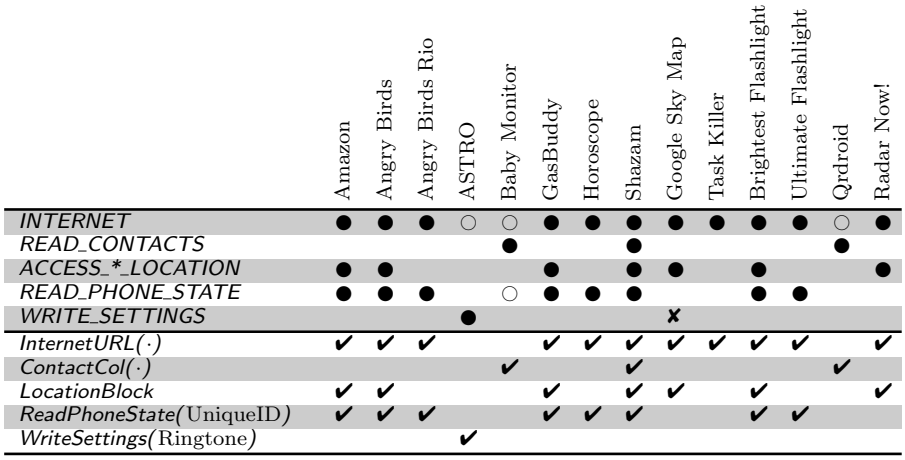


Figure 10: Fine-grained permissions for 14 apps from Google Play. Notation ● indicates a built-in Android permission that can be replaced by one or more fine-grained permissions, ○ indicates policies that cannot obviously be removed, and ✘ indicates that a permission can be removed because it does not appear to be used at all. ✔ indicates fine-grained policies needed according to RefineDroid and which we observed being used dynamically.

Name	Apk (KB)	Dex (KB)	# Ins	# Chg	Tm (s)	Domains #	Domains FP	Domains FN	Contacts #	Contacts FP	Contacts FN
Amazon	1,607	2,288	114,691	174	17.86	9	7	0			
Angry Birds	993	15,018	79,311	760	11.44	12	9	0			
Angry Birds Rio	2,081	22,716	173,441	968	21.92	12	10	1			
ASTRO	1,428	2,348	149,911	695	18.30						
Baby Monitor	163	781	12,378	1	3.81				5	1	1
Gas Buddy	781	1,269	67,514	222	11.81	6	5	0			
Horoscope	844	3,731	92,441	829	12.73	17	16	0			
Shazam	2,641	3,904	259,643	778	30.67	20	17	0			
Google Sky Map	459	2,212	33,355	193	8.38	4	3	0			
Task Killer	129	99	9,696	76	6.10	4	4	0			
Brightest Flashlight	1,870	1,756	174,159	1,265	18.94	21	21	1	6	6	0
Ultimate Flashlight	485	1,287	46,878	464	8.26	10	9	0			
Qrdroid	922	3,802	105,400	11	9.05				8	2	0
Radar Now!	379	569	26,706	121	7.66	5	3	0			

Figure 11: RefineDroid, Dr. Android and Mr. Hide results on apps

when RefineDroid fails to discover a fine-grained permission that an app needs to run. Columns grouped under heading Contacts report this information for *ContactCol(c)*.

RefineDroid has a relatively low rate of false negatives and a higher rate of false positives. The low rate of false negatives suggests that policies produced by RefineDroid may serve as effective near-upper bounds on an app's behavior. This is particularly useful in the context of rewriting, as false negatives can prevent a rewritten app from executing successfully. We examined the false positives in detail, and found that the majority are domains for third party ads or advertising developers own apps. We found that these false positives were generally reachable, but not within the configuration of the test cases. For example, Amazon uses a large set of domains which deal with access from different countries based on the IP address the device is using.

We also looked at whether we could use better constant propagation to eliminate false positives, rather than the simple analysis we use currently. We found that it would require developing a much more sophisticated constant propagation system that included inter-procedural support, modeling heap, handling format strings, and modeling Android's intent system. While we may investigate this in future work, based on our study we believe it may not be worth the effort, since most of the "false positives" found by the simpler analysis are in fact reachable.

7. RELATED WORK

Several other researchers have proposed mechanisms to refine or reduce permissions in Android. Similar to Dr. Android, Aurasium is a tool that can transform apps to, among other things, intercept system calls to enforce security policies [28]. (We note that an earlier technical report on Dr. Android and Mr. Hide was published before Aurasium [22].) One limitation of Aurasium is that the run-time monitors it inserts execute in the same process as the app, and hence are potentially subject to circumvention. Dr. Android and Mr. Hide prevent this by removing the original permissions from the app and perform fine-grained permission checking in the Mr. Hide service, which runs in a separate process.

MockDroid allows users to replace an app's view of certain private data with fake information [4]. AppFence similarly lets users provide mock data to apps requesting private information, and can also ensure private data that is released to apps does not leave the device [21]. TISSA gives users detailed control over an app's access to selected private data (phone identity, location, contacts, and the call log), letting the user decide whether the app can see the

true data, empty data, anonymized data, or mock data [32]. Apex is similar, and also lets the user enforce simple constraints such as the number of times per day a resource may be accessed [23]. CRePE suggests policies along with context attributes so that users can decide when to grant apps permissions [9]. AdDroid proposes using trusted OS code and a new permission to mediate interactions between apps and advertisers [26]. A limitation of all these approaches is that they require modifying Android platform, and hence to be used in practice must either be adopted by Google or device providers, or must be run on rooted phones. In contrast, Dr. Android and Mr. Hide run on stock, unmodified Android systems.

Researchers have also developed other ways to enhance Android's overall security framework. Kirin employs a set of user-defined security rules to flag potential malware at install time [13]. Saint enriches permissions on Android to support a variety of installation constraints, e.g., a permission can include a whitelist of apps that may request it [25]. These approaches are complementary to our system, as they take the platform permissions as is and do not refine them.

There have been several studies of apps in relation to the Android permissions they acquire. Barrera et al. [3] observe that only a small number of Android permissions are widely acquired but that some of these, in particular Internet permissions, are overly broad (as we have also found). Vidas et al. [30] and Felt et al. [15] each describe a static analysis tool that identifies acquired permissions that are never used. Our RefineDroid tool has similar functionality but analyzes apps in terms of our fine-grained permissions rather than the original platform permissions.

Finally, several tools have been developed to identify specific security vulnerabilities in Android apps. QUIRE [10], IPC Inspection [14], and XManDroid [5] address the problem of privilege escalation, in which an app is tricked into providing sensitive capabilities to another app. Woodpecker [20] uses dataflow analysis to find capability leaks on Android phones. TaintDroid tracks the flow of sensitive information [11]. Ded [12], a Dalvik-to-Java decompiler, has been used to discover previously undisclosed device identifier leaks. ComDroid [8] finds vulnerabilities related to inter-application communication. Crowdroid [7], DroidRanger [19], DroidScope [31], and Paranoid Android [27] suggest several ways to detect viruses and malware in the market. Porscha [24] provides a policy-oriented digital rights management mechanism. TrustDroid [6] proposes domain isolation on Android. These tools all identify app vulnerabilities in the context of the current set of Android permissions. Dr. Android and Mr. Hide are complementary, replacing existing permissions with finer-grained ones to reduce or eliminate such vulnerabilities.

8. CONCLUSIONS AND FUTURE WORK

We presented a new approach to understanding and implementing fine-grained permissions on the Android operating system. We began by creating a taxonomy with four categories that group standard Android permissions by the behaviors they allow, and, for each category, proposing new fine-grained variants. We then presented RefineDroid, Mr. Hide, and Dr. Android, tools that infer and implement finer-grained permissions on Android without requiring platform modifications. Using RefineDroid, we conducted a survey showing that fine-grained permissions are suitable for many popular apps. We then applied Dr. Android to transform a range of apps to use Mr. Hide. Our results suggest that fine-grained permissions via Dr. Android and Mr. Hide provide stronger privacy and security guarantees while retaining application functionality and performance. In the future, we plan to extend our approach with support for additional permissions and to perform studies of the usability of finer-grained permissions.

While we have explored using RefineDroid, Mr. Hide, and Dr. Android on one set of tasks, we believe these tools have many other uses. RefineDroid (likely enhanced with more sophisticated static analysis) could be used to look for suspicious permission use, e.g., access to unexpected Internet domains. Since Mr. Hide provides system call interposition, we could use it to implement other kinds of security policies, e.g., inserting mock data [4], changing policies at run-time, or permitting exceptional access. Finally, Dr. Android provides quite general support for modifying Dalvik bytecode, and we could use it to perform other kinds of dynamic program analysis.

Acknowledgements

Thanks to Yixin Zhu for technical contributions to a precursor of this work and to Philip Phelps for help with the microbenchmarking app. This work was supported in part by NSF CNS-1064997 and by a research award from Google.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Android Police. Massive Security Vulnerability In HTC Android Devices (EVO 3D, 4G, Thunderbolt, Others) Exposes Phone Numbers, GPS, SMS, Emails Addresses, Much More, Oct. 2011. http://www.androidpolice.com/2011/10/01/massive-security-vulnerability-in-htc-android-devices-evo-3d-4g-thunderbolt-others-exposes-phone-numbers-gps-sms-emails-addresses-much-more.

[3] D. Barrera, H. Kayacik, P. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *CCS*, pages 73–84, 2010.

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

[5] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb 2012.

[6] S. Bugiel, L. Davi, A. Dmitrienko, and S. Heuser. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, 2011.

[7] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 15–26, 2011.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *MobiSys*, 2011.

[9] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: context-related policy enforcement for android. In *Proceedings of the 13th international conference on Information security*, pages 331–345, 2011.

[10] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011.

[11] 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.

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security*, 2011.

[13] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS*, pages 235–245, 2009.

[14] A. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security*, 2011.

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

[16] Gartner. Gartner Says Sales of Mobile Devices Grew 5.6 Percent in Third Quarter of 2011; Smartphone Sales Increased 42 Percent, Nov. 15 2011. http://www.gartner.com/it/page.jsp?id=1848514.

[17] Google. Tool for reengineering Android apk files. http://code.google.com/p/android-apktool/.

[18] Google. 10 Billion Android Market Downloads and Counting, Dec. 2011. http://android-developers.blogspot.com/2011/12/10-billion-android-market-downloads-and.html.

[19] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS*, 2012.

[20] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS*, 2012.

[21] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *CCS*, pages 639–652, 2011.

[22] J. Jeon, K. K. Micinski, J. A. Vaughan, N. Reddy, Y. Zhu, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. Technical Report CS-TR-5006, Department of Computer Science, University of Maryland, College Park, December 2011.

[23] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS*, pages 328–332, 2010.

[24] M. Ongtang, K. Butler, and P. McDaniel. Porscha: policy oriented secure content handling in android. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 221–230, 2010.

[25] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *ACSAC*, pages 340–349, 2009.

[26] P. Pearce, A. P. Felt, G. Nunez, , and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2012.

[27] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, 2010.

[28] H. S. Rubin Xu and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[29] J. H. Saltzer. Protection and the control of information sharing in Multics. *Comm. of the ACM*, 17(7):388–402, July 1974.

[30] T. Vidas, N. Christin, and L. F. Cranor. Curbing Android Permission Creep. In *W2SP*, 2011.

[31] L. K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[32] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on android). *Trust and Trustworthy Computing*, pages 93–107, 2011.