

# Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android

Jinseong Jeon<sup>\*</sup>  
jsjeon@cs.umd.edu

Kristopher K. Micinski<sup>\*</sup>  
micinski@cs.umd.edu

Jeffrey A. Vaughan<sup>†</sup>  
jeff@cs.ucla.edu

Nikhilesh Reddy<sup>†</sup>  
nreddy@cs.ucla.edu

Yixin Zhu<sup>†</sup>

Jeffrey S. Foster<sup>\*</sup>  
jfoster@cs.umd.edu

Todd Millstein<sup>†</sup>  
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. We have found that Android’s current permissions are often overly broad, providing apps with more access than they truly require. This deviation from least privilege increases the threat from vulnerabilities and malware. To address this issue, we present a novel system that can replace existing platform permissions with finer-grained ones. A key property of our approach is that it runs today, on stock Android devices, requiring no platform modifications. Our solution is composed of two parts: Mr. Hide, which runs in a separate process on a device and provides access to sensitive data as a service; and Dr. Android (Dalvik Rewriter for Android), a tool that transforms existing Android apps to access sensitive resources via Mr. Hide rather than directly through the system. Together, Dr. Android and Mr. Hide can completely remove several of an app’s existing permissions and replace them with finer-grained ones, leveraging the platform to provide complete mediation for protected resources. We evaluated our ideas on several popular, free Android apps. We found that we can replace many commonly used “dangerous” permissions with finer-grained permissions. Moreover, apps transformed to use these finer-grained permissions run largely as expected, with reasonable performance overhead.

## 1. INTRODUCTION

Google’s Android is the most popular smartphone platform, running on 52.5% of all smartphones [11], and with more than 10 billion apps downloaded from the Market [13]. Android takes an “open-publish” approach to app distribution, in which any app can be installed on any phone. To help address security concerns, Android protects access to sensitive resources—including

the Internet, GPS, and telephony—with *permissions*. When an app is installed, the permissions it requests are shown to the user, who then decides whether to proceed with installation. No additional permissions may be acquired when an app runs.

While Android permissions provide an important level of security, the available permissions are often 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 `www.amazon.com`. In fact, this permission allows apps to connect to local sockets on the phone as well, which led to a recently publicized security hole whereby any app with Internet permission could access detailed system logs on HTC Android phones [1].

In this paper, we present a new system for replacing coarse Android platform permissions with finer-grained permissions that lower needed privilege levels, decreasing the potential threat from vulnerabilities and malicious apps. Our system is comprised of two novel parts. First, we introduce Mr. Hide, an Android service that protects sensitive device capabilities with fine-grained permissions that the service dynamically enforces. For example, Mr. Hide includes an API for accessing the Internet but protects this API with a new permission *InternetURL(d)*, which only grants access to the Internet domain *d*. The Amazon app mentioned above can use Mr. Hide to access the Internet by replacing Android’s full *Internet* permission with the much weaker *InternetURL(www.amazon.com)* permission, providing increased confidence to both the developer and users.

A key feature of Mr. Hide is that it runs on stock Android phones, requiring no platform modifications. Mr. Hide leverages several existing features of Android. First, it creates custom permissions for its various services. Note that although Android does not directly support *parametrized* permissions such as *InternetURL(d)*,

<sup>\*</sup>University of Maryland, College Park

<sup>†</sup>University of California, Los Angeles

we can use a simple encoding to represent them using flat permissions names. Second, Mr. Hide runs in its own process, so while it must be trusted with standard, full Android permissions, Android’s process separation ensures that client apps only access the underlying resources through Mr. Hide’s narrow API. Further, Android’s existing permission framework ensures complete mediation—if an app removes the standard *Internet* permission, the app cannot directly access the Internet, either via the standard Android APIs or via lower-level mechanisms such as dynamic loading, reflection, and native code. Finally, it is very easy to port clients to Mr. Hide, since we provide an adapter layer that is a drop-in replacement for sensitive platform APIs as well as common third-party libraries like AdMob, an advertising library; the adapter layer takes care of all necessary communication with the Mr. Hide service.

Second, we describe Dr. Android (Dalvik Rewriter for Android), a tool that retrofits existing Android application package files (apks) to use Mr. Hide (the Hide interface to the droid environment). In combination, the two components allow users to obtain the security benefits of finer-grained permissions on downloaded apps, without needing source code or recompilation. The input to Dr. Android is an apk and a list of finer-grained permissions to retrofit. Surprisingly, a small number of reusable high-level transformations suffice in order to replace an existing Android permission with a Mr. Hide permission. In particular, Dr. Android includes the ability to change the permission set of an apk; rename platform API references in bytecode to refer to their Mr. Hide counterparts; append the Mr. Hide adapter layer to the code in the apk file; and insert code to bind to the Mr. Hide service, among other, similar transformations. Together, these transformations constitute a simple but powerful API for retrofitting existing apks to use fine-grained permissions, and we expect the same transformations can be used for new permissions added to Mr. Hide in the future.

To evaluate our ideas, we studied the permissions used in several of the most popular, free Android apps. These apps use a wide range of permissions, so we selected 7 of commonly used *dangerous* (as defined by the Android platform) permissions requested by the apps. We found that the requested Android permissions are often much broader than necessary for the app’s desired usage. Specifically, we identified a set of 7 finer-grained permissions that, together, can completely replace 61% of the uses of the original dangerous permissions we studied and partially replace an additional 12% of them. We have implemented our proposed permissions in Mr. Hide and developed transformations for them in Dr. Android. Each fine-grained permission can be used in at least 3 of our apps, and the most common fine-grained permission can be used in 12 of the apps.

We evaluated our implementation of Dr. Android and Mr. Hide in several ways. First, we used microbenchmarking to measure the performance of the Mr. Hide service and found overheads of 10–50% compared to directly using sensitive platform APIs, likely due to the interprocess communication required to use Mr. Hide. However, we found that in practical use—specifically, in manual testing of the modified apps—perceived performance under Dr. Android and Mr. Hide is the same as for the original app. Next, we applied Dr. Android to our apps, finding that it performs well, taking at most around one minute to transform an app, and producing code that passes the Dalvik bytecode verifier. Finally, we evaluated the correctness of the transformed apps using automated and manual testing. We found that all automated tests passed, and we observed only a few differences from the original apps under manual testing. The differences we detected are due to limitations in Mr. Hide’s support for the Android networking API and for complex ads, and they are not fundamental. We expect that even with these limitations as a research prototype, many users would be willing to accept these differences in exchange for increased security.

Dr. Android and Mr. Hide have several advantages over prior approaches for Android [3, 16, 19, 15]. First, our approach does not require any modification to the current Android platform, and it can run on stock devices without jailbreaking. Second, we can easily add new fine-grained permissions over time, which we believe is essential; different applications and/or users will have different privacy requirements, and it is unlikely that a fixed vocabulary of permissions will suffice, especially given the rapid evolution of the Android platform. Finally, a given fine-grained permission could have multiple independent associated services that enforce the permission. We envision an ecosystem in which many different parties provide services for commonly desired fine-grained permissions, which developers and users can choose among. Such services are far simpler than full apps and hence are much easier to audit for security.

In summary, we believe our results show that Dr. Android and Mr. Hide provide stronger privacy guarantees for users while retaining application functionality and performance.

## 2. MOTIVATING EXAMPLE

In this section we motivate our approach by illustrating how it can be used to improve the security and privacy of a particular Android app. Consider installing Horoscope ([horoscope.fr](http://horoscope.fr)) one of the most popular free Android applications. The app allows users to check the horoscope for different astrological symbols for today, tomorrow, and the current week, month, and year.

Despite the simplicity of its intended functionality, version 1.5.2 of the Horoscope app requests several per-

missions that are classified as *dangerous* by Android. These include the ability to access the Internet; read the phone state; access fine (GPS-based) and coarse (network-based) location information; and write system settings. These permissions allow the Horoscope app to access a variety of kinds of sensitive information about the user and to upload that information to any location on the Internet. The permissions also allow the app to violate integrity, for example by modifying some of the phone’s settings.

We can use Dr. Android and Mr. Hide to enforce an access-control policy for Horoscope that is much closer to its least privilege policy:

- Horoscope requires Internet permission to access a range of servers on the web. To reduce the scope of this access, Mr. Hide provides the *InternetURL(d)* permission described in the previous section. In this case, we would give the app the permission *InternetURL(horoscope.fr)* and similar permissions for other domains needed for the app.

In addition its core functionality, Horoscope uses Internet access to display ads from a range of ad servers. Mr. Hide provides two more restrictive Internet permissions, *AdsPrivate* and *AdsGeo*, to ensure that requests to one commonly used ad server, AdMob, do not leak sensitive information. The first permission permits only an advertiser id to be sent to the server, and the second permission, which we use for Horoscope, also allows the user’s location to be sent in order to obtain targeted ads.

- Horoscope reads the phone state only to access a unique phone ID that can be used to track how clients use the app. Mr. Hide provides a *UniqueID* permission and associated library for this common case, which provides a (false) device ID and disallows all other privileged state access.
- Horoscope uses location information for targeted ads, as described earlier. In Mr. Hide, precise locations are currently provided for ads, but Mr. Hide provides the ability to coarsen location information for other uses. In particular, users who do not wish to provide detailed location information can employ Mr. Hide’s *LocationBlock* permission, which provides location information (either GPS- or network-based) that is only accurate up to around 150 meters (about the distance of a city block).
- Horoscope requests permission to write system settings, but it is not clear why. In fact, we found that the app does not call any APIs that require this permission, and hence it is simply over-privileged. With Dr. Android, a user can remove this permission.

While Horoscope is a particularly good illustration of the problem with Android’s permissions, it is by no means unique. As discussed in the next section, we found many popular apps whose Android permissions can be replaced with finer-grained permissions via Dr. Android and Mr. Hide in order to improve security without affecting functionality.

### 3. FINE-GRAINED PERMISSIONS FOR ANDROID

We examined 19 of the top free apps on the Android Market to understand the potential benefits of fine-grained permissions and to identify the most promising such permissions to implement in Mr. Hide. The apps represent a range of domains, including games, utilities, online shopping, and multimedia, and were gathered at a variety of times. Each app has been downloaded at least one million times (across all its versions). We focused on a set of seven *dangerous*<sup>1</sup> permissions that are acquired by many of the apps we looked at.

Our evaluation consisted of installing and running each app to understand its functionality, reading English-language security and privacy policies and other documentation when available, and looking at app bytecode to determine which privileged methods are called. For each app, we evaluated how it uses its current permission set and identified fine-grained permissions that could replace some of these permissions.

The results of our study are summarized in Figure 1. The original apps contain 66 uses of the seven dangerous permissions we considered. For 40 (61%) of these uses, a combination of Dr. Android and Mr. Hide can remove the permission and replace it with zero or more fine-grained permissions. The resulting apps have the same functionality as the original apps but have much stronger privacy and security guarantees. An additional 8 (12%) of these uses can be similarly replaced by Mr. Hide permissions but incur some loss of functionality due only to limitations of our current implementation, discussed more below. In the remainder of this section we describe our fine-grained permissions in detail.

*Internet permissions.* The default Internet permission in Android is pervasive, used in all the apps we surveyed. At the same time, it is also the most dangerous, as it allows for arbitrary communication with any domain on the Internet. Fortunately, most apps communicate with a fixed set of domains. The fine-grained permission *InternetURL(d)* targets this common case by

<sup>1</sup>The Android platform defines normal (low risk) and dangerous (high risk) *protection levels* for permissions (as well as two other special cases we do not discuss). Dangerous permissions are always shown at app installation time, and normal permissions can be shown by clicking a disclosure triangle, but by default are hidden.

	Adv. Task Killer 1.9.6B76	Amazon 1.3.0	Angry Birds 1.5.3	Angry Birds Rio 1.0.0	ASTRO 2.5.2	Barcode (zxing) 3.53	Bubble Blast 1.0.16	Bubble Blast 2 1.0.18	Brightest Flashlight 1.9.3	Dropbox 1.1.1	ESPN ScoreCenter 2.1.3	Flashlight 3.9.9.12	FreeMusic 1.8.3	GasBuddy 1.14	Google Sky Map 1.6.1	Horoscope 1.5.2	MP3 Ringtone Maker 1.93	Shazam 2.5.3-BB70302	Words With Friends 4.60
<b>INTERNET</b> (19)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
<b>InternetURL(d)</b> (13)		●	●	●	○	●	●	○	○	○	○	○	○	○	○	○	○	○	○
<b>AdsPrivate</b> (9)	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
<b>AdsGeo</b> (5)																			
<b>READ_PHONE_STATE</b> (13)	✓					✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
<b>UniqueID</b> (6)	●								●	●	●	●					●		
<b>WAKE_LOCK</b> (9)					✓	✓		✓			✓	✓	✓	✓	✓	✓	✓	✓	✓
<b>Permission Unnecessary</b> (3)						●							●					●	
<b>ACCESS_FINE_LOCATION</b> (8)	✓							✓				✓	✓	✓	✓	✓	✓	✓	✓
<b>ACCESS_COARSE_LOCATION</b> (8)	✓							✓				✓	✓	✓	✓	✓	✓	✓	✓
<b>LocationBlock</b> (8)	●							●				●	●	●	●	●	●	●	●
<b>WRITE_SETTINGS</b> (5)					✓							✓	✓	✓	✓	✓			
<b>SetRingtone</b> (3)					●								●				●		●
<b>Permission Unnecessary</b> (2)															●	●			
<b>READ_CONTACTS</b> (4)						✓						✓					✓		✓
<b>ReadVisibleContacts</b> (4)						○						●					●		●

**AdsPrivate:** May displays ads, but without sharing personal information with advertisers.

**AdsGeo:** May displays ads and may share your location, but no other personal information, with advertisers.

**InternetURL(d):** May access the Internet services at domain  $d$ .

**LocationBlock:** May access approximate location, accurate to 150m (about one city block).

**ReadVisibleContacts:** May read contacts that are designated as visible, but no other contacts.

**SetRingtone:** May modify ringtone settings on the phone, but no other settings.

**UniqueID:** May read a random ID number for the phone, with which to track the user's usage of the app.

**Figure 1: Permission usage in a range of the top free Android apps. We consider seven dangerous Android permissions. For each such permission (in all caps), we show the fine-grained permissions (described at the bottom of the figure) that can replace it in Mr. Hide. We also record situations when a permission is unnecessary. A ✓ indicates that the original permission can be removed while a ✓ indicates that the permission is still required. A ● indicates that Dr. Android and Mr. Hide can successfully transform the app to use the fine-grained permission. A ○ indicates a permission that can be retrofitted, but with some changes to app functionality. A ○ indicates a transformation that is possible but has not been implemented.**

allowing network connections only to domain  $d$  and its subdomains. As shown in Figure 1,  $InternetURL(d)$  is useful in 13 of 19 apps. For five of these apps replacing the existing Internet permission incurs some loss of functionality. In two cases this is due to a partial implementation of SSL in Mr. Hide; and in three cases it because we do not support the `WebView` user-interface widget. It would be straightforward to add the missing interfaces to support these apps fully.

Three of the 6 apps legitimately need full Internet permissions, and thus do not use our new permission; for example, the ASTRO file manager provides an `sftp` client for transferring files to and from arbitrary domains. The remaining three apps use Internet access only for ads, which we discuss next.

One common use of the Internet permission, particu-

larly for free apps, is to communicate with third-party services that provide ads, via Android libraries such as AdMob. However, this communication poses a privacy risk for users, since any private data they allow an app to access to perform its functionality (e.g., contacts, photos, etc.) can potentially be sent to untrusted ad servers. As described in the previous section, the permissions *AdsPrivate* and *AdsGeo* address this problem by providing an adapter layer that isolates the ad functionality as a service in a separate process and restricts the flow of information to the ad server. Together, these permissions can be used by 14 of the 19 apps we surveyed. In half of these cases the result is functionally identical (modulo rendering issues for ads in a new format), to the ads displayed in the original app. In the other half, AdMob ads are displayed but ads from other

ad libraries and servers are not; support for these could be added to Mr. Hide in the future.

**Phone state permissions.** The `READ_PHONE_STATE` permission allows an app to access various kinds of telephony data from the phone. We found that in 6 out of the 13 uses, the permission is used only to access either the phone-specific (IMEI) or SIM-card-specific (IMSI) ID number. This ID allows the app provider to track how clients use the app. Our fine-grained permission *UniqueID* can replace `READ_PHONE_STATE` in these 6 cases, providing access only to a randomly generated ID number (distinct from the IMEI and IMSI for greater security).

Of the remaining 7 apps, 3 of them use the permission `READ_PHONE_STATE` only to obtain notification about coarse phone status changes (e.g., to silence the music in FreeMusic when the phone rings). It would be possible to create a permission in Mr. Hide tailored to this use, but we have not done so yet.

**Wake-lock permissions.** The `WAKE_LOCK` permission enables an app to keep certain aspects of the phone “awake” while a wake lock is held. For example, an app can acquire a lock to keep the CPU awake during CPU-intensive tasks such as copying or downloading, or to keep the screen awake while a video is playing.

It would be possible to create several fine-grained permissions to replace various uses of `WAKE_LOCK`. For example, we could refine CPU wake locks so that when the phone has less than 25% battery remaining (and is running on the battery) the wake lock is dropped. However, it is unclear how widely applicable and useful such permissions would be, and thus we have chosen not to pursue them.

Surprisingly, we observed that the `WAKE_LOCK` permission is unnecessary in 3 of the 9 apps that acquire it. These apps use `android.media.MediaPlayer`, whose documentation says that the permission *may* be necessary. However, an examination of its source indicates that the permission is never used. Thus, Dr. Android can be used to simply remove `WAKE_LOCK` from these three apps.

**Location permissions.** The `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION` permissions provide GPS- and networked-based location information, respectively. In many circumstances, users may be uncomfortable providing their precise location to apps. Our *LocationBlock* permission can be used in place of the default permissions in all 8 apps to retain app functionality while revealing less information. The implementation of *LocationBlock* provides locations that are fuzzed via truncation to approximately the accuracy of a city block (150m). The reason for a simple truncation

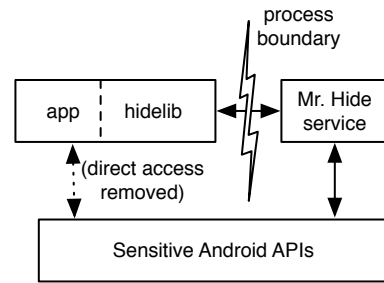


Figure 2: Mr. Hide architecture

rather than a Gaussian blur is that the latter could reveal the user’s location over time via multiple samples. It would also be easy to provide other options for users (e.g., accuracy up to the zip code or city level).

**Settings permissions.** Android’s `WRITE_SETTINGS` permission allows an app to read and write a variety of system settings. In 3 of the 5 apps that require the permission, we found that their only use was to change the phone’s ringtone settings. Our *SetRingtone* permission is dedicated to this particular privilege. In the other 2 apps we found that the permission was unnecessary and can be removed by Dr. Android.

**Read contacts permissions.** The `READ_CONTACTS` permission allows all contacts on the phone to be read. Our *ReadVisibleContacts* permission leverages Android’s notion of contact visibility, a boolean flag associated with each contact. With this restricted permission, apps may only see contacts marked as visible.

This permission is more a demonstration than a practical design—invisible contacts are typically used for special app information, such as synchronization accounts, that the app wants to see but users do not. However, it would be straightforward to generalize our approach to support Android’s notion of contact groups (e.g., friends, colleagues) or to enforce an arbitrary per-app security policy controlled by a master configuration program belonging to Mr. Hide. Currently, our implementation does not work for Barcode because that app uses a deprecated API for accessing contacts.

## 4. IMPLEMENTING FINED-GRAINED PERMISSIONS IN Mr. Hide

As discussed above, Mr. Hide provides controlled access to system resources by interposing a new API, implemented via Android services, between underlying resources and client apps. Figure 2 gives an overview of the Mr. Hide architecture, which contains two main components: the Mr. Hide service, which runs in a separate process, and *hidelib*, a drop-in replacement for sensitive Android APIs that manages all interprocess com-

munication with the Mr. Hide services. As illustrated in the figure and discussed more in Section 5, `hidelib` is actually appended to the original app’s apk.

Before presenting details of the Mr. Hide implementation, we first give an overview of some aspects of the Android platform.

## 4.1 Android platform overview

Android provides a component-oriented programming model with an associated security model [8]. An Android app consists of four main types of components: *Activities* that define the user interface screens of an application; *Services* that run in the background; *Broadcast receivers* that await asynchronous messages from other components, including components of other applications; and *Content providers* that store data and allow access to it via a relational database interface. Apps are typically written in Java and compiled to Dalvik, a type and memory safe bytecode format. Each app runs in its own Dalvik virtual machine and in its own Linux process as a distinct user. This provides system-level isolation among Android apps. Apps may also include native code, although in our experience this is uncommon (except for rendering in games). 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.

Mr. Hide uses Android’s permission framework to define its own set of permissions, which apps can 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 Mr. Hide receives a remote procedure call, it first calls a platform method, such as `checkCallingPermission()`, to check whether the caller has the right Mr. Hide permission, and throws an exception if not.

**Binding to the Mr. Hide service.** Before communicating with the Mr. Hide service via interprocess communication, applications first must bind to the service. Moreover, this binding process must be completed before running any code that communicates with Mr. Hide. We found that achieving synchronous service binding in Android is a bit tricky, because the code that binds the service must return control to the platform before the binding becomes available.

We experimented with several possible solutions, and found the following approach works reliably on our devices: We perform service binding in the app’s application-level `onCreate()` method, which is called as soon as the app is launched. A service binding established in this

way exists throughout the lifetime of the app, and will be reclaimed when the app exits. In all apps we looked at, this particular `onCreate()` method is either not provided (in which case we can supply our own to bind the service), or it exists but does not contain any code that would need to contact Mr. Hide (in which case we can chain it together with our own method to bind the service). Then, we demote the app’s “launcher” activity (the first to be executed when the app is started by clicking on its icon) to a regular activity, and insert a special Mr. Hide activity with a splash screen. The splash screen displays a message (“Protected by Dr. Android and Mr. Hide”) and then exits, passing control onto what was the main activity. This extra interposition of an activity causes the service binding begun in the app’s `onCreate()` to finish before the demoted launcher activity is reached, thus enabling the rest of the app to assume the service is bound.

Note that there are ways to start an app without invoking its launcher activity, e.g., intents can be used to invoke non-launcher activities from another app. However, these are less commonly used, and Mr. Hide could handle these cases by adding suitable logic to bind the service if needed before these other activities execute.

**Parameterized permissions.** Finally, 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

**InternetURL(x).** Android apps access the Internet using libraries that wrap low-level native code interfaces. This native code provides access to the Linux system call interface and the efficient cryptographic libraries used for SSL sockets. `hidelib` virtualizes these low-level components, implementing native calls declared in the classes `InternetAddress`, `OSNetworkSystem`, and `NativeCrypto`. Would-be native calls are forwarded to a Mr. Hide service. Structures that cannot not be marshaled for RPC, such as file handles and SSL contexts, are represented using unique proxy values, which the Mr. Hide services maps to, e.g., Linux file handles. Mr. Hide performs access control checks before establishing socket connections or performing DNS lookups.

Most apps do not directly access low-level code, such as `OSNetworkSystem`, and instead rely on high-level, built-in libraries for network access. These libraries include `java.net`, `org.apache`, `android.webkit`, and `javax.net`. Because these built-in libraries are dynamically linked, we cannot modify them using Dr. Android. Instead we build work-alike libraries by recompiling the original source to these libraries, making suitable source-level changes both automatically (e.g., renaming classes and methods) and manually (e.g., modifications to use our native code wrappers). These recompiled libraries are included in `hidelib`. `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 functions, which occasionally leads to observable differences in apps, as mentioned in Section 3. We also do not support `android.webkit`; later versions of the platform include hooks for controlling `webkit`'s use of sockets, which we believe would be a better implementation choice for Mr. Hide than recompilation.

*AdsPrivate and AdsGeo.* Mr. Hide's *AdsPrivate* service provides an interface allowing clients to request ads from the AdMob webservice. Standard apps use an unrestricted *INTERNET* permission to connect to the AdMob webservice, which may send private information, such as a cookie for ad targeting. In contrast, changing apps to use *AdsPrivate* allows apps to receive ads without being granted arbitrary network access.

Mr. Hide manages connections to the AdMob webservice in an isolated service process. When this process requests ads on behalf of clients, it forwards the developer's *advertiser id* (used to credit app developers for displaying ads), but no other information, to AdMob when requesting ads. After making a request, Mr. Hide receives an HTML formatted ad, downloads a referenced image, marshals the image as appropriate, and returns it to the client via a remote procedure call. Mr. Hide does not currently support newer ads containing javascript and multiple images, but these could be supported similarly.

The implementation of *AdsGeo* is similar, except Mr. Hide includes a location with the ad requests. This location is obtained by the Mr. Hide service itself, so apps may be given *AdsGeo* without having access to location. Currently this location is precise, but could be made coarse-grained to reduce the information provided to AdMob.

Two potential covert channels remain in this architecture: an advertiser could use two or more ids to send binary encoded strings, or could leak information in the timing of ad requests. We believe these low-bandwidth channels pose relatively minor privacy risks, and a stricter implementation of *AdsPrivate* and *AdsGeo* could mitigate these channels by memoizing the

advertiser ID and requesting ads on a suitable (fixed or randomized) schedule.

*LocationBlock and UniqueID.* Apps typically access location data via a special `LocationManager` object provided by the platform. The `LocationManager` allows the programmer to request asynchronous callbacks with current location information at programmer-selected time intervals. To implement *LocationBlock*, we provide a replacement for `LocationManager` in `hidelib` that passes on requests for asynchronous callbacks to the Mr. Hide service. That service polls the location at the specified intervals (using the system `LocationManager`) and then does a remote procedure call back to `hidelib`'s `LocationManager`, which then calls the callback specified by the app. GPS location coordinates returned by Mr. Hide are truncated to provide approximately 150m resolution.

In a few cases, apps also use `LocationManager`. `getLastKnownLocation()` to find location information. `hidelib` returns null in this case, which is allowed in the API and should be handled by the app.

Apps can also access the current location via a `TelephonyManager` object (which finds location based on cell network information). The same object is also used to return a unique identifier for the device. Thus, `hidelib` provides a replacement `TelephonyManager` object. For location information retrieved in this way, `hidelib` simply returns null values, indicating the app should get the location a different way—this case should be handled according to the API, and was in all the apps we studied.

To support *UniqueID*, our `TelephonyManager`'s `getDeviceId()` method 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.

*SetRingtone.* Device ringtones can be set in two ways on Android, using a `RingtoneManager` object or by calling `Settings.System.putString()`; the former is actually implemented on top of the latter. Thus, to implement *SetRingtone*, `hidelib` provides replacement `RingtoneManager` and `Settings.System` classes, each of which contact a Mr. Hide service to change a ringtone when requested.

The Mr. Hide service itself a thin wrapper that checks permissions and forwards `Settings.System.putString()` requests.

*ReadVisibleContacts.* Contacts are implemented on Android as a content provider, i.e., a database that can be queried by apps. Content providers are accessed by asking the platform to retrieve a particular URI that contains both the content provider and any additional

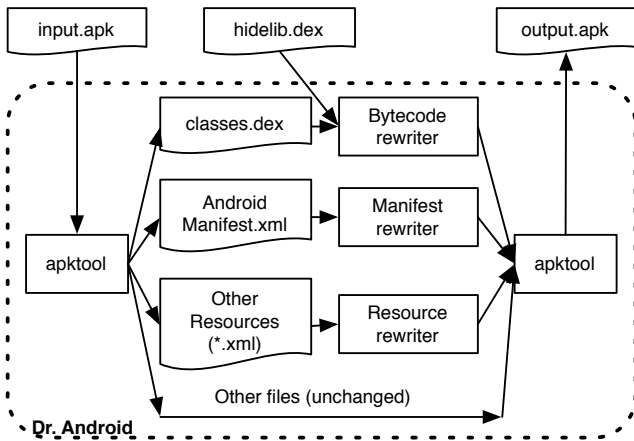


Figure 3: Dr. Android architecture

parameters to pass to it, e.g., an app can request `content://contacts` to get all the contacts on the device. For portability, apps typically derive such URIs from predefined strings, e.g., `ContactsContract.Contacts.CONTENT_URI` contains the URI just mentioned.

Mr. Hide implements a new content provider that uses the same patterns as device Contacts, but with a different name. Then `hidelib` provides replacement classes for `ContactsContract.Contacts` and similar that refer to Mr. Hide’s content provider. The Mr. Hide service captures the URIs sent to the content provider, inserts extra selection constraints to the query to filter out contacts that are invisible, and sends the modified query to the underlying Android content provider.

## 5. Dr. Android

As discussed earlier, we developed a tool called Dr. Android that performs binary transformation of Android apps to replace Android API calls with calls to Mr. Hide equivalents. Figure 3 shows the architecture of Dr. Android. Given an input apk file, Dr. Android uses `apktool` [12] to decompress the input file into its constituent files and directories. Dr. Android then performs three kinds of transformations. First, it modifies `classes.dex`, the file that contains the Dalvik bytecode for the app, to use Mr. Hide; in the process, Dr. Android also concatenates `hidelib.dex`, an adapter layer to connect to the Mr. Hide service running in a separate process, to the output `classes.dex` file. Second, Dr. Android modifies the list of permissions in `AndroidManifest.xml`, the application’s *manifest*, which contains the permissions requested at app installation time. In some cases, Dr. Android also must modify some additional XML files (details below). All of the modified files, as well as the remaining files in the app (e.g., images, data files, etc) are then repackaged using `apktool` to produce a transformed apk.

Note that we need to digitally sign `output.apk` to run

it on a phone. It is easy to generate a key to sign apps, but it will unavoidably differ from the key for the original app. Fortunately, app signatures are mostly used to establish trust relationships between different apps signed by the same key. We can preserve these relationships by consistently signing apps from the same original authors with the same new key.

Manifest and resource rewriting are quite straightforward. Existing permissions appear as `<permission>` elements in the manifest, and so those elements are modified as necessary to refer to the appropriate Mr. Hide permissions. We also modify the launcher `<Activity>` to be an ordinary, non-launcher activity, and insert Mr. Hide’s splash screen activity. Similarly, resource rewriting simply involves modifying some elements in XML files. Resource rewriting is only used to support *AdsPrivate* and *AdsGeo*, and is explained in more detail in Section 5.2. Thus, most of the challenge in Dr. Android is in transforming Dalvik bytecode files.

### 5.1 Transforming bytecode files

Dalvik bytecode files are structured as a series of indexed “pools” that contain, among others, strings, types, field signatures, method signatures, classes, field definitions, and method definitions.<sup>2</sup> The various pools are tightly intertwined, with many pointers from one to another, e.g., a method signature contains a list of pointers to the type pool, and the type pool contains pointers to elements in the string pool (containing the actual type names). Dalvik bytecode instructions (which appear in method definitions) often refer to elements in various pools, e.g., method invocation instructions include a pointer to a method signature. Before the Dalvik Virtual Machine executes a bytecode file, it first *verifies* it to check that, among other things, the file is well-formed and the method bodies are type safe.

The bytecode transformer in Dr. Android comprises approximately 10K lines of OCaml code that parses a bytecode file into an in-memory data structure, modifies that data structure, and then unparses it to produce an output bytecode file. We designed our in-memory representation to be as faithful as possible to the on-disk representation, to make the input and output process easy.

Figure 4 gives slightly simplified OCaml code for a function `rewrite` that applies Mr. Hide-specific modifications to its argument `dx`, the data structure (of type `dex`) representing a bytecode file. (Note that we chose to modify this data structure by side effects rather than using a purely functional implementation because it was slightly simpler.) The second argument, `ps`, is the list of Mr. Hide permissions for which the app should be

<sup>2</sup>Note that unlike in the Java Virtual Machine, in which each class is compiled to separate file, in Dalvik all the classes for an app appear in the same `.dex` file.

```

1 type perm = AdsPrivate | AdsGeo | ...
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   if (List.mem AdsPrivate ps || List.mem AdsGeo ps) then
9     elide_perm_checks dx ["android.Permission.INTERNET";
10      "android.Permission.ACCESS_FINE_LOCATION";
11      "android.Permission.ACCESS_COARSE_LOCATION"];
12   replace_managers dx ps perm_managers;
13   if List.mem ReadVisibleContacts ps then
14     replace_contact_strings dx;
15   insert_service_binding dx

```

Figure 4: Bytecode rewriter pseudocode

rewritten.

This function begins by calling `merge_hidelib` to append the contents of `hidelib.dex` to `dx`. This is not quite as easy as it sounds, because the Dalvik verifier requires that each of the pools in a bytecode file are both duplicate-free and sorted. For example, there must be at most one string `V` representing the type void in the type pool, but it is almost guaranteed this type appears in both the app’s code and `hidelib.dex`. In our implementation, we permit a dex data structure to contain duplicates and out-of-order elements, and we re-sort and eliminate duplicates in each pool before producing the output file.

Next, on line 7 we call `replace_classes` to change references to Android platform code to refer to `hidelib.dex` instead. As mentioned earlier, we designed the Mr. Hide adapter layer to provide drop-in replacements for platform APIs. Thus, the input to `replace_classes` is a map from platform API class names to the corresponding `hidelib.dex` class names, which is then used to modify all references to that type, e.g., in instructions, field or method signatures, debugging information, annotations, etc.. For example, to support *AdsPrivate* Dr. Android transforms accesses to `com.admob.android.ads.AdView` with accesses to a class `hidelib.ads.clientsideonly.AdView`, among many others. Here, the mappings are represented with associative lists, and `perm_map` is defined on line 2 as a function that, given a permission, returns its corresponding mapping. The mappings for every permission in `ps` are simply concatenated together on line 7.

Note that `replace_classes` does not replace references inside code from `hidelib.dex`. This lets us neatly handle the case when one class has both privileged and unprivileged methods; the privileged method wrapped in `hidelib.dex` contacts the Mr. Hide service, and the unprivileged methods call into the platform as usual, and

hence we do not rewrite those calls.

Then, on lines 8–14, we modify some code to support *AdsPrivate*, *AdsGeo*, *LocationBlock*, *SetRingtone*, and *ReadVisibleContacts*; we defer discussion of these modifications until we talk about those permissions in Section 5.2. Finally, on line 15 we insert code to bind the Mr. Hide service (running in a separate process) so it can be called from the app. As discussed earlier, we place this code in the application-level `onCreate()` method. More specifically, apps specify this `onCreate()` method by naming its containing class in the manifest. If such a class is defined, we modify it so its superclass is `hidelib.Application`, which will ensure Mr. Hide’s `onCreate()` is called. Otherwise, we simply extend the manifest to name `hidelib.Application` as the class containing the application’s `onCreate()`. Additionally, recall that after Mr. Hide’s `onCreate()` runs, it starts a splash screen. The call to `insert_service_binding` stores the name of the app’s original launcher activity in a designated string, and the splash screen code in Mr. Hide uses that name to reflectively launch that activity when it exits.

Next, we discuss in more detail the rewrites necessary for each of Mr. Hide’s new permissions.

## 5.2 Mr. Hide-specific rewrites

*InternetURL*. As mentioned earlier, `hidelib` provides an interface that replaces `java.net` and `org.apache`. Thus, the transformation that replaces *INTERNET* permission with various *InternetURL(d)* permissions modifies references to those packages to simply refer to Mr. Hide instead.

Dr. Android includes a tool that finds all URL-like strings appearing in an app’s string pool. Then when we add *InternetURL(d)* permissions to an app, we do so for all such strings. In our experience, this allows apps to run correctly, while at the same time greatly restricting their access to the Internet.

*AdsPrivate/AdsGeo*. As expected, the transformation for *AdsPrivate* and *AdsGeo* replaces calls to the AdMob library with appropriate Mr. Hide equivalents. Interestingly, there are two additional steps needed to make ads work.

First, we discovered that AdMob checks for the presence of the *INTERNET* permission, and disables display of ads if not present. Some apps that use these libraries also check for *ACCESS\_FINE\_LOCATION* and *ACCESS\_COARSE\_LOCATION*, to determine whether to show localized ads. Thus, Dr. Android disables these permission checks. In Figure 4, lines 8–11 call a function `elide_perm_checks` that takes a list of permission names and replaces checks for those with no-ops. Specifically, it finds code sequences with the following pat-

tern, where `@str_id` is the (index of the) target permission string, and `@mtd_id` refers to the method `Context.checkCallingOrSelfPermission`:

```
const-string v_x @str_id
invoke-virtual this, v_x, @mtd_id
move-result v_y
```

`elide_perm_checks` replaces the last instruction in the sequence with

```
move-const v_y 0
```

where constant 0 means `PERMISSION_GRANTED`.

Second, we found that apps often contain XML files that customize the screen layout for ads under a range of different screen resolutions. These files contain references to `com.google.ads.GoogleAdView` and other ad-related classes, and thus we modify those to refer to Mr. Hide’s ad classes. This is equivalent to class replacement in `.dex` files, except here we need to replace classes that appear textually in XML files.

One additional challenge we encountered in supporting ad permissions is that two apps (Shazam and Advanced Task Killer) had been obfuscated by changing class and method names. This is not a problem with transformations for platform APIs, since those names cannot be changed; but ad libraries are statically linked into apps, and so their classes and methods can be renamed. To solve this problem, we developed a simple tool that matches up method signatures between obfuscated and unobfuscated AdMob interfaces (the obfuscation did not change these signatures). Using the output of this tool, we can then direct Dr. Android to transform the correct set of ad API calls to use Mr. Hide instead. This approach allowed us to deobfuscate Shazam and Advanced Task Killer sufficiently to support ad permissions.

**LocationBlock and UniqueID.** As mentioned earlier, apps access locations via either a `LocationManager` or `TelephonyManager` object; the latter is also used to find the device ID. Apps request these objects by calling the platform function `getSystemService()`, which takes as its argument a string describing the type of manager needed (location, telephony, etc.). The `getSystemService()` method then returns a generic Java Object that must be downcast to a `LocationManager` or `TelephonyManager`, as appropriate. Dr. Android detects this cast and replaces it with a call to `hidelib.dex` to create a manager object of the appropriate type. Notice that this cast detection trick avoids the need for a separate static analysis to discover which calls to `getSystemService()` return which managers (or objects for other services unrelated to location or telephony). Since requests for locations and unique ids all go through the `LocationManager` and `TelephonyManager` objects, once we replace those we need not modify any other part of the class. In Fig-

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

Figure 5: Microbenchmark performance results

ure 4, this `LocationManager` substitution is performed by the calls to `replace_managers` on line 12. In addition to the dex data structure, this function takes the list of permissions and a function `perm_managers` that defines which managers need to be substituted for each permission. For example, `perm_manager LocationBlock` would return a list mapping Android’s `LocationManager` and `TelephonyManager` to `hidelib`’s equivalents.

**SetRingtone.** Apps change the phone’s ringtone using a `RingtoneManager`, and so as above, we replace such an object returned from `getSystemService()` with the corresponding object from Mr. Hide. As above, this occurs on line 12.

**ReadVisibleContacts.** As discussed earlier, contacts are implemented as a content provider, accessed via URIs that are usually constructed from constant strings defined in the platform API. `hidelib` contains replacements for those classes, so to change an app to use `ReadVisibleContacts`, we change references to the API classes containing contact URIs to the `hidelib` classes. We found that some apps also hard code the URIs instead of using platform classes, so to support these cases Dr. Android also searches for contact-related URI strings in the string pool and modifies them appropriately. Lines 13–14 in Figure 4 perform this rewriting.

## 6. EXPERIMENTS

We evaluated the combination of Mr. Hide and Dr. Android in three ways. First, we performed informal testing on Mr. Hide to ensure it implements all its permissions correctly. For example, we tested using Mr. Hide to access contacts and verified that only visible contacts were revealed, and similarly for the other permissions. As these particular results are not very interesting, we do not report on them further. Second, we used microbenchmarks to measure the overhead of using Mr. Hide and Dr. Android compared to using direct system calls. Third, we ran Dr. Android on the apps discussed in Section 3 and evaluated the correctness and usability of the transformed apps, using a purpose-built automated testing framework in conjunction with manual tests.

### 6.1 Microbenchmark performance

To measure the overhead of the interprocess communication entailed by Mr. Hide, we developed a set of mi-

crobenchmarks as Android activities that retrieve data from a sequence of 100 distinct web pages on the local network; request 10 location updates; change ringtone paths 1,000 times; and make 100 queries to the contact manager.

Figure 5 shows the running times of these microbenchmarks before and after applying Mr. Hide and Dr. Android, 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 Mr. Hide and Dr. Android on real apps

Using a combination of automated and manual testing, we evaluated if and how rewriting changes the behavior of the apps discussed in Section 3. We used Dr. Android’s reporting features and the Troyd framework (described below) to track test coverage, comparing the set of activities exercised by testing with the set declared in each application’s binary. Figure 6 summarizes this coverage information and gives pertinent data about the rewriting process for each app.

*Scope and 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 6 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 an instruction modification to refer to a different class; an elision of a permission check (counts as one change); replacement of a `Location`, `Telephony`, or `RingtoneManager` (again, one change); or changing a string in the string table for another reason (changing a content URI, or modifying the string that contains the activity to launch after the splash screen). Reported performance is based on one run on a 2.5 GHz Intel Core 2 Duo with 4 GB RAM, running Mac OS X 10.6.8. As the running times show, applying Dr. Android is reasonably fast, and as it is done only once, should not be a concern.

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 tested apps install and run without verification errors, giving confidence that

Dr. Android’s transformations are structurally correct.

*Automated and manual testing methodology.* To evaluate the behavior of rewritten apps, we developed an automated testing framework, Troyd, concurrently with this work. We describe Troyd only briefly, as it is not the focus of this paper. The key novelty of Troyd is that it can run on apps without source code, yet it allows test writers to refer to GUI elements by name or text content. Test cases—written as Ruby scripts—control apps by injecting key events like back or menu button clicks, and can click or edit attributes of all `View` elements on the screen at run time. Test cases can also get attributes of `View` elements (e.g., contents of text boxes), which can be used to write assertions. Troyd currently does not support clicking absolute screen locations or dragging, which limits its applicability to some apps such as games; we perform additional manual testing in these cases.

We wrote test cases for our apps with the goal of covering as many of an app’s `Activities` as possible. 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 developed tests in Troyd to verify that the expected buttons, menus, and other displays (e.g., ads) 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 ran our automated tests on a Google Nexus S phone.

Some activities are not covered by Troyd for various reasons, including dead code (i.e., unreachable activities), activities that are not easy to automatically test (e.g., sign up screens that would require creating fake logins to repeatedly test), activities that require clicking absolute screen locations or dragging, activities that are launched in a separate process (and hence cannot be controlled by Troyd), and activities that are tedious to reach with automated tests.

Additionally, we manually tested all rewritten apps, checking for obvious changes in performance or functionality. For each app we attempted to visit all accessible activities and use most features. Again, features with costly or annoying side effects, such as finalizing purchases using the Amazon app or posting games scores to Facebook using the Words with Friends app, were not tested. We used Troyd to log visited activities during manual testing.

The last three columns in Figure 6 report the coverage of our test suite in terms of activities, including the total number of activities defined in each app and the number covered with automated and with manual testing. Conservatively, our automatic and manual testing

Name	Apk (KB)	Dex (KB)	# Ins	# Chg	Tm (s)	# Acts	# Aut	# Man
Advanced Task Killer	98	110	6,724	115	7.51	3	3	3
Amazon	2,288	1,607	114,673	178	36.24	28	–	15
Angry Birds	17,111	254	24,212	354	22.75	2	–	2
Angry Birds Rio	12,835	203	18,539	274	21.56	2	–	2
ASTRO	2,241	1,176	112,342	135	20.01	29	13	17
Barcode (zxing)	496	454	64,045	315	15.65	9	7	8
Bubble Blast	1,413	714	66,189	1,013	17.27	11	6	8
Bubble Blast 2	4,489	879	82,233	1,063	24.34	16	5	12
Brightest Flashlight	1,752	1,857	173,172	1,289	27.85	6	–	1
Dropbox	1,137	1,040	94,367	4,111	13.35	12	8	6
ESPN ScoreCenter	1,483	670	57,541	608	26.48	5	–	4
Flashlight	1,665	464	44,592	581	18.80	6	–	1
FreeMusic	539	655	59,111	676	10.70	12	–	6
Gas Buddy	1,359	853	74,240	709	23.59	22	15	13
Google Sky Map	2,125	415	29,401	81	17.32	6	4	5
Horoscope	3,023	815	88,760	698	29.33	26	–	11
MP3 Ringtone	428	373	44,014	50	10.20	11	–	7
Shazam	1,130	756	89,310	661	21.60	19	10	10
Words With Friends	5,314	2,718	223,620	1,226	61.18	38	–	13

Figure 6: Mr. Hide and Dr. Android results on apps

together cover at least 56% of the activities across all apps.

*Behavior of rewritten apps.* During testing, we found that almost all activities of applications function normally, with no observable changes. In more detail, all automated Troyd tests pass—running our test suites before and after applying Dr. Android produced identical sets of passing assertions—and when we tested the apps manually, we could detect no differences in most activities’ behavior.

As already mentioned in Figure 1, we did find some differences in certain activities that use the Internet, due to limitations of Mr. Hide. First, ESPN ScoreCenter, Google Sky Map, and Shazam use a WebView widget, which we do not support; these views show placeholder text after rewriting. Second, recall that Mr. Hide’s implementation of ads renders a single image. We observed that this design choice causes ads containing Javascript or multiple images to display improperly. Finally, Barcode uses parts of the Java SSL framework we do not cover; even so, the Barcode app still works with Mr. Hide’s *InternetURL* permission, but users lose the ability to search a book’s text after scanning a barcode-encoded ISBN. In general, we think these differences may be acceptable to users in exchange for the increased security provided, and we expect that further engineering would eliminate them.

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.

## 7. RELATED WORK

Several other researchers have proposed mechanisms to refine or reduce permissions in Android. MockDroid allows users to replace an app’s view of certain private data with fake information [3]. Apex is similar, and also lets the user enforce simple constraints such as the number of times per day a resource may be accessed [16]. 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 [19]. 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 [15]. A limitation of all of these approaches is that they require modifications to the 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 available today.

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 [7]. 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 [17]. 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 Android’s permissions, sensitive APIs, and the use of permissions across apps. Barrera et al. [2] analyze the way permissions are used in Android apps, and observe that only a small

number of Android permissions are widely used but that some of these, in particular Internet permissions, are overly broad (as we have also found). Vidas et al. [18] describe a tool that, using documentation-derived information, can statically analyze an app’s source code to find a minimum set of permissions it needs. Stowaway [10] performs a static analysis on the Android API itself to discover which APIs require what permissions, something they found is not always well documented. (We used Stowaway’s data set in several cases to help determine what adapters we needed to implement in Mr. Hide.)

Finally, several tools have been developed that look for security issues in Android apps. TaintDroid tracks the flow of sensitive information [5]. Ded [6], a Dalvik-to-Java decompiler, has been used to discover previously undisclosed device identifier leaks. ComDroid [4] finds vulnerabilities related to Intent handling. Felt et al. [9] study the problem of permission redelegation, in which an app is tricked into providing sensitive capabilities to another app. Woodpecker [14] uses dataflow analysis to find capability leaks on Android phones. All of these tools focus on improper use of the current set of Android’s permissions. Dr. Android and Mr. Hide take a complimentary approach, replacing existing permissions with finer-grained ones to reduce or eliminate consequences of security issues.

## 8. CONCLUSIONS AND FUTURE WORK

We presented Mr. Hide and Dr. Android, a pair of tools that provide finer-grained permissions on Android without requiring any platform modifications. Mr. Hide runs as a service, providing access to sensitive capabilities along with a set of fine-grained permissions that grant access to the service. Dr. Android transforms apps to use Mr. Hide, operating directly on apks to change coarse-grain platform permissions into finer-grain Mr. Hide permissions, and modifying Dalvik bytecode to access sensitive resources via Mr. Hide’s adapter layer. We applied Mr. Hide and Dr. Android to a range of apps, and found that they leave app behavior largely unchanged, while maintaining acceptable performance. Our results suggest that Dr. Android and Mr. Hide provide stronger privacy and security guarantees while retaining application functionality and performance.

## 9. REFERENCES

- [1] 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>.
- [2] 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.
- [3] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: trading privacy for application functionality on smartphones. In *HotMobile*, 2011.
- [4] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *MobiSys*, 2011.
- [5] 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.
- [6] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security*, 2011.
- [7] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS*, pages 235–245, 2009.
- [8] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *Security Privacy, IEEE*, 7(1):50–57, jan. 2009.
- [9] A. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security*, 2011.
- [10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS*, 2011.
- [11] 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>.
- [12] Google. Tool for reengineering Android apk files. <http://code.google.com/p/android-apktool/>.
- [13] Google. 10 Billion Android Market Downloads and Counting, Dec. 2011. <http://android-developers.blogspot.com/2011/12/10-billion-android-market-downloads-and.html>.
- [14] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS*, 2012. To appear.
- [15] 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.
- [16] 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.

- [17] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *ACSAC*, pages 340–349, 2009.
- [18] T. Vidas, N. Christin, and L. F. Cranor. Curbing Android Permission Creep. In *W2SP*, 2011.
- [19] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on android). *Trust and Trustworthy Computing*, pages 93–107, 2011.