MockDroid: trading privacy for application functionality on smartphones

Alastair R. Beresford
Computer Laboratory,
University of Cambridge,
15 JJ Thomson Avenue,
Cambridge. CB3 0FD
United Kingdom
arb33@cam.ac.uk

Andrew Rice
Computer Laboratory,
University of Cambridge,
15 JJ Thomson Avenue,
Cambridge. CB3 0FD
United Kingdom
acr31@cam.ac.uk

Nicholas Skehin
Computer Laboratory,
University of Cambridge,
15 JJ Thomson Avenue,
Cambridge. CB3 0FD
United Kingdom
ns476@cam.ac.uk

Ripduman Sohan
Computer Laboratory,
University of Cambridge,
15 JJ Thomson Avenue,
Cambridge. CB3 0FD
United Kingdom
rss39@cam.ac.uk

ABSTRACT

MockDroid is a modified version of the Android operating system which allows a user to 'mock' an application's access to a resource. This resource is subsequently reported as empty or unavailable whenever the application requests access. This approach allows users to revoke access to particular resources at run-time, encouraging users to consider the trade-off between functionality and the disclosure of personal information whilst they use an application. Existing applications continue to work on MockDroid, possibly with reduced functionality, since existing applications are already written to tolerate resource failure, such as network unavailability or lack of a GPS signal. We demonstrate the practicality of our approach by successfully running a random sample of twenty-three popular applications from the Android Market.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access Controls; J.7 [**Computers in Other Systems**]: Consumer products; H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Mobile, Phone, Security, Privacy

Keywords

Android, MockDroid

HotMobile '11 Mar 01-03 2011, Phoenix, AZ, USA

1. INTRODUCTION

In the last couple of years, third-party applications for smartphones have become very popular for users and developers alike. For example, Apple has nearly 250,000 applications in its store [3] and has served over 3 billion downloads [1]. This is due to the confluence of several technical, business and economic factors, including the availability of good mobile data connectivity, high-performance low-power hardware, an online application store, and an integrated billing system which allows application developers to collect payment for applications easily. Many modern mobile phone vendors have applications stores, including the App Store (Apple), App World (Blackberry), Android Market (Google), Windows Marketplace for Mobile (Microsoft) and the Ovi Store (Nokia).

All the major platforms provide programming APIs which allow access to many of the core services of the smartphone, including: communication capabilities (e.g. IP connections, phone calling and text messaging), databases of personal information (e.g. address book and calendar data), and sensor information (e.g. microphones, cameras, phone location, and accelerometer data). Unfettered access to these data sources is a real privacy concern for users.

Smartphone operating system vendors are aware of the potential for poorly written or malicious programs to compromise privacy. Several different approaches have been developed to reduce the risk. Application vetting is a popular technique, used for a long time by Symbian, and now adopted by Nokia and Apple. In this model, the developer writes and tests an application on a specific device. Once the application is complete, the developer submits the application to a trusted party for testing; this might be the operating system vendor themselves or a third party. Typically, testing checks for adherence to published design guidelines in addition to checking for malicious behaviour. If the trusted party approves the application, it digitally signs the application; phone handsets will not install an application unless it is signed by a trusted key.

The vetting process is often opaque and suffers from both false accepts and false rejects. As the largest store, applications written for the iPhone provide several high-profile examples, although other application stores also suffer from similar problems. Once recent example is the latest version of the Facebook application for Apple's App Store. The Facebook application uploads all the contacts in the user's phone address book and then pattern matches these entries with the user's friends on Facebook. Newspaper reporters have discovered these numbers are not just stored on Facebook; they are also associated with the profiles of matching friends, and therefore a phone number which is only available to a small number of people is shared more widely [4]. In the first release of the application users were not made aware of this new feature and it cannot be disabled easily in the phone application.

A popular alternative approach to vetting is to use mandatory access control. This technique is used by Android and also the J2ME platform, which are both available on many different mobile device handsets. In this model, the developer declares which APIs the application will use and the system runtime prevents the application from accessing any APIs it does not explicitly list. The set of permissions requested by the application can be reviewed by the user. On the Android platform, the permissions are displayed when the application is installed, and the user has to make a stark choice: grant the permissions and install the application, or don't use the application at all. The J2ME model is slightly more flexible: some platforms allow the user to select options such as "always", or "ask every time" for some APIs, although applications may fail to run correctly if access is denied. Some J2ME platforms only allow access to certain APIs if the application has been vetted and signed, building a hybrid combination of the vetting and mandatory access control models.

The problem with the the mandatory access control model is the user must give permission to access communication features or data without necessarily being able to understand when and why this access is required, or what happens with their data. On Android this is especially difficult since there is no "ask every time" option; after installation, applications can access any requested API without further user intervention.

In this paper we propose an alternative approach: allow the user to provide fake or 'mock' data to applications interactively as the application is being used. In other words, the user may provide different information on the status of communication capabilities, personal databases and sensors to each application and over time. For example, consider an application which requests IP connectivity, access to location data from the GPS sensor, and read-write access to the calendar database. On MockDroid, the user may choose to provide 'mock' GPS data, such as always reporting a specific latitude and longitude, or reporting "no location fix available" regardless of the actual status of the GPS device. Similarly, IP connectivity may always "time out" for specific network addresses, or even all requests for communication. Access to the calendar may only return a subset of events, or perhaps an initially empty database which is actually separate from the main phone calendar database.

Providing fake data may impair a piece of application functionality. For example, providing "no location fix available" to the Google Maps application, regardless of the actual state of the GPS device, prevents the application from providing search results based on accurate location data, or showing the current location of the user on the map; other features of the application, such as directions, continue to function as normal. Therefore, providing fake data allows the user to explore the trade-off between functionality and privacy for any given application. Lying in this way offers a number of advantages over simple mandatory access control:

Controlling optional features It is common for applications to request access to a specific API to provide a feature which is optional; for example, the Skype application on Android allows the user to choose whether to integrate Skype contacts with the main phone address book. This kind of feature selection can be enforced by the operating system rather than by the application.

No unwarranted sharing of personal data Many applications which access personal data upload it to remote servers without the user realising. For example, a recent analysis of thirty Android applications which requested Internet connectivity along with location data or device ID found that twenty shared location data with content servers or third-party advertising networks without seeking user consent [6]. Allowing access to be mocked puts the user back in control.

Data separation For some data sources, it might be useful to provide a subset of the data, such as providing only public calendar events to a specific application. Similarly, it may be useful to control the quality of the data provided; for example, rather than providing the precise location of the phone to an application which shares such data with friends, the user might provide the location of the nearest city. Users could even maintain a separate database of personal information for each application if they wish.

Controlling expensive operations Access to some APIs cost money, and therefore lying helps control costs. For example, if the 3G data connection costs money, lying allows a user to prevent a data-hungry application from using the network whilst other applications continue to work as normal.

Enabling new features Many applications make use of sensor readings or information from personal databases to control the user experience. By allowing users to mock certain information, new features are automatically enabled. For example, accelerometers are often used to control the orientation of the screen. By providing fake accelerometer data to a specific application the user can manually rotate the application screen independently of the actual phone orientation.

Testing Developers can use mock resources to test their applications to make sure they work correctly with the full range of possible inputs.

We explored this idea by modifying the Android operating system. In the remainder of this paper we describe why Android is a good starting point (Section 2) and the changes required (Section 3). We demonstrate the effectiveness of our approach by examining a random sample of twenty-three applications on the Android Market (Section 4), compare our approach to prior work (Section 5) and discuss the merits of our approach and potential areas for future work (Section 6).

2. ANDROID

We chose Android since it is open source, already has a fine-grained permission system, and can be installed and tested on a real mobile phone handset.

Android applications are called *packages* and execute in a custom Java virtual machine running on a Linux kernel. Typically applications interact through the Android API inside the virtual machine. However it is possible to integrate native libraries written in C or C++ into a package and consequently it is normal for each package to run with its own kernel user identity to ensure strong privilege separation. Applications are built from multiple *components* that provide specific functionality.

Components are classified by type: *activity components* define a user interface; *service components* perform background processing; *content providers* provide data storage and retrieval facilities; and *broadcast receivers* receive messages from other applications. Components interact by sending messages called *intents*. Intents may be sent explicitly to named components or implicitly using a named action string. Android will redirect implicit intents to appropriate components automatically. Components use *intent filters* to subscribe to specific action strings. Inter-Component Communication (ICC) is restricted by *access permission labels*. If a component exposes an API with a specific label, any application which wants to use this API must declare this as a required permission in the package manifest file.

Android has four types of permission label: *normal* permissions (e.g. triggering the vibrate feature) are granted to any package; *dangerous* permissions alert the user to potentially insecure or fiscally expensive operations (e.g. making phone calls); *signature* permissions are only granted to packages that are signed with the same key as the package defining the permissions; and *signature or system* permissions mimic signature permissions, but exist for legacy reasons.

3. MOCKDROID

Our prototype is based on Android 2.2.1 and runs on the Google/HTC Nexus One handset. Whilst we have not tested it on other handsets, we expect that our modifications will run with minor adjustments on other Android devices which support the 2.2.1 version of the operating system. The patches, compiled firmware, and our permission control application are available on the project website.[1]

In MockDroid, the choice of lie depends on the API call under consideration. We have explored a wide range of API calls in order to demonstrate the feasibility of our approach. In particular, we provide support to mock the following:

Coarse- and fine-grained location if read requests are mocked, then the application never receives a callback with a location fix, simulating lack of available location information.

Internet if Internet connectivity is mocked then the socket never connects and always times out, simulating the lack of an available wireless network.

SMS/MMS, calendar, contacts if read requests or write requests are mocked, then they access a database which is in the same state as a brand new device (i.e. "empty"), and write requests fail by notifying the application that zero lines of the database were updated, simulating a full file system.

Device ID if read requests are mocked, then a fake constant value is returned.

Broadcast intents if the permission required to send a broadcast intent from a package is mocked, the broadcast intent is never sent; likewise, if the permission required to receive a broadcast intent by a package is mocked, it is never received. We have used this method to prevent an application from receiving notification of incoming SMS/MMS messages.

3.1 Package Manager modifications

The Package Manager service is the focal point for permissions management in Android. When a new application is installed from the Market, the set of dangerous permissions the application has requested are displayed and the user is asked to provide consent prior to installation. If the user agrees, the set of granted permissions are stored in an in-memory data structure and also serialised to disk.

Every Android API call which requires a permission (e.g. read access to the address book) first checks whether this permission has been granted to the calling package. This is done by using an internal ICC mechanism which ultimately checks the in-memory data structure maintained by the Package Manager. If an application attempts to use an API call without permission, an exception is thrown which propagates from the API call into the package runtime.

We modified the permission check performed at the start of each dangerous API call. Our modified implementation first checks whether the permission was requested by the package at installation time; if not, then an exception is thrown in the same manner as standard Android system. If the permission was requested at installation time, our modified implementation also checks whether the user has mocked the permission. We maintain separate state for each application so the user can prevent access to a resource by one application, but enable it for another.

If a permission was requested at install time and is not mocked, then the API call completes in the same manner as the standard Android system. If a permission is mocked then the API call provides a plausible but incorrect result to the application, as described earlier.

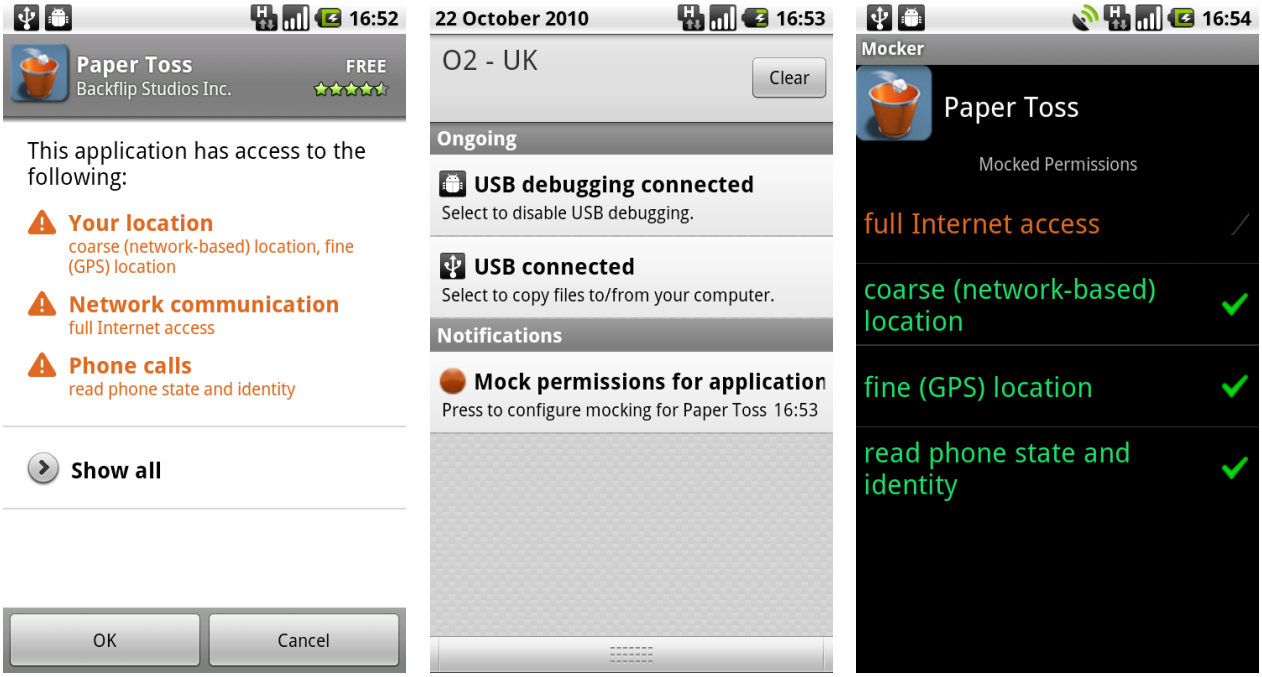
3.2 Storing mocked permissions

We have modified the Package Manager and duplicated the set of permissions so that each permission has both a 'real' and a 'mocked' version. When the application is installed, all requested permissions are granted and none are mocked.

To support control of mocked permissions at runtime, we have added an additional Unix group, called `mock`, to the Android OS. This group has read and write permissions to a directory on disk which stores the mocked permissions for all packages. Our modified version of the Package Manager service uses the Linux kernel `inotify` service to watch for changes to files in this directory, and updates its in-memory cache of mocked permissions as and when the file contents change.

We have also added an additional system permission to the Android OS. Any application which is granted this permission is placed in the `mock` group and can therefore read and

[1] `http://www.cl.cam.ac.uk/research/dtg/android/mock/`



(a) (b) (c)

Figure 1: Paper Toss: (a) installation asks for permissions which seem unnecessary; (b) if a mocked permission is used, the user is informed through an unobtrusive notification; (c) mocked permissions can be edited.

write the mocked permissions which are stored on the file system. We have written an additional application, called Mocker, which has this system permission enabled. This application allows the user to configure and change mocked permissions whilst programs are running.

We have also modified our prototype so that every time a mocked permission is checked, a broadcast intent is fired. This is received by the Mocker application and is currently used to display a notification to the user in the notification bar. It would be relatively easy to extend this to provide an archive of all requests, broken down by application and time, to allow more careful monitoring of API use by specific applications.

3.3 Internet Permissions

Internet Permissions are handled differently from other dangerous API calls in Android. This is because IP connectivity is exposed both through API calls via the Java virtual machine and through the underlying Linux kernel, the latter being available to an application library written in C or C++. The Android developers therefore modified the Linux kernel to check whether a process attempting to access the Internet is in the `inet` group. When the Android Activity Manager starts an application, it adds the application process to the `inet` group if the Internet Permission was requested at install time.

Mirroring this approach, we added an additional Linux group called `mock_inet`. When the Activity Manager starts an application it adds the application to the `mock_inet` group if the Internet Permission for the application is currently mocked; if the user attempts to change the status of this permission (say, from 'mocked' to 'real') then our modified implementation of the Package Manager restarts the application with the new group settings.

4. EVALUATION

We begin our evaluation with an example usage of MockDroid. Alice decides to install Paper Toss, a popular game on the Android Market. On installation, the application asks for permission to read the coarse- and fine-grained location data from her handset, connect to the Internet, and read the phone state and identity as shown in Figure 1(a). Alice does not understand why the application requires any of these permissions, so after installing the application, she uses the Mocker application to disable access to all these features and then runs the application.

Alice finds the game runs acceptably with all permissions mocked. After some time, Alice wishes to submit her high score to the on-line high score table. Unfortunately this does not work, as the Internet permission is currently mocked. The Mocker application informs Alice that the application wishes to use the Internet permission by placing a notification in the Android notification bar as shown in Figure 1(b). Alice can click on this notification and enable the Internet permission as shown in Figure 1(c). After her score has been submitted, Alice can then return to the Mocker application and disable access to the Internet if desired.

Displaying a red circle in the notification bar whenever a permission is mocked is relatively unobtrusive, and provides good visual feedback to the user when an action is likely to have affected the current foreground application's behaviour. Nevertheless, this user interface has its drawbacks, a topic we revisit in Section 6.

The authors of TaintDroid analysed thirty free popular applications on the Android Market which required Internet

Internet	Local
3001 Wisdom Quotes	ABC – Animals
BBC News Live Stream	Antivirus
Coupons	Astrid
Dastelefonbuch	Blackjack
Horoscope	Bump
Layar	Cestos
Manga Browser	Evernote
Movies	Probasketball
Ringtones	Solitaire
The Weather Channel	Traffic Jam
Yellow Pages	Trapster
Wertago	

Table 1: Applications tested under MockDroid

communication along with permission to access either location, camera or audio data [6]. They found that twenty of these applications violated the users privacy, primarily by sending the device ID or location to content or advertisement servers.

For the purposes of validating our MockDroid implementation, we repeated the experimental procedure carried out in the TaintDroid paper by downloading all the applications in the TaintDroid test set that required access to location data. Of the original twenty-seven applications that required location data we were unable to download three (Hearts, Spongebob slide and Babble) due to Android Market restrictions, and one (Knocking) did not function correctly because the application's remote server was unavailable at the time of testing.

Table 1 categorises the twenty-three applications as *Internet* or *local* depending on whether they required access to remote data to provide core functionality. For each application, we manually exercised application functionality with and without mocking coarse-grained location and Internet access. We were able to successfully mock both location and Internet access for all applications. All applications continue to function without failure in the presence of faked location and Internet state.

Applications in the local category access coarse-grained location data infrequently. The primary use appears to be the display of locationally-relevant adverts. Unless the user wants to see local adverts, mocking access to location data did not reduce application functionality, it simply increased privacy. Similarly, disabling access to the Internet prevented adverts from loading; in most other respects applications continued to function normally. Access to on-line high-score tables was one area in which access to the Internet is required and this represents the primary trade-off between application functionality and privacy for local applications.

Applications in the Internet category still run with network access mocked, but only provide limited functionality. In order to use these applications properly, network access is required. Many applications which request location data, such as The Weather Channel and Movies, provided mechanisms for manually entering a location in addition to acquiring data from sensors. Therefore, when mocking access to location data the trade-off is that the user must manually enter a location if they want the application to operate in a location-dependent fashion; the frequency at which this update must occur depends on the application, but in the case of The Weather Channel and Movies, this is likely to be relatively infrequent, and therefore is likely to be an acceptable trade-off to many users.

In addition to the above tests, we also mocked access to the contact database for the default SMS messaging application which comes with Android. With the contact database mocked, the message application continues to work, except that existing stored messages no longer display the sender name (messages are indexed by phone number instead) and all new messages require a phone number, as the application is unable to use the contact database to translate nicknames into phone numbers.

5. RELATED WORK

The amount of data available to applications running on mobile phones contrasts starkly with another mainstream application platform: the world-wide web. Unlike the smartphone runtime environment, a web application runtime provides very limited access to the underlying operating system and services: a typical web application today cannot access databases of personal information or sensor data, and the web browser limits IP connectivity with the same-origin policy. This situation is likely to change over time—W3C have a draft Geolocation API [7] and some smartphone web browsers, such as the Blackberry Browser [2], already provide access to the GPS location of the device via Javascript.

Conversely, a typical desktop computing environment offers the user with almost no protection from malicious applications since all applications and data are accessible by any application. Some virtual machines such as the Java Runtime Environment (JRE) support the concept of a Security Manager which is able to prevent access to specific API calls. These access controls are at the level of the programming API, and are often difficult to translate into meaningful concepts that the user could understand. For example, the JRE equivalent of the permission to read from the address book might be translated into permission to read the file `/home/user/contacts.vcf`.

The security-by-contract paradigm requires developers to specify the set of API calls their application requires [5]. This contract is then checked against a user policy which defines what applications are allowed to access. In essence, this adds mandatory access control to the .NET framework for Windows Mobile, but it does not permit the user to provide 'mock' or fake data.

TaintDroid is a real-time information flow analysis tool for Android [6]. It allows specific data processed by the Java virtual machine, such as GPS location, to be tracked as it is passed between processes via ICC. Analysis using this system, together with hand inspection, found that twenty out of the thirty applications under test commited some form of privacy violation. The system does suffer from false negatives and false positives and is currently only able to track information flow, not prevent it.

6. DISCUSSION AND FUTURE WORK

The Android security framework has been criticised for not allowing users to make fine-grained choices about which permissions to grant to applications. To date, the only alternative example is J2ME which does allow limited runtime control of permissions by the user, but requires the programmer to add additional code to explicitly cope with user de-

cisions. Our approach has identified an alternative. Instead of writing applications to handle partial access to resources, the programmer instead writes their application to handle resource failure; a task which must be accomplished anyway when writing a robust mobile application.

We also believe that combining the concepts of unavailability through lack of privileges and resource unavailability at the API level makes the trade-off between privacy and functionality more visible. Notifications provided to the user when data is faked facilitate a negotiation between the user of the device and the application. We have not explored in any great detail how to best support this negotiation. The use of the notification bar appears to provide good feedback in the common case, but in other situations it does not. Poor examples include: (a) games which use the full screen, thereby hiding the notification bar; and (b) mocking a resource used by a background service, because such a service may have no visible user interface at the time of notification. Further work is needed here.

Our prototype has only considered a few of the many permissions which an Android application might request. However, a similar approach could be applied in other areas: a blank image fed from the camera, silence from the microphone, and so on. Our examples so far have considered all-or-nothing decisions, and more fine-grained control might also be useful for some applications. For example, the ability to quota or filter an application's Internet access is a natural extension. Similarly, data from the GPS device might be processed to reduce its accuracy, such as returning the centre of the nearest city as its current location.

Instead of simply limiting an application's access to resources one could also enforce particular security practices on to applications. For example, data could be transparently encrypted and decrypted to disk or before being sent over the network. This would work in a straightforward manner for application databases such as the SMS database. By exploiting existing tools such as Fuse, one could provide a file system layer which enforces these restrictions for external storage, such as the SD card found in Android devices. Another example is to preserve communication privacy by routing an application's Internet traffic over the Tor network.

There are obviously a huge range of policies which could be applied to applications. In some settings (perhaps corporate environments) this might be proscribed by a central system administrator to enforce certain restrictions on applications. Users without the inclination (or expertise) to make policy decisions might instead share a set of policies in the manner popularised by tools such as Ad-blocker.

Free-to-download applications on application stores will commonly display adverts to the user as a means for generating revenue. However, we have shown that mocking Internet and location permissions effectively disables this functionality. Users may argue this is a feature, however the application publisher will certainly want to persuade the user to grant permission so their revenue model continues to function. This might work by offering an incentive to the user by providing some additional benefit through the availability of the resource. It is much harder for the application developer to force the user to enable access to the Internet, since the lack of network connectivity or current location data is a valid system state, however more advanced detection heuristics may become commonplace if systems like MockDroid become popular.

7. ACKNOWLEDGEMENTS

Thank you to Simon Hay and Alastair Tse for their insightful comments on MockDroid. Many thanks to RedGate for providing the financial support for Nicholas Skehin, and Google for donating Nexus One handsets.

8. REFERENCES

[1] `http://bits.blogs.nytimes.com/2010/01/05/apples-app-store-tops-3-billion-downloads/`. *Retrieved 17th October 2010.*

[2] `http://docs.blackberry.com/en/developers/deliverables/11944/CS_Using_the_Location_API_using_JavaScript_898722_11.jsp`. *Retrieved 22th October 2010.*

[3] `http://www.apple.com/ipad/features/app-store.html`. *Retrieved 17th October 2010.*

[4] `http://www.guardian.co.uk/technology/blog/2010/oct/06/facebook-privacy-phone-numbers-upload`. *Retrieved 20th October 2010.*

[5] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .net platform. *Information Security Technical Report*, 13(1):25–32, 2008.

[6] W. Enck, P. Gilbert, B. gon 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 *Proceedings of the 9th Usenix Symposium on Operating Systems Design and Implementation*, pages 393–408, August 2010.

[7] A. Popescu. Geolocation API Specification. Technical report, W3C, February 2010. `http://dev.w3.org/geo/api/spec-source.html` *Retrieved 22nd October 2010.*