Android Permissions Demystified

Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, David Wagner
University of California, Berkeley
{ apf, emc, sch, dawnsong, daw }@ cs.berkeley.edu

ABSTRACT

Android provides third-party applications with an extensive API that includes access to phone hardware, settings, and user data. Access to privacy- and security-relevant parts of the API is controlled with an install-time application permission system. We study Android applications to determine whether Android developers follow least privilege with their permission requests. We built Stowaway, a tool that detects overprivilege in compiled Android applications. Stowaway determines the set of API calls that an application uses and then maps those API calls to permissions. We used automated testing tools on the Android API in order to build the permission map that is necessary for detecting overprivilege. We apply Stowaway to a set of 940 applications and find that about one-third are overprivileged. We investigate the causes of overprivilege and find evidence that developers are trying to follow least privilege but sometimes fail due to insufficient API documentation.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Android, permissions, least privilege

1. INTRODUCTION

Android's unrestricted application market and open source have made it a popular platform for third-party applications. As of 2011, the Android Market includes more applications than the Apple App Store [10]. Android supports third-party development with an extensive API that provides applications with access to phone hardware (e.g., the camera), WiFi and cellular networks, user data, and phone settings.

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

Access to privacy- and security-relevant parts of Android's rich API is controlled by an install-time application permission system. Each application must declare upfront what permissions it requires, and the user is notified during installation about what permissions it will receive. If a user does not want to grant a permission to an application, he or she can cancel the installation process.

Install-time permissions can provide users with control over their privacy and reduce the impact of bugs and vulnerabilities in applications. However, an install-time permission system is ineffective if developers routinely request more permissions than they require. Overprivileged applications expose users to unnecessary permission warnings and increase the impact of a bug or vulnerability. We study Android applications to determine whether Android developers follow least privilege or overprivilege their applications.

We present a tool, Stowaway, that detects overprivilege in compiled Android applications. Stowaway is composed of two parts: a static analysis tool that determines what API calls an application makes, and a permission map that identifies what permissions are needed for each API call. Android's documentation does not provide sufficient permission information for such an analysis, so we empirically determined Android 2.2's access control policy. Using automated testing techniques, we achieved 85% coverage of the Android API. Our permission map provides insight into the Android permission system and enables us to identify overprivilege.

We apply Stowaway to 940 Android applications from the Android Market and find that about one-third of applications are overprivileged. The overprivileged applications generally request few extra privileges: more than half only contain one extra permission, and only 6% request more than four unnecessary permissions. We investigate causes of overprivilege and find that many developer errors stem from confusion about the permission system. Our results indicate that developers are trying to follow least privilege, which supports the potential effectiveness of install-time permission systems like Android's.

Android provides developer documentation, but its permission information is limited. The lack of reliable permission information may cause developer error. The documentation lists permission requirements for only 78 methods, whereas our testing reveals permission requirements for 1,259 methods (a sixteen-fold improvement over the documentation). Additionally, we identify 6 errors in the Android permission documentation. This imprecision leaves developers to supplement reference material with guesses and message boards. Developer confusion can lead to overprivileged applications, as the developer adds unnecessary permissions in an attempt to make the application work correctly.

Contributions. We provide the following contributions:

1. We developed Stowaway, a tool for detecting overprivilege in Android applications. We evaluate 940 applications from the Android Market with Stowaway and find that about one-third are overprivileged.
2. We identify and quantify patterns of developer error that lead to overprivilege.
3. Using automated testing techniques, we determine Android's access control policy. Our results represent a fifteen-fold improvement over the documentation.

Other existing tools [11, 12] and future program analyses could make use of our permission map to study permission usage in Android applications. Stowaway and the permission map data are available at `android-permissions.org`.

Organization. Section 2 provides an overview of Android and its permission system, Section 3 discusses our API testing methodology, and Section 4 describes our analysis of the Android API. Section 5 describes our static analysis tools for detecting overprivilege, and Section 6 discusses our application overprivilege analysis.

2. THE ANDROID PERMISSION SYSTEM

Android has an extensive API and permission system. We first provide a high-level overview of the Android application platform and permissions. We then present a detailed description of how Android permissions are enforced.

2.1 Android Background

Android smartphone users can install third-party applications through the Android Market [3] or Amazon Appstore [1]. The quality and trustworthiness of these third-party applications vary widely, so Android treats all applications as potentially buggy or malicious. Each application runs in a process with a low-privilege user ID, and applications can access only their own files by default. Applications are written in Java (possibly accompanied by native code), and each application runs in its own virtual machine.

Android controls access to system resources with install-time permissions. Android 2.2 defines 134 permissions, categorized into three threat levels:

1. *Normal* permissions protect access to API calls that could annoy but not harm the user. For example, `SET_WALLPAPER` controls the ability to change the user's background wallpaper.
2. *Dangerous* permissions control access to potentially harmful API calls, like those related to spending money or gathering private information. For example, Dangerous permissions are required to send text messages or read the list of contacts.
3. *Signature/System* permissions regulate access to the most dangerous privileges, such as the ability to control the backup process or delete application packages. These permissions are difficult to obtain: Signature permissions are granted only to applications that are signed with the device manufacturer's certificate, and SignatureOrSystem permissions are granted to applications that are signed or installed in a special system folder. These restrictions essentially limit Signature/System permissions to pre-installed applications, and requests for Signature/System permissions by other applications will be ignored.

Applications can define their own permissions for the purpose of self-protection, but we focus on Android-defined permissions that protect system resources. We do not consider developer-defined permissions at any stage of our analysis. Similarly, we do not consider Google-defined permissions that are included in Google applications like Google Reader but are not part of the operating system.

Permissions may be required when interacting with the system API, databases, and the message-passing system. The public API [2] describes 8,648 methods, some of which are protected by permissions. User data is stored in *Content Providers*, and permissions are required for operations on some system Content Providers. For example, applications must hold the `READ_CONTACTS` permission in order to execute READ queries on the Contacts Content Provider. Applications may also need permissions to receive *Intents* (i.e., messages) from the operating system. Intents notify applications of events, such as a change in network connectivity, and some Intents sent by the system are delivered only to applications with appropriate permissions. Furthermore, permissions are required to send Intents that mimic the contents of system Intents.

2.2 Permission Enforcement

We describe how the system API, Content Providers, and Intents are implemented and protected. To our knowledge, we are the first to describe the Android permission enforcement mechanisms in detail.

2.2.1 The API

API Structure. The Android API framework is composed of two parts: a library that resides in each application's virtual machine and an implementation of the API that runs in the system process(es). The API library runs with the same permissions as the application it accompanies, whereas the API implementation in the system process has no restrictions. The library provides syntactic sugar for interacting with the API implementation. API calls that read or change global phone state are proxied by the library to the API implementation in the system process.

API calls are handled in three steps (Figure 1). First, the application invokes the public API in the library. Second, the library invokes a private interface, also in the library. The private interface is an RPC stub. Third, the RPC stub initiates an RPC request with the system process that asks a system service to perform the desired operation. For example, if an application calls `ClipboardManager.getText()`, the call will be relayed to `IClipboard$Stub$Proxy`, which proxies the call to the system process's `ClipboardService`.

An application can use Java reflection [19] to access all of the API library's hidden and private classes, methods, and fields. Some private interfaces do not have any corresponding public API; however, applications can still invoke them using reflection. These non-public library methods are intended for use by Google applications or the framework itself, and developers are advised against using them because they may change or disappear between releases [17]. Nonetheless, some applications use them. Java code running in the system process is in a separate virtual machine and therefore immune to reflection.

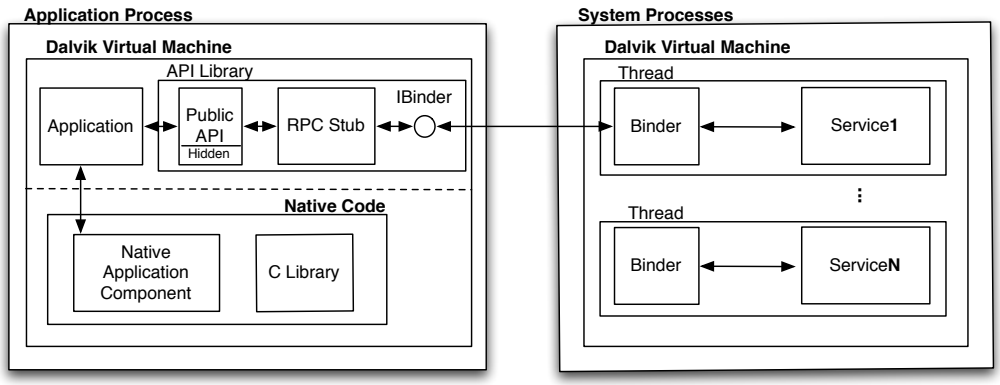

Figure 1: The architecture of the Android platform. Permission checks occur in the system process.

Permissions. To enforce permissions, various parts of the system invoke a permission validation mechanism to check whether a given application has a specified permission. The permission validation mechanism is implemented as part of the trusted system process, and invocations of the permission validation mechanism are spread throughout the API. There is no centralized policy for checking permissions when an API is called. Rather, mediation is contingent on the correct placement of permission validation calls.

Permission checks are placed in the API implementation in the system process. When necessary, the API implementation calls the permission validation mechanism to check that the invoking application has the necessary permissions. In some cases, the API library may also redundantly check these permissions, but such checks cannot be relied upon: applications can circumvent them by directly communicating with the system process via the RPC stubs. Permission checks therefore should not occur in the API library. Instead, the API implementation in the system process should invoke the permission validation mechanism.

A small number of permissions are enforced by Unix groups, rather than the Android permission validation mechanism. In particular, when an application is installed with the `INTERNET`, `WRITE_EXTERNAL_STORAGE`, or `BLUETOOTH` permissions, it is assigned to a Linux group that has access to the pertinent sockets and files. Thus, the Linux kernel enforces the access control policy for these permissions. The API library (which runs with the same rights as the application) can accordingly operate directly on these sockets and files, without needing to invoke the API implementation in the system process.

Native Code. Applications can include native code in addition to Java code, but native code is still beholden to the permission system. Attempts to open sockets or files are mediated by Linux permissions. Native code cannot communicate directly with the system API. Instead, the application must create Java wrapper methods to invoke the API on behalf of the native code. Android permissions are enforced as usual when the API calls are executed.

2.2.2 Content Providers

System Content Providers are installed as standalone applications, separate from the system process and API library. They are protected with both static and dynamic permission checks, using the same mechanisms that are available to applications to protect their own Content Providers.

Static declarations assign separate read and write permissions to a given Content Provider. By default, these permissions are applied to all resources stored by the Content Provider. Restrictions can also be applied at a finer granularity by associating permissions with a path (e.g., `content://a/b`). For example, a Content Provider that stores both public and private notes might want to set a default permission requirement for the whole Content Provider, but then allow unrestricted access to the public notes. Extra permission requirements can similarly be set for certain paths, making data under those paths accessible only if the calling application has the default permissions for the provider as well as the path-specific permissions.

Content Providers can also enforce permissions programmatically: the Content Provider code that handles a query can explicitly call the system's permission validation mechanism to require certain permissions. This gives the developer greater control over the granularity of the permission enforcement mechanism, allowing her to selectively require permissions for query values or database data.

2.2.3 Intents

Android's Intent system is used extensively for inter- and intra-application communication. To prevent applications from mimicking system Intents, Android restricts who may send certain Intents. All Intents are sent through the ActivityManagerService (a system service), which enforces this restriction. Two techniques are used to restrict the sending of system Intent. Some Intents can only be sent by applications with appropriate permissions. Other system Intents can only be sent by processes whose UID matches the system's. Intents in the latter category cannot be sent by applications, regardless of what permissions they hold, because these Intents must originate from the system process.

Applications may also need permissions to receive some system Intents. The OS uses the standard Android mechanism for restricting its Intent recipients. An application (in this case, the OS) may restrict who can receive an Intent by attaching a permission requirement to the Intent [13].

3. PERMISSION TESTING METHODOLOGY

Android's access control policy is not well documented, but the policy is necessary to determine whether applications are overprivileged. To address this shortcoming, we empirically determined the access control policy that Android enforces. We used testing to construct a permission map that identifies the permissions required for each method in the Android API. In particular, we modified Android 2.2's

permission verification mechanism to log permission checks as they occur. We then generated unit test cases for API calls, Content Providers, and Intents. Executing these tests allowed us to observe the permissions required to interact with system APIs. A core challenge was to build unit tests that obtain call coverage of all platform resources.

3.1 The API

As described in §2.2.1, the Android API provides applications with a library that includes public, private, and hidden classes and methods. The set of private classes includes the RPC stubs for the system services.[1] All of these classes and methods are accessible to applications using Java reflection, so we must test them to identify permission checks. We conducted testing in three phases: feedback-directed testing; customizable test case generation; and manual verification.

3.1.1 Feedback-Directed Testing

For the first phase of testing, we used Randoop, an automated, feedback-directed, object-oriented test generator for Java [20, 22]. Randoop takes a list of classes as input and searches the space of possible sequences of methods from these classes. We modified Randoop to run as an Android application and to log every method it invokes. Our modifications to Android log every permission that is checked by the Android permission validation mechanism, which lets us deduce which API calls trigger permission checks.

Randoop searches the space of methods to find methods whose return values can be used as parameters for other methods. It maintains a pool of valid initial input sequences and parameters, initially seeded with primitive values (e.g., `int` and `String`). Randoop builds test sequences incrementally by randomly selecting a method from the test class's methods and selecting sequences from the input pool to populate the method's arguments. If the new sequence is unique, then it is executed. Sequences that complete successfully (i.e., without generating an exception) are added to the sequence pool. Randoop's goal is full coverage of the test space. Unlike comparable techniques [4, 9, 21], Randoop does not need a sample execution trace as input, making large-scale testing such as API fuzzing more manageable. Because Randoop uses Java reflection to generate the test methods from the supplied list of classes, it supports testing non-public methods. We modified Randoop to also test nested classes of the input classes.

Limitations. Randoop's feedback-guided space exploration is limited by the objects and input values it has access to. If Randoop cannot find an object of the correct type needed to invoke a method in the sequence pool, then it will never try to invoke the method. The Android API is too large to test all interdependent classes at once, so in practice many objects are not available in the sequence pool. We mitigated this problem by testing related classes together (for example, `Account` and `AccountManager`) and adding seed sequences that return common Android-specific data types. Unfortunately, this was insufficient to produce valid input parameters for many methods. Many singleton object instances can only be created through API calls with specific parameters; for example, a `WifiManager` instance can be obtained by calling `android.content.Context.getSystemService(String)` with the parameter `"wifi"`. We addressed this by augmenting the input pool with specific primitive constants and sequences. Additionally, some API calls expect memory addresses that store specific values for parameters, which we were unable to solve at scale.

Randoop also does not handle ordering requirements that are independent of input parameters. In some cases, Android expects methods to precede each other in a very specific order. Randoop only generates sequence chains for the purpose of creating arguments for methods; it is not able to generate sequences to satisfy dependencies that are not in the form of an input variable. Further aggravating this problem, many Android methods with underlying native code generate segmentation faults if called out of order, which terminates the Randoop testing process.

3.1.2 Customizable Test Case Generation

Randoop's feedback-directed approach to testing failed to cover certain types of methods. When this happened, there was no way to manually edit its test sequences to control sequence order or establish method pre-conditions. To address these limitations and improve coverage, we built our own test generation tool. Our tool accepts a list of method signatures as input, and outputs at least one unit test for each method. It maintains a pool of default input parameters that can be passed to methods to be called. If multiple values are available for a parameter, then our tool creates multiple unit tests for that method. (Tests are created combinatorially when multiple parameters of the same method have multiple possible values.) It also generates tests using null values if it cannot find a suitable parameter. Because our tool separates test case generation from execution, a human tester can edit the test sequences produced by our tool. When tests fail, we manually adjust the order of method calls, introduce extra code to satisfy method pre-conditions, or add new parameters for the failing tests.

Our test generation tool requires more human effort than Randoop, but it is effective for quickly achieving coverage of methods that Randoop was unable to properly invoke. Overseeing and editing a set of generated test cases produced by our tool is still substantially less work than manually writing test cases. Our experience with large-scale API testing was that methods that are challenging to invoke by feedback-directed testing occur often enough to be problematic. When a human tester has the ability to edit failing sequences, these methods can be properly invoked.

3.1.3 Manual Verification

The first two phases of testing generate a map of the permission checks performed by each method in the API. However, these results contain three types of inconsistencies. First, the permission checks caused by asynchronous API calls are sometimes incorrectly associated with subsequent API calls. Second, a method's permission requirements can be argument-dependent, in which case we see intermittent or different permission checks for that method. Third, permission checks can be dependent on the order in which API calls are made. To identify and resolve these inconsistencies, we manually verified the correctness of the permission map generated by the first two phases of testing.

[1] The operating system also includes many internal methods that make permission checks. However, applications cannot invoke them because they are not currently exposed with RPC stubs. Since we are focused on the application-facing API, we do not test or discuss these permission checks.

We used our customizable test generation tool to create tests to confirm the permission(s) associated with each API method in our permission map. We carefully experimented with the ordering and arguments of the test cases to ensure that we correctly matched permission checks to asynchronous API calls and identified the conditions of permission checks. When confirming permissions for potentially asynchronous or order-dependent API calls, we also created confirmation test cases for related methods in the pertinent class that were not initially associated with permission checks. We ran every test case both with and without their required permissions in order to identify API calls with multiple or substitutable permission requirements. If a test case throws a security exception without a permission but succeeds with a permission, then we know that the permission map for the method under test is correct.

Testing The Internet Permission. Applications can access the Internet through the Android API, but other packages such as `java.net` and `org.apache` also provide Internet access. In order to determine which methods require access to the Internet, we scoured the documentation and searched the Internet for any and all methods that suggest Internet access. Using this list, we wrote test cases to determine which of those methods require the `INTERNET` permission.

3.2 Content Providers

Our Content Provider test application executes `query`, `insert`, `update`, and `delete` operations on Content Provider URIs associated with the Android system and preinstalled appliactions. We collected a list of URIs from the `android.provider` package to determine the core set of Content Providers to test. We additionally collected Content Provider URIs that we discovered during other phases of testing. For each URI, we attempted to execute each type of database operation without any permissions. If a security exception was thrown, we recorded the required permission. We added and tested combinations of permissions to identify multiple or substitutable permission requirements. Each Content Provider was tested until security exceptions were no longer thrown for a given operation, indicating the minimum set of permissions required to complete that operation. In addition to testing, we also examined the system Content Providers' static permission declarations.

3.3 Intents

We built a pair of applications to send and receive Intents. The Android documentation does not provide a single, comprehensive list of the available system Intents, so we scraped the public API to find string constants that could be the contents of an Intent.[2] We sent and received Intents with these constants between our test applications. In order to test the permissions needed to receive system broadcast Intents, we triggered system broadcasts by sending and receiving text messages, sending and receiving phone calls, connecting and disconnecting WiFi, connecting and disconnecting Bluetooth devices, etc. For all of these tests, we recorded whether permission checks occurred and whether the Intents were delivered or received successfully.

[2]For those familiar with Android terminology, we searched for Intent *action* strings.

4. PERMISSION MAP RESULTS

Our testing of the Android application platform resulted in a permission map that correlates permission requirements with API calls, Content Providers, and Intents. In this section, we discuss our coverage of the API, compare our results to the official Android documentation, and present characteristics of the Android API and permission map.

4.1 Coverage

The Android API consists of 1,665 classes with a total of 16,732 public and private methods. We attained **85%** coverage of the Android API through two phases of testing. (We define a method as *covered* if we executed it without generating an exception; we do not measure branch coverage.) Randoop attained an initial method coverage of 60%, spread across all packages. We supplemented Randoop's coverage with our proprietary test generation tool, accomplishing close to 100% coverage of the methods that belong to classes with at least one permission check.

The uncovered portion of the API is due to native calls and the omission of second-phase tests for packages that did not yield permission checks in the first phase. First, native methods often crashed the application when incorrect parameters were supplied, making them difficult to test. Many native method parameters are integers that represent pointers to objects in native code, making it difficult to supply correct parameters. Approximately one-third of uncovered methods are native calls. Second, we decided to omit supplemental tests for packages that did not reveal permission checks during the Randoop testing phase. If Randoop did not trigger at least one permission check in a package, we did not add more tests to the classes in the package.

4.2 Comparison With Documentation

Clear and well-developed documentation promotes correct permission usage and safe programming practices. Errors and omissions in the documentation can lead to incorrect developer assumptions and overprivilege. Android's documentation of permissions is limited, which is likely due to their lack of a centralized access control policy. Our testing identified 1,259 API calls with permission checks. We compare this to the Android 2.2 documentation.

We crawled the Android 2.2 documentation and found that it specifies permission requirements for 78 methods. The documentation additionally lists permissions in several class descriptions, but it is not clear which methods of the classes require the stated permissions. Of the 78 permission-protected API calls in the documentation, our testing indicates that the documentation for 6 API calls is incorrect. It is unknown to us whether the documentation or implementation is wrong; if the documentation is correct, then these discrepancies may be security errors.

Three of the documentation errors list a different permission than was found through testing. In one place, the documentation claims an API call is protected by the Dangerous permission `MANAGE_ACCOUNTS`, when it actually can be accessed with the lower-privilege Normal permission `GET_ACCOUNTS`. Another error claims an API call requires the `ACCESS_COARSE_UPDATES` permission, which does not exist. As a result, 5 of the 900 applications that we study in §6.2 request this non-existent permission. A third error states that a method is protected with the `BLUETOOTH` permission, when the method is in fact protected with `BLUETOOTH_ADMIN`.

Permission	Usage
BLUETOOTH	85
BlUETOOTH_ADMIN	45
READ_CONTACTS	38
ACCESS_NETWORK_STATE	24
WAKE_LOCK	24
ACCESS_FINE_LOCATION	22
WRITE_SETTINGS	21
MODIFY_AUDIO_SETTINGS	21
ACCESS_COARSE_LOCATION	18
CHANGE_WIFI_STATE	16

Table 1: Android's 10 most checked permissions.

The other three documentation errors pertain to methods with multiple permission requirements. In one error, the documentation claims that a method requires one permission, but our testing shows that two are required. For the last two errors, the documentation states that two methods require one permission each; in practice, however, the two methods both accept two permissions (i.e., they are ORs).

4.3 Characterizing Permissions

Based on our permission map, we characterize how permission checks are distributed throughout the API.

4.3.1 API Calls

We examined the Android API to see how many methods and classes have permission checks. We present the number of permission checks, unused permissions, hierarchical permissions, permission granularity, and class characteristics.

Number of Permissions Checks. We identified 1,244 API calls with permission checks, which is 6.45% of all API methods (including hidden and private methods). Of those, 816 are methods of normal API classes, and 428 are methods of RPC stubs that are used to communicate with system services. We additionally identified 15 API calls with permission checks in a supplementary part of the API added by a manufacturer, for a total of 1,259 API calls with permission checks. Table 1 provides the rates of the most commonly-checked permissions for the normal API.

Signature/System Permissions. We found that 12% of the normal API calls are protected with Signature/System permissions, and 35% of the RPC stubs are protected with Signature/System permissions. This effectively limits the use of these API calls to pre-installed applications.

Unused Permissions. We found that some permissions are defined by the platform but never used within the API. For example, the BRICK permission is never used, despite being oft-cited as an example of a particularly dire permission [26]. The only use of the BRICK permission is in dead code that is incapable of causing harm to the device. Our testing found that 15 of the 134 Android-defined permissions are unused. For each case where a permission was never found during testing, we searched the source tree to verify that the permission is not used. After examining several devices, we discovered that one of the otherwise unused permissions is used by the custom classes that HTC and Samsung added to the API to support 4G on their phones.

Hierarchical Permissions. The names of many permissions imply that there are hierarchical relationships between them. Intuitively, we expect that more powerful permissions should be substitutable for lesser permissions relating to the same resource. However, we find no evidence of planned hierarchy. Our testing indicates that BLUETOOTH_ADMIN is not substitutable for BLUETOOTH, nor is WRITE_CONTACTS substitutable for READ_CONTACTS. Similarly, CHANGE_WIFI_STATE cannot be used in place of ACCESS_WIFI_STATE.

Only one pair of permissions has a hierarchical relationship: ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION. Every method that accepts the COARSE permission also accepts FINE as a substitute. We found only one exception to this, which may be a bug: TelephonyManager.listen() accepts either ACCESS_COARSE_LOCATION or READ_PHONE_STATE, but it does not accept ACCESS_FINE_LOCATION.

Permission Granularity. If a single permission is applied to a diverse set of functionality, applications that request the permission for a subset of the functionality will have unnecessary access to the rest. Android aims to prevent this by splitting functionality into multiple permissions when possible, and their approach has been shown to benefit platform security [15]. As a case study, we examine the division of Bluetooth functionality, as the Bluetooth permissions are the most heavily checked permissions.

We find that the two Bluetooth permissions are applied to 6 large classes. They are divided between methods that change state (BLUETOOTH_ADMIN) and methods that get device information (BLUETOOTH). The BluetoothAdapter class is one of several that use the Bluetooth permissions, and it appropriately divides most of its permission assignments. However, it features some inconsistencies. One method only returns information but requires the BLUETOOTH_ADMIN permission, and another method changes state but requires both permissions. This type of inconsistency may lead to developer confusion about which permissions are required for which types of operations.

Class Characteristics. Figure 2 presents the percentage of methods that are protected per class. We initially expected that the distribution would be bimodal, with most classes protected entirely or not at all. Instead, however, we see a wide array of class protection rates. Of these classes, only 8 require permissions to instantiate an object, and 4 require permissions only for the object constructor.

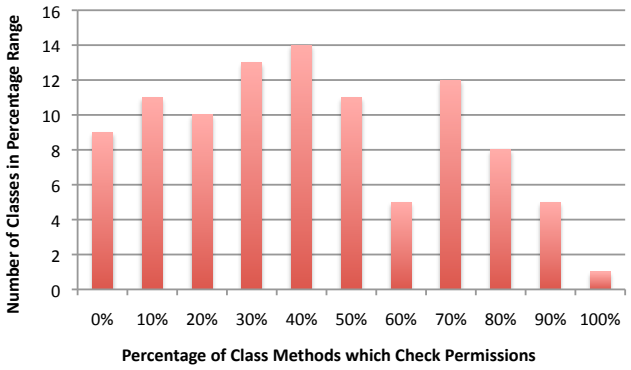

Figure 2: A histogram of the number of classes, sorted by the percentage of the classes' methods that require permissions. The numbers shown represent ranges, i.e., 10% represents $[10-20\%)$. We only consider classes with at least 1 permission check.

4.3.2 Content Providers and Intents

We examined Content Providers to determine whether they are protected by permissions. We investigated a total of 62 Content Providers. We found that there are 18 Content Providers that do not have permissions for any of the methods that we tested (insert, query, update, and delete). All of the Content Providers that lack permissions are associated with the `content://media` content URI.

We examined Intent communication and measured whether permissions are required for sending and receiving Intents. When sending broadcast Intents, 62 broadcasts are prohibited by non-system senders, 6 require permissions before sending the Intent, and 2 can be broadcast but not received by system receivers. Broadcast receivers must have permissions to receive 23 broadcast Intents, of which 14 are protected by a Bluetooth permission. When sending Intents to start Activities, 7 Intent messages require permissions. When starting Services, 2 Intents require permissions.

5. APPLICATION ANALYSIS TOOL

We built a static analysis tool, Stowaway, which analyzes an Android application and determines the maximum set of permissions it may require. Stowaway analyzes the application's use of API calls, Content Providers, and Intents and then uses the permission map built in §3 to determine what permissions those operations require.

Compiled applications for the Android platform include Dalvik executable (DEX) files that run on Android's Dalvik Virtual Machine. We disassemble application DEX files using the publicly available Dedexer tool [23]. Each stage of Stowaway takes the disassembled DEX as input.

5.1 API Calls

Stowaway first parses the disassembled DEX files and identifies all calls to standard API methods. Stowaway tracks application-defined classes that inherit methods from Android classes so we can differentiate between invocations of application-defined methods and Android-defined inherited methods. We use heuristics to handle Java reflection and two unusual permissions.

Reflection. Java reflection is a challenging problem [6, 18, 24]. In Java, methods can be reflectively invoked with `java.lang.reflect.Method.invoke()` or `java.lang.reflect.Constructor.newInstance()`. Stowaway tracks which Class objects and method names are propagated to the reflective invocation. It performs flow-sensitive, intra-procedural static analysis, augmented with inter-procedural analysis to a depth of 2 method calls. Within each method body, it tracks the value of each String, StringBuilder, Class, Method, Constructor, Field, and Object. We also track the state of static member variables of these types. We identify method calls that convert strings and objects to type Class, as well as method calls that convert Class objects to Methods, Constructors, and Fields.

We also apply Android-specific heuristics to resolving reflection by handling methods and fields that may affect reflective calls. We cannot model the behavior of the entire Android and Java APIs, but we identify special cases. First, `Context.getSystemService(String)` returns different types of objects depending on the argument. We maintain a mapping of arguments to the types of return objects. Second, some API classes contain private member variables that hold references to hidden interfaces. Applications can only access these member variables reflectively, which obscures their type information. We created a mapping between member variables and their types and propagate the type data accordingly. If an application subsequently accesses methods on a member variable after retrieving it, we can resolve the member variable's type.

Internet. Any application that includes a *WebView* must have the Internet permission. A WebView is a user interface component that allows an application to embed a web site into its UI. WebViews can be instantiated programmatically or declared in XML files. Stowaway identifies programmatic instantiations of WebViews. It also decompiles application XML files and parses them to detect WebView declarations.

External Storage. If an application wants to access files stored on the SD card, it must have the `WRITE_EXTERNAL_STORAGE` permission. This permission does not appear in our permission map because it (1) is enforced entirely using Linux permissions and (2) can be associated with any file operation or API call that accesses the SD card from within the library. We handle this permission by searching the application's string literals and XML files for strings that contain `sdcard`; if any are found, we assume `WRITE_EXTERNAL_STORAGE` is needed. Additionally, we assume this permission is needed if we see API calls that return paths to the SD card, such as `Environment.getExternalStorageDirectory()`.

5.2 Content Providers

Content Providers are accessed by performing a database operation on a URI. Stowaway collects all strings that could be used as Content Provider URIs and links those strings to the Content Providers' permission requirements. Content Provider URIs can be obtained in two ways:

1. A string or set of strings can be passed into a method that returns a URI. For example, the API call `android.net.Uri.parse("content://browser/bookmarks")` returns a URI for accessing the Browser bookmarks. To handle this case, Stowaway finds all string literals that begin with `content://`.
2. The API provides Content Provider helper classes that include public URI constants. For example, the value of `android.provider.Browser.BOOKMARKS_URI` is `content://browser/bookmarks`. Stowaway recognizes known URI constants, and we created a mapping from all known URI constants to their string values.

A limitation of our tool is that we cannot tell which database operations an application performs with a URI; there are many ways to perform an operation on a Content Provider, and users can set their own query strings. To account for this, we say that an application may require any permission associated with any operation on a given Content Provider URI. This provides an upper bound on the permissions that could be required in order to use a specific Content Provider.

5.3 Intents

We use ComDroid [8] to detect the sending and receiving of Intents that require permissions. ComDroid performs flow-sensitive, intra-procedural static analysis, augmented with limited inter-procedural analysis that follows method invocations to a depth of one method call. ComDroid tracks the state of Intents, registers, sinks (e.g., `sendBroadcast`), and application components. When an Intent object is in-

stantiated, passed as a method parameter, or obtained as a return value, ComDroid tracks all changes to it from its source to its sink and outputs all information about the Intent and all components expecting to receive messages.

Stowaway takes ComDroid's output and, for each sent Intent, checks whether a permission is required to send that Intent. For each Intent that an application is registered to receive, Stowaway checks whether a permission is required to receive the Intent. Occasionally ComDroid is unable to identify the message or sink of an Intent. To mitigate these cases, Stowaway searches for protected Intents in the list of all string literals in the application.

6. APPLICATION ANALYSIS RESULTS

We applied Stowaway to 940 Android applications to identify the prevalence of overprivilege. Applications with unnecessary permissions violate the principle of least privilege. Overprivilege undermines the benefits of a per-application permission system: extra permissions unnecessarily condition users to casually accept dangerous permissions and needlessly exacerbate application vulnerabilities.

Stowaway calculates the maximum set of Android permissions that an application may need. We compare that set to the permissions actually requested by the application. If the application requests more permissions, then it is overprivileged. Our full set of applications consists of 964 Android 2.2 applications.[3] We set aside 24 randomly selected applications for tool testing and training, leaving 940 for analysis.

6.1 Manual Analysis

6.1.1 Methodology

We randomly selected 40 applications from the set of 940 and ran Stowaway on them. Stowaway identified 18 applications as overprivileged. We then manually analyzed each overprivilege warning to attribute it to either tool error (i.e., a false positive) or developer error. We looked for false positives due to three types of failures:

1. Stowaway misses an API, Content Provider, or Intent operation that needs a permission. For example, Stowaway misses an API call when it cannot resolve the target of a reflective call.
2. Stowaway correctly identifies the API, Content Provider, or Intent operation, but our permission map lacks an entry for that platform resource.
3. The application sends an Intent to some other application, and the recipient accepts Intents only from senders with a certain permission. Stowaway cannot detect this case because we cannot determine the permission requirements of other non-system applications.

We reviewed the 18 applications' bytecode, searching for any of these three types of error. If we found functionality that could plausibly pertain to a permission that Stowaway identified as unnecessary, we manually wrote additional test cases to confirm the accuracy of our permission map. We investigated the third type of error by checking whether the application sends Intents to pre-installed or well-known applications. When we determined that a warning was not a false positive, we attempted to identify why the developer had added the unnecessary permission.

[3]In October 2010, we downloaded the 100 most popular paid applications, the 764 most popular free applications, and 100 recently added free applications from the Android Market.

We also analyzed overprivilege warnings by running the application in our modified version of Android (which records permission checks as they occur) and interacting with it. It was not possible to test all applications at runtime; for example, some applications rely on server-side resources that have moved or changed since we downloaded them. We were able to test 10 of the 18 application in this way. In each case, runtime testing confirmed the results of our code review.

6.1.2 False Positives

Stowaway identified 18 of the 40 applications (45%) as having 42 unnecessary permissions. Our manual review determined that 17 applications (42.5%) are overprivileged, with a total of 39 unnecessary permissions. This represents a **7%** false positive rate.

All three of the false warnings were caused by incompleteness in our permission map. Each was a special case that we failed to anticipate. Two of the three false positives were caused by applications using `Runtime.exec` to execute a permission-protected shell command. (For example, the `logcat` command performs a `READ_LOGS` permission check.) The third false positive was caused by an application that embeds a web site that uses HTML5 geolocation, which requires a location permission. We wrote test cases for these scenarios and updated our permission map.

Of the 40 applications in this set, 4 contain at least one reflective call that our static analysis tool cannot resolve or dismiss. 2 of them are overprivileged. This means that 50% of the applications with at least one unresolved reflective call are overprivileged, whereas other applications are overprivileged at a rate of 42%. However, a sample size of 4 is too small to draw conclusions. We investigated the unresolved reflective calls and do not believe they led to false positives.

6.2 Automated Analysis

We ran Stowaway on 900 Android applications. Overall, Stowaway identified 323 applications (35.8%) as having unnecessary permissions. Stowaway was unable to resolve some applications' reflective calls, which might lead to a higher false positive rate in those applications. Consequently, we discuss applications with unresolved reflective calls separately from other applications.

6.2.1 Applications With Fully Handled Reflection

Stowaway was able to handle all reflective calls for 795 of the 900 applications, meaning that it should have identified all API access for those applications. Stowaway produces overprivilege warnings for 32.7% of the 795 applications. Table 2 shows the 10 most common unnecessary permissions among these applications.

56% of overprivileged applications have 1 extra permission, and 94% have 4 or fewer extra permissions. Although one-third of applications are overprivileged, the low degree of per-application overprivilege indicates that developers are attempting to add correct permissions rather than arbitrarily requesting large numbers of unneeded permissions. This supports the potential effectiveness of install-time permission systems like Android's.

We believe that Stowaway should produce approximately the same false positive rate for these applications as it did for the set of 40 that we evaluated in §6.1. If we assume that the 7% false positive rate from our manual analysis applies to these results, then 30.4% of the 795 applications

Permission	Usage
ACCESS_NETWORK_STATE	16%
READ_PHONE_STATE	13%
ACCESS_WIFI_STATE	8%
WRITE_EXTERNAL_STORAGE	7%
CALL_PHONE	6%
ACCESS_COARSE_LOCATION	6%
CAMERA	6%
WRITE_SETTINGS	5%
ACCESS_MOCK_LOCATION	5%
GET_TASKS	5%

Table 2: The 10 most common unnecessary permissions and the percentage of overprivileged applications that request them.

	Apps with Warnings	Total Apps	Rate
Reflection, failures	56	105	53%
Reflection, no failures	151	440	34%
No reflection	109	355	31%

Table 3: The rates at which Stowaway issues overprivilege warnings, by reflection status.

are truly overprivileged. Applications could also be *more* overprivileged in practice than indicated by our tool, due to unreachable code. Stowaway does not perform dead code elimination; dead code elimination for Android applications would need to take into account the unique Android lifecycle and application entry points. Additionally, our overapproximation of Content Provider operations (§5.2) might overlook some overprivilege. We did not quantify Stowaway's false negative rate, and we leave dead code elimination and improved Content Provider string tracking to future work.

6.2.2 The Challenges of Java Reflection

Reflection is commonly used in Android applications. Of the 900 applications, 545 (61%) use Java reflection to make API calls. We found that reflection is used for many purposes, such as to deserialize JSON and XML, invoke hidden or private API calls, and handle API classes whose names changed between versions. The prevalence of reflection indicates that it is important for an Android static analysis tool to handle Java reflection, even if the static analysis tool is not intended for obfuscated or malicious code.

Stowaway was able to fully resolve the targets of reflective calls in 59% of the applications that use reflection. We handled a further 117 applications with two techniques: eliminating failures where the target class of the reflective call was known to be defined within the application, and manually examining and handling failures in 21 highly popular libraries. This left us with 105 applications with reflective calls that Stowaway could not resolve or dismiss, which is 12% of the 900 applications.

Stowaway identifies 53.3% of the 105 applications as overprivileged. Table 3 compares this to the rate at which warnings are issued for applications without unhandled reflections. There are two possible explanations for the difference: Stowaway might have a higher false positive rate in applications with unresolved reflective calls, or applications that use Java reflection in complicated ways might have a higher rate of actual overprivilege due to a correlated trait.

We suspect that both factors play a role in the higher overprivilege warning rate in applications with unhandled reflective calls. Although our manual review (§6.1) did not find that reflective failures led to false positives, a subsequent review of additional applications identified several erroneous warnings that were caused by reflection. On the other hand, developer error may increase with the complexity associated with complicated reflective calls.

Improving the resolution of reflective calls in Android applications is an important open problem. Stowaway's reflection analysis fails when presented with the creation of method names based on non-static environment variables, direct generation of Dalvik bytecode, arrays with two pointers that reference the same location, or `Method` and `Class` objects that are stored in hash tables. Stowaway's primarily linear traversal of a method also experiences problems with non-linear control flow, such as `jump`s; we only handle simple `goto`s that appear at the ends of methods. We also observed several applications that iterate over a set of classes or methods, testing each element to decide which one to invoke reflectively. If multiple comparison values are tested and none are used within the block, Stowaway only tracks the last comparison value beyond the block; this value may be `null`. Future work may be able to solve some of these problems, possibly with the use of dynamic analysis.

6.3 Common Developer Errors

In some cases, we are able to determine why developers asked for unnecessary permissions. Here, we consider the prevalence of different types of developer error among the 40 applications from our manual review and the 795 fully handled applications from our automated analysis.

Permission Name. Developers sometimes request permissions with names that sound related to their applications' functionality, even if the permissions are not required. For example, one application from our manual review unnecessarily requests the `MOUNT_UNMOUNT_FILESYSTEMS` permission to receive the `android.intent.action.MEDIA_MOUNTED` Intent. As another example, the `ACCESS_NETWORK_STATE` and `ACCESS_WIFI_STATE` permissions have similar-sounding names, but they are required by different classes. Developers often request them in pairs, even if only one is necessary. Of the applications that unnecessarily request the network permission, 32% legitimately require the WiFi permission. Of the applications that unnecessarily request the WiFi permission, 71% legitimately need the network permission.

Deputies. An application can send an Intent to another *deputy* application, asking the deputy to perform an operation. If the deputy makes a permission-protected API call, then the deputy needs a permission. The sender of the Intent, however, does not. We noticed instances of applications requesting permissions for actions that they asked deputies to do. For example, one application asks the Android Market to install another application. The sender asks for `INSTALL_PACKAGES`, which it does not need because the Market application does the installation.

We find widespread evidence of this type of error. Of the applications that unnecessarily request the `CAMERA` permission, 81% send an Intent that opens the default Camera application to take a picture. 82% of the applications that unnecessarily request `INTERNET` send an Intent that opens a URL in the browser. Similarly, 44% of the applications that unnecessarily request `CALL_PHONE` send an Intent to the default Phone Dialer application.

Related Methods. As shown in Figure 2, most classes contain a mix of permission-protected and unprotected methods. We have observed applications that use unprotected methods but request permissions that are required for other methods in the same class. For example, `android.provider.Settings.Secure` is a convenience class in the API for accessing the Settings Content Provider. The class includes both setters and getters. The setters require the `WRITE_SETTINGS` permission, but the getters do not. Two of the applications that we manually reviewed use only the getters but request the `WRITE_SETTINGS` permission.

Copy and Paste. Popular message boards contain Android code snippets and advice about permission requirements. Sometimes this information is inaccurate, and developers who copy it will overprivilege their applications. For example, one of the applications that we manually reviewed registers to receive the `android.net.wifi.STATE_CHANGE` Intent and requests the `ACCESS_WIFI_STATE` permission. As of May 2011, the third-highest Google search result for that Intent contains the incorrect assertion that it requires that permission [25].

Deprecated Permissions. Permissions that are unnecessary in Android 2.2 could be necessary in older Android releases. Old or backwards-compatible applications therefore might have seemingly extra permissions. However, developers may also accidentally use these permissions because they have read out-of-date material. 8% of the overprivileged applications request either `ACCESS_GPS` or `ACCESS_LOCATION`, which were deprecated in 2008. Of those, all but one specify that their lowest supported API version is *higher* than the last version that included those permissions.

Testing Artifacts. A developer might add a permission during testing and then forget to remove it when the test code is removed. For example, `ACCESS_MOCK_LOCATION` is typically used only for testing but can be found in released applications. All of the applications in our data set that unnecessarily include the `ACCESS_MOCK_LOCATION` permission also include a real location permission.

Signature/System Permissions. We find that 9% of overprivileged applications request unneeded Signature or SignatureOrSystem permissions. Standard versions of Android will silently refuse to grant those permissions to applications that are not signed by the device manufacturer. The permissions were either requested in error, or the developers removed the related code after discovering it did not work on standard handsets.

We can attribute many instances of overprivilege to developer confusion over the permission system. Confusion over permission names, related methods, deputies, and deprecated permissions could be addressed with improved API documentation. To avoid overprivilege due to related methods, we recommend listing permission requirements on a per-method (rather than per-class) basis. Confusion over deputies could be reduced by clarifying the relationship between permissions and pre-installed system applications.

Despite the number of unnecessary permissions that we can attribute to error, it is possible that some developers request extra permissions intentionally. Developers are incentivized to ask for unnecessary permissions because applications will not receive automatic updates if the updated version of the application requests more permissions [15].

7. RELATED WORK

Android Permissions. Previous studies of Android applications have been limited in their understanding of permission usage. Our permission map can be used to greatly increase the scope of application analysis. Enck et al. apply Fortify's Java static analysis tool to decompiled applications; they study their API use [11]. However, they are limited to studying applications' use of a small number of permissions and API calls. In a recent study, Felt et al. manually classify a small set of Android applications as overprivileged or not, but they were limited by the Android documentation [15]. Kirin [12] reads application permission requirements during installation and checks them against a set of security rules. They rely solely on developer permission requests, rather than examining whether or how permissions are used by applications. Barrera et al. examine 1,100 Android applications' permission requirements and use self-organizing maps to visualize which permissions are used in applications with similar characteristics [5]. Their work also relies on the permissions requested by the applications.

Vidas et al. [27] provide a tool that performs an overprivilege analysis on application source code. Their tool could be improved by using our permission map; theirs is based on the limited Android documentation. Our static analysis tool also performs a more sophisticated application analysis. Unlike their Eclipse plugin, Stowaway attempts to handle reflective calls, Content Providers, and Intents.

In concurrent work, Gibler et al. [16] applied static analysis to the Android API to find permission checks. Their permission map includes internal methods within the system process that are not reachable across the RPC boundary, which we excluded because applications cannot access them. Unlike our dynamic approach, their static analysis might have false positives, will miss permission checks in native code, and will miss Android-specific control flow.

Java Testing. Randoop is not the only Java unit test generation tool. Tools like Eclat [21], Palulu [4] and JCrasher [9] work similarly but require an example execution as input. Given the size of the Android API, building such an example execution would be a challenge. Enhanced JUnit [14] generates tests by chaining constructors to some fixed depth. However, it does not use subtyping to provide instances and relies on bytecode as input. Korat [7] requires formal specifications of methods as input, which is infeasible for post-facto testing of the Android API.

Java Reflection. Handling Java reflection is necessary to develop sound and complete program analyses. However, resolving reflective calls is an area of open research. Livshits et al. created a static algorithm which approximates reflective targets by tracking string constants passed to reflections [18]. Their approach falls short when the reflective call depends on user input or environment variables. We use the same approach and suffer from the same limitations. They improve their results with developer annotations, which is not a feasible approach for our domain. A more advanced technique combines static analysis with information about the environment of the Java program in order to resolve reflections [24]. However, their results are sound only if the program is executed in an identical environment as the original evaluation. Even with their modifications, they are able to resolve only 74% of reflective calls in the Java 1.4 API. We do not claim

to improve the state of the art in resolving Java reflection; instead, we focus on domain-specific heuristics for how reflection is used in Android applications. We are the first to discuss reflection in Android applications.

8. CONCLUSION

In this paper, we developed tools to detect overprivilege in Android applications. We applied automated testing techniques to Android 2.2 to determine the permissions required to invoke each API method. Our tool, Stowaway, generates the maximum set of permissions needed for an application and compares them to the set of permissions actually requested. Currently, Stowaway is unable to handle some complex reflective calls, and we identify Java reflection as an important open problem for Android static analysis tools.

We applied Stowaway to 940 Android applications and found that about one-third of them are overprivileged. Our results show that applications generally are overprivileged by only a few permissions, and many extra permissions can be attributed to developer confusion. This indicates that developers attempt to obtain least privilege for their applications but fall short due to API documentation errors and lack of developer understanding.

Acknowledgements

We thank Royce Cheng-Yue and Kathryn Lingel for their help testing the API and Content Providers. This work is partially supported by NSF grants CCF-0424422, 0311808, 0832943, 0448452, 0842694, a gift from Google, and the MURI program under AFOSR grant FA9550-08-1-0352. This material is also based upon work supported under a NSF Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF.

9. REFERENCES

[1] Amazon Appstore for Android. `http://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011`.

[2] Android Developers Reference. `http://developer.android.com/reference/`.

[3] Android Market. `http://www.android.com/market/`.

[4] Artzi, S., Ernst, M., Kiezun, A., Pacheco, C., and Perkins, J. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *Workshop on Model-Based Testing and Object-Oriented Systems* (2006).

[5] Barrera, D., Kayacik, H., van Oorschot, P., and Somayaji, A. A methodology for empirical analysis of permission-based security models and its application to android. In *Proc. of the ACM conference on Computer and Communications Security* (2010).

[6] Bodden, E., Sewe, A., Sinschek, J., and Mezini, M. Taming reflection: Static analysis in the presence of reflection and custom class loaders. Tech. Rep. TUD-CS-2010-0066, CASED, Mar. 2010.

[7] Boyapati, C., Khurshid, S., and Marinov, D. Korat: Automated testing based on Java predicates. In *Proc. of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2002).

[8] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing Inter-Application Communication in Android. In *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services* (2011).

[9] Csallner, C., and Smaragdakis, Y. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience 34*, 11 (2004).

[10] Distimo. The battle for the most content and the emerging tablet market. `http://www.distimo.com/blog/2011_04_the-battle-for-the-most-content-and-the-emerging-tablet-market`.

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

[12] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proc. of the ACM conference on Computer and Communications Security* (2009).

[13] Enck, W., Ongtang, M., and McDaniel, P. Understanding Android security. *IEEE Security and Privacy 7*, 1 (2009).

[14] Enhanced JUnit. `http://www.silvermark.com/Product/java/enhancedjunit/index.html`.

[15] Felt, A. P., Greenwood, K., and Wagner, D. The Effectiveness of Application Permissions. In *Proc. of the USENIX Conference on Web Application Development* (2011).

[16] Gibler, C., Crussell, J., Erickson, J., and Chen, H. AndroidLeaks: Detecting Privacy Leaks in Android Applications. Tech. rep., UC Davis, 2011.

[17] Hackborn, D. Re: List of private / hidden / system APIs? `http://groups.google.com/group/android-developers/msg/a9248b18cba59f5a`.

[18] Livshits, B., Whaley, J., and Lam, M. S. Reflection Analysis for Java. In *Asian Symposium on Programming Languages and Systems* (2005).

[19] McCluskey, G. Using Java Reflection. `http://java.sun.com/developer/technicalArticles/ALT/Reflection/`, 1998.

[20] Pacheco, C., and Ernst, M. Randoop. `http://code.google.com/p/randoop/`.

[21] Pacheco, C., and Ernst, M. Eclat: Automatic generation and classification of test inputs. *European Conference on Object-Oriented Programming* (2005).

[22] Pacheco, C., Lahiri, S., Ernst, M., and Ball, T. Feedback-directed random test generation. In *Proc. of the International Conference on Software Engineering* (2007).

[23] Paller, G. Dedexer. `http://dedexer.sourceforge.net`.

[24] Sawin, J., and Rountev, A. Improving static resolution of dynamic class loading in java using dynamically gathered environment information. *Automated Software Eng. 16* (June 2009), 357–381.

[25] Stack Overflow. Broadcast Intent when network state has changend. `http://stackoverflow.com/questions/2676044/broadcast-intent-when-network-state-has-changend`.

[26] Vennon, T., and Stroop, D. Threat Analysis of the Android Market. Tech. rep., SMobile Systems, 2010.

[27] Vidas, T., Christin, N., and Cranor, L. Curbing Android Permission Creep. In *W2SP* (2011).