科技文献翻译

原文题目：Android Permissions Demystified

译文题目：神秘的 Android 权限

指导教师：李晓宇　职称：副教授

指导教师(校外)：张创伟　职称：研发经理

学生姓名：李伟　学号：20162430211

专　　业：软件工程

院（系）：信息工程学院

完成时间：2020 年 6 月 5 日

2020 年 6 月 5 日

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.

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.

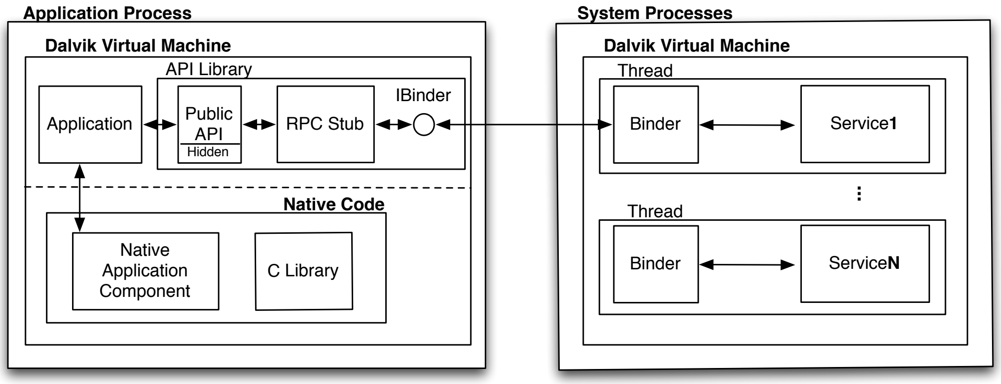

Figure 1: The architecture of the Android platform. Permission checks occur in the system process.

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 pre-installed 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 `OR`s).

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.

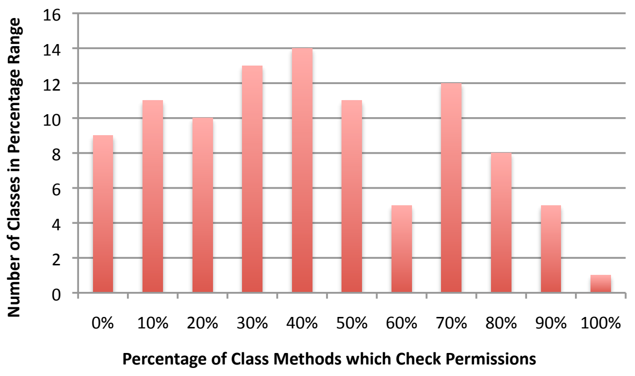

Figure 2: A histogram of the number of classes, sorted by the percentage of the classes' methods that require permissions. The numbers shown represent ranges, i.e., 10% represents $[10-20\%)$. We only consider classes with at least 1 permission check.

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 instantiated, 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).

作者：Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, David Wagner

摘要

Android 为第三方应用程序提供了广泛的功能包括访问手机硬件，设置和用户数据。访问与隐私和安全相关的部分 API 由安装时应用程序许可系统控制。我们研究 Android 应用程序，以确定 Android 开发人员是否遵循最低特权权限请求。我们构建了 Stowaway，该工具可以检测已编译的 Android 应用程序中的特权。偷渡者确定应用程序使用的 API 调用集，并然后将这些 API 调用映射到权限。我们使用 Android API 上的自动化测试工具来构建检测超权限所必需的权限图。我们将 Stowaway 应用于 940 个应用程序，并且发现大约三分之一的人享有特权。我们调查特权过高的原因，并找到证据表明开发商正在尝试遵循最小特权，但有时由于 API 文档不足。

类别和主题描述符

D.2.5 [软件工程]：测试和调试；
D.4.6 [操作系统]：安全性和保护

一般条款

安全

关键字

Android, permissions, least privilege

1. 介绍

Android 不受限制的应用程序市场和开源使其成为第三方应用程序的流行平台。截至 2011 年，Android Market 包含更多应用程序比 Apple App Store [10]。 Android 通过广泛的 API 支持第三方开发，该 API 为应用程序提供对电话硬件（例如摄像头），WiFi 和蜂窝网络，用户数据和电话设置。

访问 Android 的与隐私和安全性相关的部分丰富的 API 由安装时应用程序权限系统控制。每个应用程序必须预先声明所需的权限，并且在安装过程中会向用户通知其将获得的权限。如果用户不想授予应用程序许可，他或她可以取消安装过程。

安装时权限可以为用户提供控制保护他们的隐私，并减少错误和漏洞对应用程序的影响。但是，如果开发人员通常要求安装时间许可系统无效超出所需的权限。特权过高的应用程序使用户面临不必要的权限警告和增加错误或漏洞的影响。我们研究 Android 应用程序，以确定 Android 开发人员是否遵循最低特权或过度特权其应用程序。

我们提供了一种 Stowaway 工具，可以检测到特权过高在已编译的 Android 应用程序中。偷渡者组成分为两个部分：确定什么内容的静态分析工具 API 会调用应用程序生成的内容，以及一个标识每个 API 调用需要哪些权限。 Android 的文档没有提供足够的权限信息以进行此类分析，因此我们根据经验确定了 Android 2.2 的访问控制策略。使用自动化测试技术，我们实现了 85％的 Android 覆盖率 API。我们的权限图可让您深入了解 Android 许可系统，使我们能够识别超权限。

我们将 Stowaway 从以下版本应用到 940 个 Android 应用程序：Android Market，发现大约三分之一的应用程序拥有特权。特权过高的应用通常要求的额外特权很少：仅一半以上包含一个额外的权限，只有 6％的请求超过四个不必要的权限。 我们调查原因特权过剩，并发现许多开发人员错误源于对许可系统的困惑。我们的结果表示开发人员正在尝试遵循最小特权，支持诸如 Android 的安装时许可系统的潜在有效性。

Android 提供了开发人员文档，但其权限信息有限。缺少可靠的权限信息可能会导致开发人员错误。该文档仅列出了 78 种方法的权限要求，而我们的测试则揭示了针对以下方法的权限要求 1 259 种方法（比文档多 16 倍）。此外，我们确定了 Android 中的 6 个错误权限文档。 这种不精确性使开发人员可以在猜测和留言板上添加参考资料。开发人员的困惑可能导致特权过高应用程序，因为开发人员添加了不必要的权限试图使应用程序正常工作。

贡献。我们作出以下贡献：

1. 我们开发了 Stowaway，这是一种用于检测 Android 应用程序中超权限的工具。我们使用 Stowaway 和发现大约三分之一的人享有特权。
2. 我们确定并量化了导致特权过高的开发人员的错误模式
3. 我们使用自动化测试技术来确定 Android 的访问控制策略。与文档相比，我们的结果提高了 15 倍。

其他现有工具[11，12]和将来的程序分析可以利用我们的权限图来研究 Android 应用程序中的权限使用情况。可以在 android-permissions.org 上获得 Stowaway 和权限图数据。

组织。第 2 节概述了 Android 及其权限系统，第 3 节讨论了我们的 API 测试方法，第 4 节介绍了我们对 Android API 的分析。第 5 节介绍了用于检测过度特权的静态分析工具，而第 6 部分则讨论了我们的应用程序过度特权分析。

2. ANDROID 权限系统

Android 具有广泛的 API 和权限系统。我们首先提供 Android 应用程序平台和权限的高级概述。然后，我们将详细介绍如何强制执行 Android 权限。

2.1 Android 背景

Android 智能手机用户可以通过 Android Market 或 Amazon Appstore 安装第三方应用程序。这些第三方应用程序的质量和可信度差异很大，因此 Android 将所有应用程序视为潜在的漏洞或恶意软件。每个应用程序在具有低特权用户 ID 的进程中运行，并且默认情况下，应用程序只能访问自己的文件。应用程序用 Java 编写（可能随附本机代码），并且每个应用程序都在自己的虚拟机中运行。

Android 通过安装时权限控制对系统资源的访问。 Android 2.2 定义了 134 个权限，分为三个威胁级别：

普通权限可以保护对 API 调用的访问，这些访问可能会惹恼但不会伤害用户。例如，SET_WALLPAPER 控制更改用户背景墙纸的功能。

危险权限控制对可能有害的 API 调用的访问，例如与花钱或收集私人信息有关的调用。例如，发送短信或阅读联系人列表需要危险权限。

签名/系统权限可控制对最危险特权的访问，例如控制备份过程或删除应用程序包的能力。 这些权限很难获得：签名权限仅授予使用设备制造商证书签名

的应用程序，而 SignatureOrSystem 权限则授予使用特殊系统文件夹签名或安装的应用程序。这些限制实际上将签名/系统权限限制为预安装的应用程序，其他应用程序对签名/系统权限的请求将被忽略。

应用程序可以出于自我保护的目的定义自己的权限，但是我们专注于保护系统资源的 Android 定义的权限。在分析的任何阶段，我们都不会考虑开发人员定义的权限。同样，我们不会考虑包含在 Google 应用程序（例如 Google Reader）中但不是操作系统一部分的 Google 定义的权限。

与系统 API，数据库和消息传递系统进行交互时，可能需要权限。公用 API 描述 8648 个方法，其中一些受权限保护。用户数据存储在内容提供程序中，并且在某些系统内容提供程序上进行操作需要权限。例如，应用程序必须拥有 READ_CONTACTS 权限才能在 Contacts Content Provider 上执行 READ 查询。 应用程序可能还需要获得许可才能从操作系统接收 Intent（即消息）。Intent 会通知应用程序事件，例如网络连接的更改，并且系统发送的某些 Intent 仅传递给具有适当权限的应用程序。此外，发送模拟系统 Intent 内容的 Intent 需要权限。

2.2 执行权限

我们描述了如何实现和保护系统 API，内容提供者和意图。 据我们所知，我们是第一个详细描述 Android 权限执行机制的人。

2.2.1 API

API 结构。Android API 框架由两部分组成：一个驻留在每个应用程序的虚拟机中的库，以及一个在系统进程中运行的 API 的实现。API 库以与其附带的应用程序相同的权限运行，而系统进程中的 API 实现不受限制。该库提供了与 API 实现进行交互的语法糖。库将在系统进程中将读取或更改全局电话状态的 API 调用代理到 API 实现。

API 调用分为三个步骤（图 1）。 首先，应用程序调用库中的公共 API。其次，该库调用一个私有接口，也在该库中。专用接口是 RPC 存根。 第三，RPC 存根通过系统进程启动 RPC 请求，该请求要求系统服务执行所需的操作。例如，如果应用程序调用 ClipboardManager.getText（），则该调用将中继到 IClipboard $ Stub $ Proxy，该代理将对系统进程的 ClipboardService 的调用代理。

应用程序可以使用 Java Reflection 访问所有 API 库的隐藏和私有类，方法和字段。某些私有接口没有任何相应的公共 API。但是，应用程序仍然可以使用反射来调用它们。这些非公共库方法仅供 Google 应用程序或框架本身使用，建议开发人员不要使用它们，因为它们可能在发行版之间更改或消失。但是，某些应用程序仍在使用它们。在系统进程中运行的 Java 代码位于单独的虚拟机中，因此不受反射的影响。

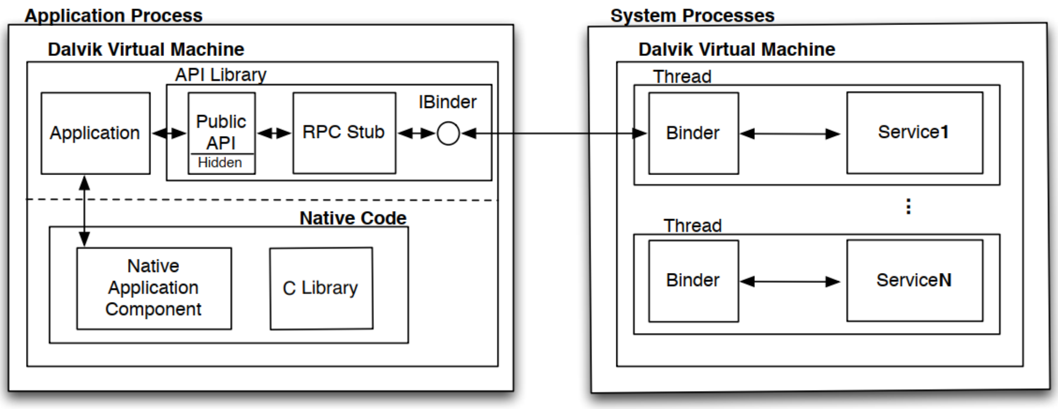

图 1 Android 平台的架构。权限检查在系统进程中进行。

权限。为了实施权限，系统的各个部分调用权限验证机制来检查给定的应用程序是否具有指定的权限。权限验证机制是作为受信任系统过程的一部分实现的，权限验证机制的调用分布在整个 API 中。调用 API 时，没有用于检查权限的集中策略。而是，调解取决于权限验证调用的正确放置。

权限检查放在系统过程中的 API 实现中。必要时，API 实现将调用权限验证机制以检查调用的应用程序是否具有必需的权限。在某些情况下，API 库也可以冗余地检查这些权限，但不能依靠这些检查：应用程序可以通过 RPC 存根与系统进程直接通信来规避它们。因此，权限检查不应在 API 库中进行。相反，系统进程中的 API 实现应调用权限验证机制。

少数权限是由 Unix 组强制执行的，而不是 Android 权限验证机制。尤其是，当安装具有 INTERNET，WRITE_EXTERNAL_STORAGE 或 BLUETOOTH 权限的应用程序时，会将其分配给可以访问相关套接字和文件的 Linux 组。 因此，Linux 内核针对这些权限实施访问控制策略。API 库（具有与应用程序相同的权限运行）因此可以直接在这些套接字和文件上运行，而无需在系统进程中调用 API 实现。

本地代码。应用程序除了 Java 代码外，还可以包括本机代码，但是本机代码仍然属于许可系统。尝试打开套接字或文件是由 Linux 权限引起的。本机代码无法直接与系统 API 通信，相反，应用程序必须创建 Java 包装器方法才能代表本机代

码调用 API。执行 API 调用时，将照常实施 Android 权限。权限将应用于内容提供商存储的所有资源。通过将权限与路径（例如，content：//a/b）相关联，也可以以更精细的粒度应用限制。例如，既存储公共注释又存储私有注释的内容提供者可能想要为整个内容提供者设置默认权限要求，但随后允许无限制地访问公共注释。可以类似地为某些路径设置额外的权限要求，仅当调用应用程序具有提供者的默认权限以及特定于路径的权限时，才可以访问这些路径下的数据。

2.2.2 Content Providers

系统内容提供程序是作为独立的应用程序安装的，与系统进程和 API 库分开安装。它们使用静态和动态权限检查进行保护，并使用与应用程序可用的相同机制来保护自己的内容提供程序。

静态声明为给定的内容提供程序分配了单独的读取和写入权限。默认情况下，这些权限将应用于内容提供者存储的所有资源。通过将权限与路径（例如，content：//a/b）相关联，也可以以更精细的粒度应用限制。例如，既存储公共注释又存储私有注释的内容提供者可能想要为整个内容提供者设置默认权限要求，但随后允许无限制地访问公共注释。可以类似地为某些路径设置额外的权限要求，仅当调用应用程序具有提供者的默认权限以及特定于路径的权限时，才可以访问这些路径下的数据。

2.2.3 Intents

Android 的 Intent 系统已广泛用于应用程序内部和应用程序之间的通信。为了防止应用程序模仿系统意图，Android 限制了谁可以发送某些意图。所有 Intent 都通过 ActivityManagerService（系统服务）发送，该服务强制执行此限制。 使用两种技术来限制系统意图的发送。某些 Intent 只能由具有适当权限的应用程序发送。其他系统意图只能由 UID 与系统匹配的进程发送。不管它们拥有什么权限，应用程序都无法发送后一种类别的 Intent，因为这些 Intent 必须源自系统进程。

应用程序可能还需要权限才能接收某些系统意图。操作系统使用标准的 Android 机制来限制其 Intent 收件人。应用程序（在这种情况下为 OS）可以通过向 Intent 附加权限要求来限制谁可以接收 Intent。

3. 权限测试方法

Android 的访问控制政策没有充分记录，但是该政策对于确定应用程序是否特权过多是必需的。为了解决这个缺点，我们根据经验确定了 Android 实施的访问控制策略。我们使用测试来构建权限图，该权限图标识 Android API 中每种方法所需的权限。特别是，我们修改了 Android 2.2 的权限验证机制，以记录发生的权限检查。然后，我们为 API 调用，内容提供者和意图生成了单元测试用例。 执行这些测试使我们能够观察与系统 API 交互所需的权限。一项核心挑战是建立单元测试，以获取所有平台资源的呼叫覆盖范围。

3.1 API

如 2.2.1 中所述，Android API 为应用程序提供了一个库，该库包含公共，私有和隐藏的类和方法。私有类的集合包括用于系统服务的 RPC 存根。1 所有这些类和方法都可以使用 Java 反射对应用程序进行访问，因此我们必须对其进行测试以识别权限检查。我们分三个阶段进行测试：针对反馈的测试；可定制的测试用例生成；手动验证。

3.1.1 定向反馈测试

在测试的第一阶段，我们使用了 Randoop，这是一种针对 Java 的，面向反馈的，面向对象的自动化测试生成器[20，22]。Randoop 将类列表作为输入，并从这些类中搜索可能的方法序列的空间。我们修改了 Randoop，使其可以作为 Android 应用程序运行并记录其调用的每个方法。我们对 Android 的修改记录了 Android 权限验证机制检查的每个权限，这使我们可以推断出哪个 API 调用触发了权限检查。

Randoop 搜索方法的空间以查找其返回值可用作其他方法的参数的方法。它维护有效的初始输入序列和参数的池，这些初始输入序列和参数最初是用原始值（例如 int 和 String）播种的。Randoop 通过从测试类的方法中随机选择一个方法，并从输入池中选择序列以填充该方法的参数来逐步构建测试序列。如果新序列是唯一的，则将执行它。成功完成的序列（即不产生异常）将添加到序列池中。Randoop 的目标是全面覆盖测试空间。与同类技术[4,9,21]不同，Randoop 不需要示例执行跟踪作为输入，从而使诸如 API 模糊测试之类的大规模测试更加易于管理。由于 Randoop 使用 Java 反射从提供的类列表中生成测试方法，因此它支持

测试非公共方法。我们修改了 Randoop 以测试输入类的嵌套类。

局限性。Randoop 的反馈制导空间探索受到它可以访问的对象和输入值的限制。如果 Randoop 无法在序列池中找到调用方法所需的正确类型的对象，则它将永远不会尝试调用该方法。Android API 太大，无法一次测试所有相互依赖的类，因此实际上在序列池中没有许多对象可用。我们通过一起测试相关类（例如 Account 和 AccountManager）并添加返回常见 Android 特定数据类型的种子序列来缓解此问题。不幸的是，这不足以为许多方法产生有效的输入参数。许多单例对象实例只能通过带有特定参数的 API 调用来创建；例如，可以通过使用参数“wifi”调用 android.content.Context.getSystemService（String）来获得 WifiManager 实例。我们通过使用特定的原始常量和序列扩展输入池来解决此问题。另外，一些 API 调用期望内存地址存储参数的特定值，而我们无法大规模解决这些问题。

Randoop 也不处理与输入参数无关的订购要求。在某些情况下，Android 期望方法以非常特定的顺序彼此优先。Randoop 仅生成序列链是为了为方法创建参数。它不能生成序列来满足非输入变量形式的依赖关系。进一步加重了这个问题，许多具有底层本机代码的 Android 方法如果被无序调用会产生分段错误，从而终止 Randoop 测试过程。

3.1.2 可定制的测试用例生成

Randoop 的以反馈为导向的测试方法未能涵盖某些类型的方法。发生这种情况时，无法手动编辑其测试序列以控制序列顺序或建立方法前提条件。为了解决这些限制并提高覆盖率，我们构建了自己的测试生成工具。我们的工具接受方法签名列表作为输入，并为每种方法输出至少一个单元测试。它维护一个默认输入参数池，这些默认参数可以传递给要调用的方法。如果一个参数有多个值，那么我们的工具将为该方法创建多个单元测试。（当同一方法的多个参数具有多个可能的值时，将组合创建测试。）如果找不到合适的参数，它也会使用空值生成测试。由于我们的工具将测试用例的生成与执行分开，因此人工测试人员可以编辑由我们的工具生成的测试序列。如果测试失败，我们将手动调整方法调用的顺序，引入额外的代码以满足方法的先决条件，或者为失败的测试添加新的参数。

我们的测试生成工具比 Randoop 需要更多的人工，但是对于快速覆盖 Randoop 无法正确调用的方法来说，它是有效的。与手动编写测试用例相比，监督和编辑由我们的工具生成的一组生成的测试用例的工作量仍然要少得多。我们在大规模 API 测试中的经验是，反馈定向测试难以调用的方法经常会出现问题。

当人类测试人员能够编辑失败的序列时，可以正确调用这些方法。

3.1.3 手动验证

测试的前两个阶段生成 API 中每种方法执行的权限检查的映射。但是，这些结果包含三种类型的不一致。首先，由异步 API 调用引起的权限检查有时会错误地与后续 API 调用相关联。其次，方法的权限要求可能取决于参数，在这种情况下，我们会对该方法进行间歇性或不同的权限检查。第三，权限检查可以取决于 API 调用的顺序。为了识别和解决这些不一致之处，我们手动验证了测试的前两个阶段生成的权限图的正确性。

我们使用了可定制的测试生成工具来创建测试，以确认与权限图中每个 API 方法相关联的权限。我们仔细测试了测试用例的顺序和参数，以确保我们将权限检查与异步 API 调用正确匹配，并确定了权限检查的条件。在确认潜在的异步或依赖于订单的 API 调用的权限时，我们还为相关类中最初没有与权限检查相关联的相关方法创建了确认测试用例。我们运行每个测试用例，无论它们是否具有必需的权限，以标识具有多个或可替换权限要求的 API 调用。如果测试用例在未经许可的情况下引发安全异常，但在获得许可后成功，那么我们知道被测方法的许可权映射是正确的。

*测试 Internet 权限。*应用程序可以通过 Android API 访问 Internet，但是其他包（例如 java.net 和 org.apache）也提供 Internet 访问。为了确定哪些方法需要访问 Internet，我们搜索了文档，并在 Internet 上搜索了建议访问 Internet 的所有方法。使用此列表，我们编写了测试用例，以确定哪些方法需要 INTERNET 权限。

3.2 Content Providers

我们的 Content Provider 测试应用程序对与 Android 系统和预安装的应用程序相关联的 Content Provider URI 执行查询、插入、更新和删除操作。我们从 android.provider 包中收集了 URI 列表，以确定要测试的内容提供商的核心集。我们还收集了在其他测试阶段发现的 Content Provider URI。对于每个 URI，我们尝试在没有任何权限的情况下执行每种类型的数据库操作。如果引发了安全异常，我们将记录所需的权限。我们添加并测试了权限的组合，以标识多个或可替代的权限要求。每个内容提供者都经过测试，直到不再为给定操作引发安全异常，这表明完成该操作所需的最小权限集。除了测试外，我们还检查了系统内容提供商的静态权限声明。

3.3 Intents

我们构建了一对应用程序来发送和接收 Intent。Android 文档没有提供可用系统 Intent 的单个完整列表，因此我们抓取了公共 API 来查找可能是 Intent 内容的字符串常量。我们在测试应用程序之间使用这些常量发送和接收 Intent。为了测试接收系统广播意图所需的权限，我们通过发送和接收短信，发送和接收电话，连接和断开 WiFi，连接和断开蓝牙设备等来触发系统广播。对于所有这些测试，我们记录是否进行了权限检查以及意图是否已成功交付或接收。

4. 权限映射结果

我们对 Android 应用程序平台的测试产生了一个权限图，该权限图将权限要求与 API 调用，内容提供者和意图相关联。在本节中，我们将讨论 API 的涵盖范围，将结果与 Android 官方文档进行比较，并介绍 Android API 和权限映射的特征。

4.1 覆盖范围

Android API 包含 1665 个类，共有 16732 个公共和私有方法。 通过两个阶段的测试，我们达到了 Android API 覆盖率的 85％。（如果在不产生异常的情况下执行该方法，则将其定义为被覆盖的方法；我们不衡量分支的覆盖范围。）Randoop 的初始方法覆盖率为 60％，分布在所有程序包中。 我们使用专有的测试生成工具补充了 Randoop 的覆盖范围，并通过至少一项权限检查来完成对属于类的方法的覆盖率接近 100％。

API 的未发现部分是由于本机调用和未在第一阶段进行权限检查的软件包的第二阶段测试所致。首先，当提供不正确的参数时，本机方法经常使应用程序崩溃，从而使其难以测试。 许多本机方法参数是整数，它们表示本机代码中对象的指针，因此很难提供正确的参数。 大约三分之一的未发现方法是本机调用。其次，我们决定对在 Randoop 测试阶段未显示权限检查的软件包省略补充测试。如果 Randoop 没有在包中触发至少一项权限检查，则我们不会在包中的类上添加更多测试。

4.2 与文档比较

清晰，完善的文档可促进正确使用权限和安全的编程习惯。 文档中的错误和遗漏可能导致错误的开发人员假设和特权。 Android 的权限文档受到限制，这很可能是因为它们缺乏集中式访问控制策略。我们的测试确定了 1259 个具有权限检查的 API 调用。 我们将此与 Android 2.2 文档进行了比较。

我们检索了 Android 2.2 文档，发现该文档指定了 78 种方法的权限要求。 该文档另外在几个类描述中列出了权限，但是尚不清楚类的哪些方法需要声明的权限。在文档中的 78 个受权限保护的 API 调用中，我们的测试表明 6 个 API 调用的文档不正确。我们不清楚文件或实现是否错误；如果文档正确，则这些差异可能是安全错误。

其中三个文档错误列出了与通过测试发现的权限不同的权限。在一个地方，文档声称一个 API 调用实际上受到较低权限的普通权限 GET_ACCOUNTS 的访问，并且受到危险权限 MANAGE_ACCOUNTS 的保护。另一个错误声称 API 调用需要 ACCESS_COARSE_UPDATES 权限，该权限不存在。结果，我们在 x6.2 中研究的 900 个应用程序中有 5 个请求此不存在的权限。第三个错误指出，当方法实际上受 BLUETOOTH_ADMIN 保护时，该方法受 BLUETOOTH 许可保护。

Permission	**Usage**
BLUETOOTH	85
B1UETOOTH_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

表 1：Android 的 10 个最常检查权限。

其他三个文档错误与具有多个权限要求的方法有关。 在一个错误中，文档声称一种方法需要一个许可，但是我们的测试表明需要两个许可。 对于最后两个错误，文档指出两个方法每个都需要一个许可权。 但是实际上，这两种方法都接受两种权限（即，它们是 ORs）。

4.3 权限描述

基于我们的权限映射，我们描述了如何在整个 API 中分布权限检查。

4.3.1 API 调用

我们检查了 Android API，看看有多少方法和类进行了权限检查。我们提供权限检查的数量、未使用的权限、层次权限、权限粒度和类特征。

权限检查数。我们通过权限检查识别了 1244 个 API 调用，占所有 API 方法（包括隐藏和私有方法）的 6.45%。其中，816 是普通 API 类的方法，428 是用于与系统服务通信的 RPC 存根的方法。我们在一个制造商添加的 API 的一个补充部分中另外识别了 15 个带有权限检查的 API 调用，总共 1259 个带有权限检查的 API 调用。表 1 提供了普通 API 最常见的检查权限的速率。

签名/系统权限。我们发现 12%的普通 API 调用使用签名/系统权限进行保护，35%的 RPC 存根使用签名/系统权限进行保护。这有效地限制了这些 API 调用对预安装应用程序的使用。

未使用的权限。我们发现有些权限是由平台定义的，但从未在 API 中使用过。例如，BRICK 许可从未被使用，尽管经常被引用为一个特别可怕的许可的例子。BRICK 权限的唯一使用是死代码，它不能对设备造成损害。我们的测试发现，134 个 Android 定义的权限中有 15 个未使用。对于在测试期间从未找到权限的每个情况，我们搜索源树以验证是否未使用该权限。在检查了几个设备之后，我们发现 HTC 和 Samsung 添加到 API 中以支持手机 4G 的自定义类使用了其中一个未使用的权限。

分级权限。许多权限的名称意味着它们之间存在层次关系。直观地说，我们期望更强大的权限应该可以替换与同一资源相关的较小权限。然而，我们没有发现有计划的等级制度的证据。我们的测试表明，BLUETOOTH_ADMIN 不能替代 BLUETOOTH，WRITE_CONTACTS 也不能替代 READ_CONTACTS。同样，不能使用更改 WIFI 状态来代替访问 WIFI 状态。

只有一对权限具有层次关系：ACCESS_COARSE_LOCATION 和

ACCESS_FINE_LOCATION。每个接受 COARSE 权限的方法也接受 FINE 作为替代。我们发现只有一个例外，这可能是一个 bug：电话管理员。TelephonyManager.listen() 支持 ACCESS_COARSE_LOCATION 或 READ_PHONE_STATE 权限，但不支持 ACCESS_FINE_LOCATION 权限。

权限粒度。如果将单个权限应用于不同的功能集，则请求对功能子集的权限的应用程序将对其余部分具有不必要的访问权限。Android 的目标是在可能的情况下，通过将功能分成多个权限来防止这种情况，并且他们的方法已经被证明有利于平台安全。作为一个案例，我们研究了蓝牙功能的划分，因为蓝牙权限是检查最多的权限。

我们发现这两个蓝牙权限应用于 6 个大类。它们分为更改状态的方法（蓝牙管理）和获取设备信息的方法（蓝牙）。BluetoothAdapter 类是使用 Bluetooth 权限的几个类之一，它适当地划分了大部分权限分配。然而，它有一些不一致之处。一个方法只返回信息，但需要 BLUETOOTH_ADMIN 权限，另一个方法更改状态，但同时需要这两个权限。这种类型的不一致可能会导致开发人员混淆哪些类型的操作需要哪些权限。

阶级特征。图 2 显示了每个类受保护的方法的百分比。我们最初预计，分布将是双峰的，大多数类完全或根本不受保护。然而，我们看到的是一系列的等级保护率。在这些类中，只有 8 个类需要权限来实例化对象，4 个类只需要对象构造函数的权限。

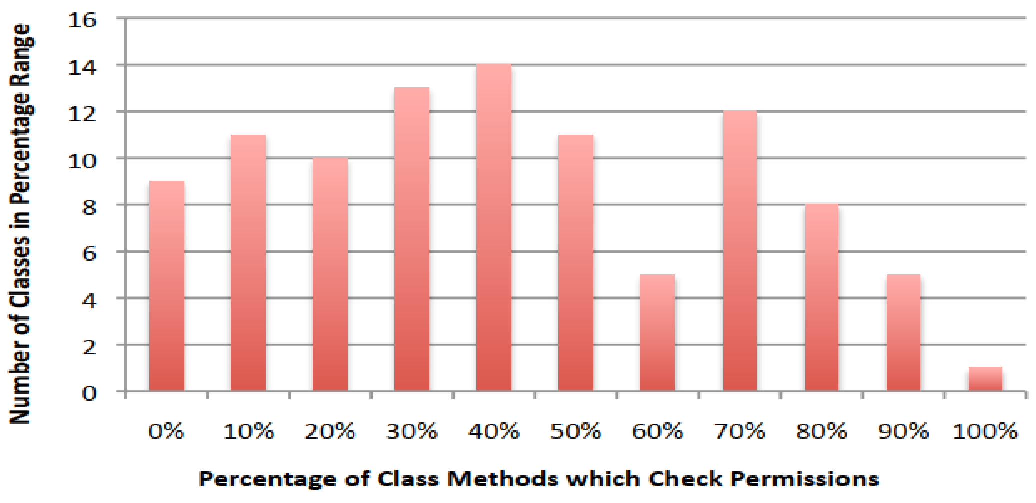

图 2：类的数量直方图，按需要权限的类的方法的百分比排序。所示数字代表范围，即 10%代表[10-20%]。我们只考虑至少有一个权限检查的类。

4.3.2 Content Providers and Intents

我们检查了内容提供者，以确定它们是否受权限保护。我们总共调查了 62 个内容提供商。我们发现有 18 个内容提供者不具有我们测试的任何方法（插入，查询，更新和删除）的权限。所有缺少权限的内容提供者都与 content：// media 内容 URI 相关联。

我们检查了意图通信，并测量了发送和接收意图是否需要权限。发送广播 Intent 时，非系统发送者禁止 62 个广播，发送 Intent 之前有 6 个需要许可，并且 2 个广播可以被广播，但系统接收者无法接收。广播接收方必须具有接收 23 个广播 Intent 的权限，其中有 14 个受蓝牙许可保护。发送意图以开始活动时，7 条意图消息需要权限。 启动服务时，2 个意图需要权限。

5. 应用分析工具

我们构建了一个静态分析工具 Stowaway，它可以分析 Android 应用程序并确定其可能需要的最大权限集。Stowaway 分析了应用程序对 API 调用，Content Providers 和 Intents 的使用，然后使用 x3 中内置的权限图来确定这些操作需要哪些权限。

Android 平台的已编译应用程序包括可在 Android Dalvik 虚拟机上运行的 Dalvik 可执行文件（DEX）文件。我们使用公开可用的 Dedexer 工具来分解应用程序 DEX。Stowaway 的每个阶段都将分解后的 DEX 作为输入。

5.1 API 调用

Stowaway rst 解析反汇编的 DEX 文件并识别对标准 API 方法的所有调用。Stowaway 跟踪从 An-droid 类继承方法的应用程序定义的类，因此我们可以区分应用程序定义的方法和 Android 定义的继承方法的调用。我们使用启发式来处理 Java 响应和两个异常权限。

反射。Java 反射是一个具有挑战性的问题。在 Java 中，可以使用 Java 反射地调用方法。lang.reflect.Method.invoke（）或 java.lang.reflect。Constructor.newInstance（）。偷偷摸摸跟踪哪些 Class 对象和方法名称传播到反射调用。它执行流程敏感的过程内静态分析，并通过过程间分析将其扩展到 2 个方法调用的深度。在每个方法主体中，它跟踪每个 String，StringBuilder，Class，Method，构造函数，Field 和 Object 的值。我们还跟踪这些类型的静态成员变量的状态。

我们确定将字符串和对象转换为 Class 类型的方法调用，以及将 Class 对象转换为 Method，Constructor 和 Fields 的方法调用。

我们还通过处理可能影响反射调用的方法和字段，将 Android 特定的启发式方法应用于反射解决方案。我们无法对整个 Android 和 Java API 的行为进行建模，但是我们会识别特殊情况。首先，Context.getSystemService（String）根据参数返回不同类型的对象。我们维护参数到返回对象类型的映射。其次，一些 API 类包含私有成员变量，这些变量包含对隐藏接口的引用。应用程序只能以反射方式访问这些成员变量，从而模糊了它们的类型信息。我们在成员变量及其类型之间创建了映射，并相应地传播了类型数据。如果应用程序在检索到某个成员变量后随后访问该方法，我们可以解析该成员变量的类型。

互联网。包含 WebView 的任何应用程序都必须具有 Internet 权限。WebView 是一个用户界面组件，它允许应用程序将网站嵌入其 UI 中。WebView 可以以编程方式实例化，也可以在 XML 文件中声明。Stowaway 标识 WebView 的程序化实例。它还会反编译应用程序 XML 文件并解析它们以检测 WebView 声明。

外部存储。 如果应用程序要访问 SD 卡上存储的文件，则必须具有 WRITE_EXTERNAL_STORAGE 权限。 该权限未出现在我们的权限图中，因为（1）完全使用 Linux 权限强制执行，并且（2）可以与从库中访问 SD 卡的任何文件操作或 API 调用相关联。 我们通过在应用程序的字符串文字和 XML 文件中搜索包含 sdcard 的字符串来处理此权限； 如果找到，则假定需要 WRITE_EXTERNAL_STORAGE。 此外，如果我们看到返回到 SD 卡路径的 API 调用（例如 Environment.getExternalStorageDirectory（）），则假定需要此权限。

5.2 Content Providers

通过对 URI 执行数据库操作来访问内容提供者。 Stowaway 收集了所有可用作内容提供者 URI 的字符串，并将这些字符串链接到内容提供者的权限要求。内容提供者 URI 可以通过两种方式获得：

1. 可以将一个字符串或一组字符串传递到返回 URI 的方法中。 例如，API 调用 android。 net.Uri.parse（“ content：// browser / bookmarks”）返回用于访问浏览器书签的 URI。 为了处理这种情况，Stowaway 查找了所有以 content：//。
2. 该 API 提供了包含公共 URI 常量的 Content Provider 帮助器类。例如，android.provider.Browser.BOOKMARKS_URI 的值为 content：// browser /

bookmarks。Stowaway 识别已知的 URI 常量，我们创建了一个从所有已知 URI 常量到其字符串值的映射。

我们工具的局限性在于我们无法判断应用程序使用 URI 执行哪些数据库操作。在 Content Provider 上执行操作的方式有很多，用户可以设置自己的查询字符串。为了解决这个问题，我们说应用程序可能需要与给定内容提供者 URI 上的任何操作相关的任何许可。这提供了使用特定内容提供程序可能需要的权限上限。

5.3 Intents

我们使用 ComDroid 检测需要权限的 Intent 的发送和接收。ComDroid 执行对流程敏感的过程内静态分析，并通过有限的过程间分析进行扩充，该过程在方法调用之后达到一个方法调用的深度。ComDroid 跟踪 Intent，寄存器，接收器（例如 sendBroadcast）和应用程序组件的状态。当实例化一个 Intent 对象，将其作为方法参数传递或作为返回值获取时，ComDroid 会跟踪从其源到其接收器的所有更改，并输出有关 Intent 的所有信息以及希望接收消息的所有组件。

Stowaway 获取 ComDroid 的输出，并针对每个发送的 Intent，检查是否需要许可才能发送该 Intent。对于注册应用程序要接收的每个 Intent，Stowaway 会检查是否需要许可才能接收 Intent。有时，ComDroid 无法识别消息或意图的接收器。为了减轻这些情况，Stowaway 在应用程序中所有字符串文字的列表中搜索受保护的 Intent。

6. 应用分析结果

我们将 Stowaway 应用到 940 个 Android 应用程序中，以识别过度特权的普遍性。具有不必要权限的应用程序违反了最小特权原则。特权过高损害了每个应用程序权限系统的好处：额外的权限不必要地限制用户随意接受危险的权限，并不必要地加剧应用程序漏洞。

Stowaway 计算应用程序可能需要的最大 Android 权限集。我们将该设置与应用程序实际请求的权限进行比较。如果应用程序请求更多权限，则它具有特权。我们全套的应用程序由 964 个 Android 2.2 应用程序组成。我们预留了 24 个随机选择的应用程序进行工具测试和培训，剩下 940 个用于分析。

6.1　手动分析

6.1.1　方法

我们从 940 个集合中随机选择了 40 个应用程序，并在其中运行了 Stowaway。斯托沃威将 18 项申请确定为特权过高的。然后，我们手动分析了每个特权特权警告，将其归因于工具错误（即误报）或开发人员错误。由于三类故障，我们寻找误报：

1. Stowaway 错过了需要权限的 API，Content Provider 或 Intent 操作。例如，当 Stowaway 无法解决反射调用的目标时，它会错过 API 调用。
2. Stowaway 可以正确识别 API，Content Provider 或 Intent 操作，但是我们的权限图缺少该平台资源的条目。
3. 该应用程序将 Intent 发送到其他某个应用程序，并且收件人仅接受具有特定许可权的发件人的 Intent。Stowaway 无法检测到这种情况，因为我们无法确定其他非系统应用程序的许可要求。

我们检查了 18 个应用程序的字节码，以查找这三种错误中的任何一种。 如果我们发现功能可能与 Stowaway 认为不必要的许可有关，则我们手动编写其他测试用例以确认我们的许可权地图的准确性。 我们通过检查应用程序是否将 Intents 发送到预安装的或知名的应用程序来研究第三种类型的错误。 当我们确定警告不是误报时，我们试图确定为什么开发人员添加了不必要的权限。

我们还通过在修改后的 Android 版本中运行应用程序（在发生权限检查时将其记录下来）并与之交互来分析过特权警告。无法在运行时测试所有应用程序。例如，某些应用程序依赖自我们下载它们以来已移动或更改的服务器端资源。我们能够以此方式测试 18 个应用程序中的 10 个。在每种情况下，运行时测试都确认了我们的代码审查的结果。

6.1.2　错误报告

Stowaway 确定了 40 个应用程序中的 18 个（占 45％）具有 42 个不必要的权限。我们的手动审查确定了 17 个应用程序（42：5％）具有特权，共有 39 个不必要的权限。 这代表 7％的误报率。

所有这三个错误警告都是由于我们权限图中的不完整所致。每个都是我们无法预期的特例。三个误报中的两个是由使用 Runtime.exec 来执行权限保护的 shell 命令的应用程序引起的。（例如，logcat 命令执行 READ_LOGS 权限检查。）第三个

误报是由以下应用程序引起的：嵌入使用 HTML5 地理位置的网站，该网站需要位置许可。我们针对这些情况编写了测试用例，并更新了权限图。

在这套应用程序中的 40 个应用程序中，有 4 个包含至少一个反射调用，我们的静态分析工具无法解决或关闭这些反射调用。其中 2 个享有特权。这意味着具有至少一个未解决的反射调用的应用程序中有 50％的特权过高，而其他应用程序的特权为 42％。但是，样本量 4 太小，无法得出结论。我们调查了未解决的反省电话，并且不相信它们会导致误报。

6.2 自动化分析

我们在 900 个 Android 应用程序上运行了 Stowaway。总体而言，Stowaway 确定 323 个应用程序（占 35：8％）具有不必要的权限。Stowaway 无法解决某些应用程序的反射调用，这可能导致这些应用程序中更高的误报率。因此，我们将与其他应用程序分开讨论具有未解决的反射调用的应用程序。

6.2.1 完全处理反射的应用

Stowaway 能够处理 900 个应用程序中的 795 个的所有反射调用，这意味着它应该已经标识了那些应用程序的所有 API 访问。Stowaway 为 795 个应用程序中的 32：7％生成了特权特权警告。表 2 显示了这些应用程序中 10 种最常见的不必要权限。

56％的超特权应用程序具有 1 个额外的权限，而 94％的应用程序具有 4 个或更少的额外权限。尽管三分之一的应用程序享有特权，但每个应用程序的过度特权程度低表明开发人员正在尝试添加正确的权限，而不是任意请求大量不需要的权限。这支持了 Android 等安装时权限系统的潜在效力。

我们相信，Stowaway 对于这些应用程序应产生与在 x6.1 中评估的 40 个一组相同的假阳性率。 如果我们假设手动分析得出的 7％误报率适用于这些结果，那么 795 项申请中的 30.4％确实享有特权。 实际上，由于以下原因，应用程序实际上可能比我们的工具所显示的特权更多。

无法访问的代码。 偷渡者不执行无效代码消除； 消除 Android 应用程序的死代码需要考虑到独特的 Android 生命周期和应用程序入口点。 此外，我们对 Content Provider 操作（x5.2）的过高估计可能会忽略一些特权。 我们没有量化 Stowaway 的误报率，我们将消除无效代码和改进 Content Provider 字符串跟踪留给以后的工作。

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%

表 2：10 种最常见的不必要权限以及请求它们的过度特权应用程序的百分比。

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

表 3：按反射状态，Stowaway 发出超权限警告的速率。

6.2.2 Java 反射的挑战

反射通常在 Android 应用程序中使用。在 900 个应用程序中，有 545 个（占 61％）使用 Java 反射进行 API 调用。我们发现反射被用于许多目的，例如反序列化 JSON 和 XML，调用隐藏或私有 API 调用以及处理名称在版本之间变化的 API 类。反射的普遍性表明，即使 Android 静态分析工具并非旨在用于混淆代码或恶意代码，对于 Android 静态分析工具来说也很重要。

在 59％的使用反射的应用程序中，Stowaway 能够完全解决反射呼叫的目标。我们使用两种技术处理了 117 个应用程序：消除了已知在应用程序中定义了反射调用的目标类的故障，以及手动检查和处理了 21 个非常流行的库中的故障。这使我们有 105 个具有反射功能的应用程序 Stowaway 无法解决或驳回的呼叫，占 900 项申请的 12％。

Stowaway 将 105 个应用程序中的 53：3％确定为特权级别较高。 表 3 将其

与没有未处理反射的应用程序发出警告的比率进行了比较。 对此差异有两种可能的解释：Stowaway 在具有未解决的反射调用的应用程序中可能具有较高的误报率，或者以复杂方式使用 Java 反射的应用程序可能由于相关的特性而具有较高的实际超特权率。

我们怀疑这两个因素在未处理反射呼叫的应用程序中较高的特权特权警告率中都起作用。尽管我们的人工审查（x6.1）并未发现反射失败会导致误报，但随后对其他应用程序的审查却发现了一些由反射引起的错误警告。另一方面，开发人员错误可能随着与复杂的反射调用相关的复杂性而增加。

在 Android 应用程序中提高反射调用的分辨率是一个重要的开放问题。出现基于非静态环境变量的方法名称，直接生成 Dalvik 字节码，带有两个引用相同位置的指针的数组或存储在哈希表中的 Method 和 Class 对象时，Stowaway 的反射分析将失败。Stowaway 的方法主要是线性遍历，也遇到非线性控制流的问题，例如跳跃； 我们只处理出现在方法末尾的简单 getos。我们还观察到了几个在一组类或方法上进行迭代的应用程序，测试每个元素以确定反射性调用哪个元素。如果测试了多个比较值并且在该块中未使用任何比较值，则 Stowaway 仅跟踪该块之外的最后一个比较值；该值可以为空。将来的工作也许可以使用动态分析来解决其中的一些问题。

6.3 常见的开发人员错误

在某些情况下，我们能够确定为什么开发人员要求不必要的权限。在这里，我们考虑了手动审核的 40 个应用程序和自动化分析的 795 个完全处理的应用程序中不同类型的开发人员错误的普遍性。

权限名称。开发人员有时会要求发出听起来与他们的应用程序功能相关的名称的权限，即使这些权限不是必需的。 例如，我们手动审核中的一个应用程序不必要地请求 MOUNT_UNMOUNT_FILESYSTEMS 权限来接收 android.intent.action.MEDIA_MOUNTED Intent。作为另一个示例，ACCESS_NETWORK_STATE 和 ACCESS_WIFI_STATE 权限具有相似的名称，但是不同的类需要它们。开发人员经常成对地请求它们，即使只需要一个。在不必要地请求网络许可的应用程序中，有 32％合法地需要 WiFi 许可。在不必要地请求 WiFi 许可的应用程序中，有 71％合法地需要网络许可。

代表。一个应用程序可以向另一个代理应用程序发送一个 Intent，要求代理执行操作。如果代理进行权限保护的 API 调用，则代理需要权限。但是，Intent 的发

送者没有。我们注意到，有许多应用程序实例要求他们代表
代表执行操作的权限。例如，一个应用程序要求 Android Market 安装另一个应用程序。发件人询问 INSTALL_PACKAGES，因为 Market 应用程序进行安装，所以不需要。

我们发现这种错误的广泛证据。在不必要地请求 CAMERA 许可的应用程序中，有 81％会发送一个 Intent 来打开默认的 Camera 应用程序进行拍照。不必要地请求 INTERNET 的应用程序中有 82％发送了在浏览器中打开 URL 的 Intent。同样，不必要地请求 CALL_PHONE 的应用程序中有 44％会将 Intent 发送到默认的 Phone Dialer 应用程序。

相关方法。 如图 2 所示，大多数类包含权限保护和不受保护的方法的混合。我们发现应用程序使用不受保护的方法，但请求同一类中其他方法所需的权限。例如 android.provider。Settings.Secure 是 API 中用于访问 Settings Content Provider 的便捷类。该类包括设置器和获取器。设置器需要 WRITE_SETTINGS 权限，但获取器则不需要。我们手动检查的两个应用程序仅使用吸气剂，但请求 WRITE_SETTINGS 权限。

复制和粘贴。热门留言板包含 Android 代码段和有关权限要求的建议。有时，此信息不准确，复制该信息的开发人员将使他们的应用程序拥有特权。例如，我们手动检查的应用程序之一注册以接收 android.net.wifi.STATE_CHANGE 意向并请求 ACCESS_WIFI_STATE 权限。截至 2011 年 5 月，该 Intent 的第三高 Google 搜索结果包含错误的主张，即它需要该许可权。

不建议使用的权限。 在旧版 Android 中，可能需要 Android 2.2 中不必要的权限。因此，旧的或向后兼容的应用程序可能具有额外的权限。但是，开发人员也可能不小心使用了这些权限，因为他们已阅读了过时的资料。8％的特权应用程序请求 ACCESS_GPS 或 ACCESS_LOCATION，它们在 2008 年已弃用。在这些应用程序中，除一个之外，所有应用程序均指定其受支持的最低 API 版本高于该版本。
包含这些权限的最新版本。

测试工件。开发人员可能会在测试过程中添加一个权限，然后在删除测试代码时忘记将其删除。例如，ACCESS_MOCK_LOCATION 通常仅用于测试，但可以在已发布的应用程序中找到。数据集中所有不必要地包含 ACCESS_MOCK_LOCATION 权

限的应用程序也都包含真实位置权限。

签名/系统权限。我们发现 9%的特权过多的应用程序请求不需要的 Signature 或 SignatureOrSystem 权限。Android 的标准版本将默默拒绝将那些权限授予未由设备制造商签名的应用程序。权限被错误地请求，或者开发人员发现相关代码在标准手机上不起作用后删除了相关代码。

我们可以将许多特权实例归因于开发人员对许可系统的困惑。 可以通过改进的 API 文档解决权限名称，相关方法，代理和不赞成使用的权限的混淆。为避免由于相关方法而产生的超权限，建议您按方法（而不是按类）列出权限要求。通过阐明权限和预安装的系统应用程序之间的关系，可以减少对代理人的混淆。

尽管我们可以将许多不必要的权限归因于错误，但仍有一些开发人员有意地请求额外的权限。 激励开发人员要求不必要的权限，因为如果应用程序的更新版本请求更多权限，则应用程序将不会收到自动更新。

7. 相关工作

Android 权限。以前对 Android 应用程序的研究在了解权限使用方面受到了限制。我们的权限图可用于大大增加应用程序分析的范围。Enck 等。将 Fortify 的 Java 静态分析工具应用于反编译的应用程序；他们研究 API 的用法。但是，他们仅限于研究应用程序对少量权限和 API 调用的使用。在最近的一项研究中，Felt 等人。手动将一小部分 Android 应用程序分类为是否特权过度，但是它们受到 Android 文档的限制。麒麟在安装过程中读取应用程序许可要求，并根据一组安全规则进行检查。它们仅依赖于开发人员权限请求，而不是检查应用程序是否使用权限或如何使用权限。Barrera 等。检查 1100 个 Android 应用程序的权限要求，并使用自组织映射来可视化具有相似特征的应用程序中使用了哪些权限。他们的工作还依赖于应用程序请求的权限。

维达斯等。提供对应用程序源代码执行特权分析的工具。通过使用我们的权限图可以改进他们的工具；他们基于有限的 Android 文档。我们的静态分析工具还可以执行更复杂的应用程序分析。与他们的 Eclipse 插件不同，Stowaway 尝试处理反射调用，Content Providers 和 Intent。

在并发工作中，Gibler 等。将静态分析应用于 Android API 以查找权限检查。它们的权限映射包括系统进程内无法跨 RPC 边界访问的内部方法，我们将其排

除在外，因为应用程序无法访问它们。与我们的动态方法不同，它们的静态分析可能会误报，会丢失本机代码中的权限检查，并且会丢失特定于 Android 的控制流。

Java 测试。Randoop 不是唯一的 Java 单元测试生成工具。诸如 Eclat，Palulu 和 JCrasher 之类的工具的工作原理相似，但需要示例执行作为输入。考虑到 Android API 的大小，构建这样的示例执行将是一个挑战。 增强的 JUnit 通过将构造函数链接到某个固定深度来生成测试。但是，它不使用子类型提供实例，而是依靠字节码作为输入。Korat 需要正式的方法规范规范作为输入，这对于 Android API 的事后测试是不可行的。

Java 反射。处理 Java 反射对于开发完善的程序分析是必需的。但是，解决反射性呼叫是开放研究的领域。Livshits 等。创建了一种静态算法，通过跟踪传递给反射的字符串常数来近似反射目标。当反射调用依赖于用户输入或环境变量时，他们的方法就不够用了。我们使用相同的方法并遭受相同的限制。他们通过开发人员注释来改善结果，这对于我们的领域而言并不可行。一种更高级的技术将静态分析与有关 Java 程序环境的信息相结合，以解决反射问题。但是，仅当程序在与原始评估相同的环境中执行时，它们的结果才是正确的。即使进行了修改，它们也只能解决 Java 1.4 API 中 74％的反射调用。我们不主张在解决 Java 反射方面改进现有技术；相反，我们专注于特定于域的启发式方法，以了解如何在 Android 应用程序中使用反射。我们是第一个讨论 Android 应用程序反射的人。

8. 结论

在本文中，我们开发了用于检测 Android 应用程序中超权限的工具。我们将自动化测试技术应用于 Android 2.2，以确定调用每种 API 方法所需的权限。我们的工具 Stowaway 会生成应用程序所需的最大权限集，并将它们与实际请求的权限集进行比较。目前，Stowaway 无法处理一些复杂的反射调用，并且我们将 Java 反射确定为 Android 静态分析工具的重要开放问题。我们将 Stowaway 应用到 940 个 Android 应用程序中，发现其中约有三分之一具有特权。我们的结果表明，应用程序通常仅受到少数权限的特权，而许多额外的权限可归因于开发人员的困惑。这表明开发人员试图为其应用程序获取最少的特权，但由于 API 文档错误和缺乏开发人员的了解而未能达到要求。

致谢

感谢 Royce Cheng-Yue 和 Kathryn Lingel 在测试 API 和内容提供程序方面的帮助。 NSF 赠款 CCF-0424422、0311808、0832943、0444482、0942694，来自 Google 的礼物以及 AFOSR 赠款 FA9550-08-1-0352 下的 MURI 程序部分支持了这项工作。该材料还基于 NSF 研究生研究奖学金的支持。这里表达的任何观点，发现，结论或建议均为作者的观点，不一定反映 NSF 的观点。