PScout: Analyzing the Android Permission Specification

Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang and David Lie
Dept. of Electrical and Computer Engineering
University of Toronto, Canada

ABSTRACT

Modern smartphone operating systems (OSs) have been developed with a greater emphasis on security and protecting privacy. One of the mechanisms these systems use to protect users is a permission system, which requires developers to declare what sensitive resources their applications will use, has users agree with this request when they install the application and constrains the application to the requested resources during runtime. As these permission systems become more common, questions have risen about their design and implementation. In this paper, we perform an analysis of the permission system of the Android smartphone OS in an attempt to begin answering some of these questions. Because the documentation of Android's permission system is incomplete and because we wanted to be able to analyze several versions of Android, we developed PScout, a tool that extracts the permission specification from the Android OS source code using static analysis. PScout overcomes several challenges, such as scalability due to Android's 3.4 million line code base, accounting for permission enforcement across processes due to Android's use of IPC, and abstracting Android's diverse permission checking mechanisms into a single primitive for analysis.

We use PScout to analyze 4 versions of Android spanning version 2.2 up to the recently released Android 4.0. Our main findings are that while Android has over 75 permissions, there is little redundancy in the permission specification. However, if applications could be constrained to only use documented APIs, then about 22% of the non-system permissions are actually unnecessary. Finally, we find that a trade-off exists between enabling least-privilege security with fine-grained permissions and maintaining stability of the permission specification as the Android OS evolves.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls, Information flow controls

General Terms

Design, Security

Keywords

Android, Permissions, Smartphone

1. INTRODUCTION

One of the distinguishing and desirable characteristics of smartphones over traditional feature phones is that they are capable of running applications written by third party developers. This capability, combined with their highly personal nature, has raised concerns about the threat that smartphones pose to the privacy and security of their users. To address these concerns, many smartphone operating systems (OS) implement some sort of permission system to control access by third party applications to sensitive resources, such as the user's contact list or the phone's microphone [1][1].

Because of the rapidly growing number of smartphone users and the wide use of permission systems on these platforms, it is important that we gain a better understanding of the implementation and design of smartphone permission systems. In this paper, we take some first steps towards answering some of the key questions that have arisen about the permission system of Android, which is the most widely deployed smartphone OS at the time of writing. In particular:

- Is the plethora of permissions offered by Android (79 permissions available to third party applications as of Android 4.0) useful or would a smaller number of coarser permissions provide an equal amount of protection [1, 3]?
- Android has many undocumented APIs that are used by third party applications [14]. Do many of these undocumented APIs require permissions and how commonly do third party applications use such undocumented APIs?
- The large number of permissions and APIs in Android suggests that the permission specification for Android will be very complicated. How complex is the specification and how heavily interconnected are different API calls with different permissions?

[1]The notable exception is Apple's iOS, which relies on manual vetting of applications. Manual vetting may protect against malicious applications, but does not prevent privilege escalation if a benign application is compromised.

- Android has gone through several major revisions and now has one of the more mature permission systems among smartphone OSs. How has it evolved over time and what might newer smartphone OSs such as Windows Phone 7 and Boot to Gecko (B2G) learn from this?

To perform such a study, we need a specification for the Android permission system that lists the permission requirements for every API call. More importantly, we also need permission specifications for more than one version of Android. Unfortunately, Google does not provide such a specification as the Android permission documentation is very incomplete [21]. The Stowaway project currently lists the permissions required for over 1,200 documented and undocumented API calls for Android 2.2 [14]. To produce valid results, Stowaway requires manually specifying some API input arguments and sequences, which unfortunately, must be re-specified for other versions of Android. Using a slightly different permission specification extraction tool for each Android version will introduce noise into any comparisons drawn across versions. The amount of manual effort required makes it infeasible to reuse Stowaway to generate specifications for multiple Android versions. In addition, because Stowaway relies on feedback directed API fuzzing to extract a specification, its specification is incomplete – Stowaway can only exercise API calls it can find, and of those, it is only able to successfully execute 85% of them.

To perform our study, we needed a method of extracting the permission specification from Android that is applicable to any Android version without modification. The tool should be able to capture permission requirement for every documented and undocumented API calls. To accomplish this, we implemented a static analysis tool called PScout (short for Permission Scout), which performs a reachability analysis between API calls and permission checks to produce a specification that lists the permissions that every Android API call requires. Because PScout examines the entire source code implementing the Android API, it can identify every Android API that can be called and produce a mapping to the permissions that the API call may need. An added benefit of using reachability is that PScout also extracts information about the execution path an API takes to reach a permission check, which we also use in our analysis. PScout relies only on non-version specific Android functions and components such as Binder, Intents, Content Providers and permission check functions, allowing it be applied without modification to any version of Android.

We make the following contributions in this paper:

- We design and implement PScout, a version-independent static analysis tool that extracts a permission specification from Android that is more complete than existing specifications. PScout's analysis finds over 17 thousand mappings between API calls and permissions, which is considerably more than that found by Stowaway.

- We measure the amount of imprecision introduced by PScout's scalable static analysis using an extensive evaluation methodology that compares against application developers. However, since developers themselves make errors in specifying permissions, we follow up with an automated application UI fuzzer that validates the permission mapping by exhaustively trying to find flaws in it.

- We analyze the permission system of Android 4.0 as well as how it has changed across 4 versions ranging from Android 2.2 to 4.0. Our analysis shows that while there is little redundancy in the permission specification, about 22% of the non-system permissions can be hidden if applications only use documented APIs. We also find that while the Android permission specification is very broad, it is not very heavily interconnected. More than 80% of API calls that may require permissions check for at most one permission and 75% of permissions are checked by fewer than 20 API calls. Finally, we find that across versions the amount of protection from permissions has remained relatively constant to the increase in code size and functionality of Android.

We begin with background on the Android OS in Section 2. Readers familiar with Android may wish to start at Section 3, which describes PScout's static analysis and method for extracting permission specification. Section 4 evaluates the completeness and soundness of PScout's extracted permission specification. Section 5 describes our analysis of the permission specifications of 4 Android versions. We discuss related work in Section 6 and conclude in Section 7.

2. BACKGROUND

Android is a middleware OS implemented above a customized Linux kernel. Android applications and system services are all implemented in Java and make cross-domain calls via a set of documented APIs exported by the Android system services. We define the "Android Framework", as the set of Java libraries and system services that implement these Android APIs. However, rather than restricting themselves to the documented Android APIs, third party developers may use Java reflection or examine the source code of Android to obtain references to any Java method and use them in their code. We and others [14] have found that applications do in fact use undocumented APIs, so we include undocumented APIs in our analysis as well.

Android is characterized by the heavy use of IPC both within applications and as a means of communication between applications and components of the OS. In addition to traditional IPC mechanisms like shared memory and sockets, Android introduces two Android-specific IPC mechanisms. The first are Intents, which is a uni-direction message with an arbitrary action string that can be broadcasted to all applications or sent to a specific application. Permissions may be used to both restrict who may receive an intent sent by an application, or to restrict who may send intents to a particular application.

Android's second IPC mechanism is Binder, which implements Remote Procedure Calls (RPC). Once a remote interface is properly defined in an Android Interface Definition Language (AIDL) file, it can be called just like any other local method. System services accept asynchronous Binder communication through the Message Handler class, which wraps the Binder interface.

The Android framework also includes a set of system components. One type of component is called a content provider, which implements databases that provide persistent data storage for other application components. Applications specify which content provider they are addressing using Uniform Resource Identifiers (URI) that start with `content://`. Content providers may require that applications hold certain permissions to access them.

To access sensitive resources, users must grant the requested permissions to applications. There are two major types of permissions in Android: "Signature or system permissions", which are only available to privileged services and content providers, and regular permissions, which are available to all applications. Since third party applications can only request regular permissions, we focus on extracting a specification for only the non-system permissions in our analysis. Android developers manually declare all required permissions in the Android Manifest file (AndroidManifest.xml). During installation, the Package Manager Service, parses the Manifest file into a permission list. Whenever the application tries to access a privileged system resource, the Android framework will query Package Manager Service to check if the application has the necessary permission to do so.

3. PSCOUT DESIGN AND IMPLEMENTATION

One of the challenges for PScout is the sheer size of the Android framework. Table 1 (in Section 5) gives some statistics across the different Android versions to give the reader an idea of the scale of the framework. Because of this, PScout's design is oriented towards making it scalable, with only selective use of more detailed analysis to minimize loss of precision.

PScout produces a permission specification that is a set of mappings between elements in the set of API calls and the set of permissions that third party applications may request. Because the mapping is produced between an API call and a permission, it is necessarily an approximation – an API may not require a particular permission in every context it is invoked. In such cases, PScout returns a conservative result – a mapping between an API call and a permission is included in the specification if the permission is required on some execution of the API. In cases where PScout finds that an API call may need more than one permission, it assumes all permissions will be required when in fact only a subset of those permissions may be needed on any particular invocation of the API.

PScout leverages the Soot [20] Java bytecode analysis framework to perform static analysis. The extraction of the permission specification from the Android framework has three phases. First, PScout identifies all the permission checks in the Android framework and labels them with the permission that is being checked. Then, it builds a call graph over the entire Android framework including IPCs and RPCs. Finally, it performs a backwards reachability traversal over the graph to identify all API calls that could reach a particular permission check. In some cases, the reachability traversal may find additional permission checks that were missed in the first phase. As a result, the reachability analysis is repeated until the number of permission checks converges. Figure 1 gives a high level summary of the analysis flow in PScout and each phase is discussed in more detail in the following Sections 3.1, 3.2 and 3.3.

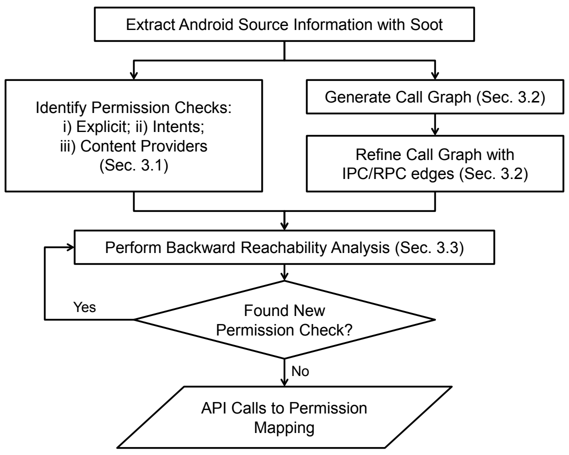
Extract Android Source Information with Soot
Identify Permission Checks: i) Explicit; ii) Intents; iii) Content Providers (Sec. 3.1)
Generate Call Graph (Sec. 3.2)
Refine Call Graph with IPC/RPC edges (Sec. 3.2)
Perform Backward Reachability Analysis (Sec. 3.3)
Yes
Found New Permission Check?
No
API Calls to Permission Mapping

Figure 1: PScout Analysis Flow.

PScout extracts a permission specification from the Android 4.0 Framework on an Intel Core 2 Duo 2.53 GHz CPU with 4 GB of memory in 33 hours. The vast majority of time is spent in two iterations of using Soot to extract information needed by PScout from the bytecode. Since each bytecode file is analyzed independently, the total Soot analysis time scales linearly with the number of classes and each file takes about 4 seconds to process. The rest of PScout's analysis completes within 30 minutes.

We restrict PScout's analysis to only the non-system permissions implemented in the Android Open Source Project (AOSP) that third party applications may hold. PScout's analysis only runs on the Android Framework where the vast majority of permissions are checked in Java. PScout cannot find permission mappings for the four permissions that are checked in native C/C++ code. As a result, we handle these permissions by manually inspecting the source code where these permission strings are used. In addition to being enforced in the Java framework, some permissions are also enforced by the kernel using Group IDs assigned at installation time. PScout only captures the enforcement done in the framework for these permissions and currently does not capture the enforcement done in the kernel.

3.1 Permission Check Identification

Android has three types of operations that will succeed or fail depending on the permissions held by an application: (1) an explicit call to `checkPermission` function that checks for the presence of a permission, (2) methods involving Intents and (3) methods involving Content Providers. We abstract each of these into a *permission check* that indicates that a certain permission must be held at that point by the application. We describe how we abstract each of Android's permission check mechanisms below.

Explicit Functions: Permissions in Android appear as string literals in the framework source code and values of the permissions strings are documented by Google. The strings are passed to the `checkPermission` function along with the application's user ID (UID) and the function checks if an application with the UID holds the specified permission.

The first step of identifying this type of permission check is to find all instances of permission string literals in the framework source. In most cases, these are passed directly to a `checkPermission` function, which PScout then abstracts as a permission check. However, in some cases, the permission string is passed to a function that is not a `checkPermission` function. In these cases, PScout must determine if this string is eventually passed to a `checkPermission` function or not. To do this, PScout uses Soot's def-use analysis to find all the uses of the string. If any of the uses is a `checkPermission` function or a permission wrapper function, PScout abstracts the function where the string literal appears as a permission check. If no `checkPermission` function appears in any of the uses of the literal, then PScout checks if it is an Intent function, which we discuss below.

Finally, some functions use a permission string without invoking a `checkPermission` function or an Intent function. There were a few instances of such functions which we inspected manually. We found that none of these are permission checks. For example, a permission string is used to query whether the permission is listed for a particular service; two permissions are implicitly added to applications developed for older versions of the Android SDK for compatibility and a permission string is used to set an internal flag in a class.

Intents: Sending and receiving of Intents may require permissions. This requirement can be expressed in two ways. First, a requirement to hold a permission to send or receive an Intent can be specified in the Manifest file. PScout extracts the Intent action strings associated with each permission from the Manifest file. Second, permission to send or receive an Intent can be expressed programmatically when the method to send or receive an Intent is called. To send an Intent, an application may call `sendBroadcast` with an optional permission string that specifies that the receiver of the Intent must hold that permission. Similarly, to receive Intents, an application may call `registerReceiver` with an optional permission string that specifies that the sender of the Intent must hold that permission. Using Soot, an intra-procedural backward flow analysis is performed on the methods that call these two APIs and their variants to extract the permission parameter and the action string assigned to the Intent parameter. By extracting information from the Manifest file and invocations of `sendBroadcast` and `registerReceiver`, PScout builds a global mapping between permissions and the Intent action strings.

Since this mapping tells PScout which Intent action strings require permissions to send or receive, PScout abstracts any send or registration to receive such an Intent as a permission check. The type of permission that is being checked is computed by translating the action string of the Intent being sent or received into an Android permission.

Content Providers: Methods that implicitly access a content provider protected by a permission are categorized as content provider permission checks. To access a content provider, an URI object designating the recipient content provider is passed to a `ContentResolver` class, which then provides a reference to the content provider targeted by the URI object.

Our handling of content provider permission checks is similar to the way we handle Intent permission checks. PScout first constructs a mapping of content provider URIs to permission strings. Each content provider declares the permissions required to read and write to it in its Manifest file, so PScout parses the Manifest file to extract this information. It is also possible that the content provider programmatically checks the permissions of the caller. In these cases, PScout uses all previous identified permission checks and performs a backward reachability analysis to see if any content provider access methods can reach one of those permission checks. If so, PScout then extracts the URI associated with the content provider and maps that to the permission being checked. When this phase is completed, PScout has a mapping between content provider URIs and permissions.

Finally, to identify all actions on content providers that require permissions, PScout identifies all instances where a content provider URI is passed to a content provider access method. If the URI has a permission associated with it in the mapping, the access method is abstracted into a permission check. As with the `checkPermission` functions, the URI may be manipulated symbolically and passed through several variables before being passed to a content provider method. Thus, PScout again uses Soot's backward flow analysis to determine what the content of the URI parameter is when the content provider access method is called.

3.2 Call Graph Generation

To generate the global call graph, PScout first generates a set of standard call graphs for each Android framework component. PScout then combines these into a single call graph by adding edges for RPCs that occur between Android components, and then further incorporates Message Handler IPCs into the graph. Message Handlers often have high fan-in/fan-out functions so PScout refines them using flow-sensitive analysis as they are added to the call graph to reduce the number of infeasible paths.

PScout begins with all classes in the Android framework including application and system level classes. In this phase, PScout only uses Soot to extract the class hierarchy for each class, a list of methods invoked by each method and the def-use information mentioned above. PScout then generates its call graph from this information using Class Hierarchy Analysis [10], which uses the following rules: (1) a virtual call to a class can potentially targets all its subclasses; (2) an interface call can be resolved to call any class that implements the interface and its subinterfaces; and (3) the target method of each subclass is the closest ancestor that implements the method.

Next, PScout adds execution flows across IPCs and RPCs into its call graph. IPCs and RPCs tend to use generic functions to marshal and send arguments across process boundaries, so we must refine with flow-sensitive analysis to avoid too much imprecision. In Android, all RPCs flow through Android's Binder class. A naïve call graph analysis of the Binder class would give a result that any RPC stub could flow to any RPC handler and create many infeasible paths. To refine these edges, PScout takes advantage of the fact that Android uses the Android Interface Definition Language (AIDL) to automatically generate stubs and interfaces for all RPCs. By parsing the AIDL files that describe each RPC interface, PScout can add edges between the corresponding interface and stub functions of each RPC.

PScout also adds Message Handlers IPCs to the call graph. Similar to the Binder class described above, sending of Message Handler IPCs uses a generic Handler class in Android which would also result in many infeasible paths if handled naïvely. To send messages to a Message Handler class, the senders must obtain a reference to the Message Handler object. PScout performs static analysis to determine the class

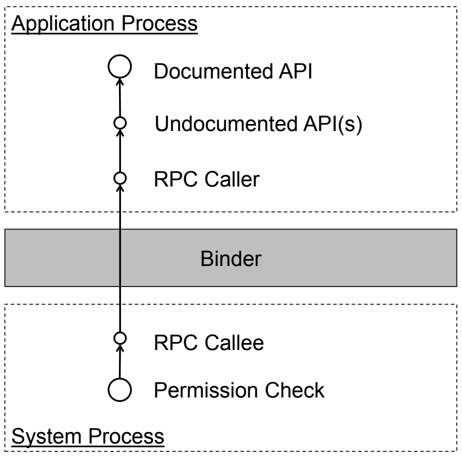

Figure 2: A reachability path starts from a permission check in the system processes. Not all permissions can reach a documented API, meaning that some permission checks can only be reached from undocumented APIs.

name of the Message Handler object, which can be used to link the method sending the message to the method receiving the message.

In addition to linking the senders and receivers of messages, PScout also performs simple flow-sensitive analysis for these IPCs. A large number of message handlers will use an input passed to them by the sender to index into a table of handler functions and call the indexed function. PScout performs intra-procedural backward flow analysis on the sender to obtain all possible values the index may take. The call graph is then refined to only include edges from that call site to the specific functions that can be invoked by the receiver in its lookup table.

3.3 Reachability Analysis

Finally, PScout performs a backward reachability analysis starting from the permission checks on the generated call graph. PScout creates a mapping for every method that can be reached via a path from a permission check. However, not every method is an API that can be called by a third party application – permission protected resources typically reside in system services, which can only be reached by third party applications via an RPC or through IPCs using intents. Thus, any path from a permssion check that does not cross a process boundary is filtered from PScout's specification. Figure 2 shows an example of a path identified by the reachability analysis.

The backward reachability analysis continues until one of three stopping conditions is met. First, any calls to `checkPermission` and its wrappers always succeed if they are between calls to `clearCallingIdentity` and `restoreCallingIdentity` because these functions temporarily set the UID to the system services. As a result, any call path that passes through a function called between these two functions will always pass a permission check. Thus, it is not necessary to perform any reachability analysis beyond one of these points and any method called between the `clearCallingIdentity` and `restoreCallingIdentity` functions is a stopping point.

Second, access to a content provider is made with a virtual call to the generic `ContentProvider` class. PScout stops reachability traversal when it reaches a class or subclass of `ContentProvider`. At this point, PScout infers the URI string of the content provider and associates that URI with the permission of the permission check where the reachability analysis started. Thus, access to the `ContentProvider` found on the reachability path is abstracted to a content provider permission check and PScout iterates the reachability analysis until it converges.

Finally, documented APIs often have a generic parent class that is called by many other methods. As a result, once a documented API is reached, calls to its parent class methods are excluded from the analysis.

4. EVALUATION

Before we rely on the specification extracted by PScout, we wish to evaluate the accuracy of the permission specification by measuring the completeness and soundness of the mapping produced by PScout. We define completeness as the fraction of mappings that PScout finds over the total number of mappings that exist. We define soundness as the fraction of *correct mappings* over the total number of mappings found by PScout, where a correct mapping is a mapping between an API call and a permission such that there exists some invocation of that API requiring the permission. The opposite of a correct mapping is an *incorrect mapping*, which is a mapping between an API and a permission extracted by PScout where no invocation of that API could possibly require the mapped permission. Incorrect mappings can occur because PScout uses path-insensitive analysis and thus can include impossible paths in its reachability analysis.

Unfortunately, no "ground truth" exists for the Android permission specification, so we cannot precisely measure the soundness and completeness. Instead, we estimate the quality of these by measuring them relative to existing sources of Android permission information. First, we compare with the permissions declared by the developers of a corpus of 1,260 applications extracted from the Google Android market, which we further refine using our Android UI fuzzer. Second, we compare against a permission mapping produced by API fuzzing in the Stowaway project [14]. Because Stowaway actually executes every path it finds, it cannot have any impossible paths like PScout. As a result, we expect PScout to be more complete, but slightly less sound than Stowaway. Even though PScout works on any version of Android, Stowaway's specification is for Android 2.2 so for the purposes of evaluating PScout's accuracy, all experiments in this section are performed on Android 2.2 as well.

4.1 Application UI Fuzzer

Our application UI fuzzer exercises the UI of Android applications and logs the resultant API calls and permission checks. In contrast to Stowaway, which fuzzes the Android API directly, we indirectly fuzz the API by fuzzing applications that use the API. While it is difficult to obtain a more complete coverage of the API because we are constrained to the APIs that the applications use, fuzzing applications has the advantage that we obtain realistic sequences and parameters to API calls.

Our fuzzing infrastructure consists of a single fuzzer virtual machine (VM) and several Android VMs. For performance, we use an x86 port of Android. To fuzz an application, the fuzzer VM first installs the application onto one of the Android VMs using the standard Android Debug Bridge (ADB). It then proceeds to fuzz the application using an iterative process. Initially, the fuzzer receives UI elements, as well as the mapping from application names to PIDs and UIDs, from a system service we install inside each Android VM. It then proceeds to classify the UI using several screen-specific handlers. Each handler performs two actions. First, it checks if the UI matches the type of screen it is meant to handle. For example, the Login Screen Handler checks that the UI has a username field and a password field and that there is at least one button to click. Second, it generates a candidate action for the screen. To continue the example of the Login Screen Handler, it would heuristically identify the username and password fields, and return a sequence of actions that would fill in the username and password for accounts we have created on various services (we use the same username and password on each service for simplicity) and then click the heuristically identified login button. Finally, if more than one handler returns success on classification, the fuzzer selects the best action to perform based on a predefined ranking of actions and a history of previous actions taken for similar screens.

4.2 Completeness

To evaluate the completeness of PScout's permission specification, we extract a list of API calls made by an application and feed that to PScout's mapping to produce a list of required permissions. We extract the API calls within an application using a combination of static analysis and our UI fuzzer. We need the UI fuzzer to enhance our static extraction because applications may execute API calls that are not present in the static image of the application by dynamically calling methods using Java reflection for example. We note that like Stowaway, our application analysis may be incomplete because it cannot catch all APIs that are invoked through Java reflection. Therefore, the overdeclaration and underdeclaration measurements presented in this section should not be interpreted as absolute measures but only as measures of the relative accuracy of PScout versus Stowaway.

For each application, we compare the list of permissions produced by PScout with those specified by the developer in the application's Manifest file. If we find a permission in the developer declared list that is not in the PScout generated list, this indicates that either the developer overdeclared that permission or PScout is missing a mapping between an API and that permission. To distinguish between these two cases we substitute the developer's permission list with PScout's more constrained permission list and use our UI fuzzer to exercise the application. If the application does not experience a permission error with PScout's permission list, it can either be because the fuzzer did not successfully trigger a permission error, or that no permission error is possible because the developer overdeclared the permissions required for their application. If a permission error does occur, this indicates that PScout is missing an API to permission mapping.

When we perform this experiment, we find that 543 out of 1,260 applications declare at least one "extra" permission that is not in the list produced by PScout. While many applications overdeclare, most do not do so severely – 53% of applications overdeclare by one permission and 95% of applications overdeclare by less than 4 permissions. These numbers agree with the measurement of overdeclaration when Stowaway's specification is applied to the same set of API calls extracted from our corpus of applications, despite using a completely different method for extracting the permission specification (564 overdeclaring apps, 54% and 95% respectively). We conjecture that our slightly lower level of overdeclaration overall is a result of PScout's mapping being more complete than Stowaway's.

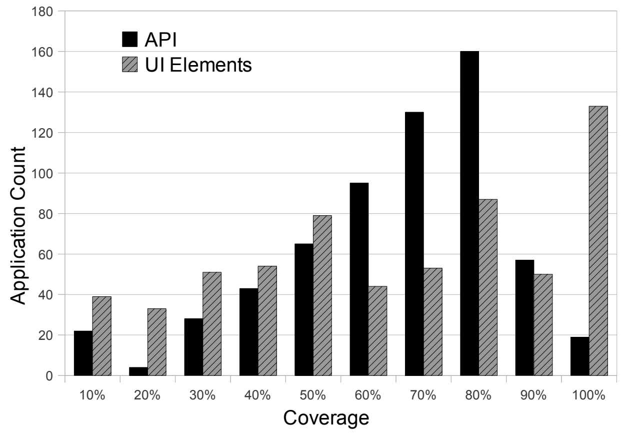

Figure 3: Histogram of UI fuzzer coverage for API calls and UI elements.

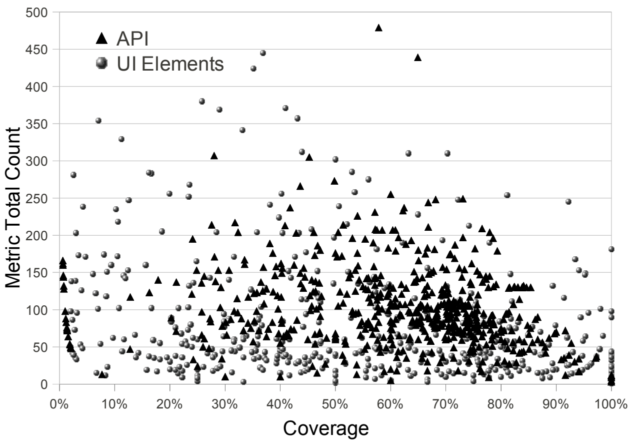

Figure 4: Scatter plot showing relationship between fuzzer coverage and application size

We find that our fuzzer was not able to cause a permission error in any of these applications when executed with PScout's more constrained list of permissions. While our UI fuzzer obtains good coverage in general, it is not able to do so in some cases. This can be due to several factors, such as the application requiring functionality not present on the x86 port of Android (telephony, SMS, GPS) or that certain application behavior is only triggered by non-UI events, which would not be exercised by the fuzzer (network packet arrival, system broadcast, intent from third party applications). We use two metrics to measure the coverage we obtained with our UI fuzzer. First, we statically count the number of UI elements defined in the Android XML layout file and then track which of those UI elements are seen by the fuzzer at run time. Second, we also track the number API calls exercised by the fuzzer as a percentage over the statically identified API calls in each application. We summarize the results in Figure 3. Figure 4 gives a scatter plot showing the relationship between coverage and the number of UI elements and APIs in an application. The broad distribution shows that coverage of our fuzzer is relatively independent of the size and complexity of the application and is likely more dependent on the other factors mentioned above. As a whole, the UI fuzzer is able to obtain coverage of over 70% for both metrics on half of the fuzzed applications.

As a second evaluation of completeness, we directly compare the specification generated by PScout against that generated by Stowaway's API fuzzing. Because API fuzzing tends to be incomplete (the authors could only successfully execute 85% of the API calls they were able to find), we expect and confirm that PScout's mapping is a superset of Stowaway's mapping. Moreover, because we are able to identify and cover many more API calls with our static analysis than Stowaway – we identify 17,218 mappings while Stowaway identifies only 1259. Stowaway's incompleteness is likely due to the fuzzer's inability to find and exercise all APIs in Android and thus produce mappings for those APIs. Despite the much larger set of mappings that PScout finds, PScout does not find significantly more overdeclaration than Stowaway because many of the mappings that PScout has are for undocumented APIs. As we explain in Section 5, applications do not use many undocumented APIs, so the omissions of these APIs from Stowaway's mapping does not result in false detection of overdeclaration by Stowaway on real applications. However, the inclusion of these undocumented APIs in PScout's mapping allows us to perform a more complete analysis of Android's permission map.

We take our evaluation against both Android application developers and Stowaway's API fuzzing results as evidence that the mapping produced by PScout is fairly complete.

4.3 Soundness

We determine the number of incorrect mappings in PScout's specification by examining applications where PScout's permission list included a permission that was not present in the developer list of permissions. In total there are 1072 applications where PScout found "extra" permissions that are not in the developer's list. When we convert the extra permissions into mappings by finding the API that caused PScout to request the permission in the first place, we find 292 unique mappings exercised by the applications, out of which 31 (11%) do not produce a correct permission in any application.

However, not all 31 mappings are necessarily incorrect. Missing permissions in an application's Manifest could be caused by developers forgetting to declare a required permission or by PScout listing a permission that an API requires in certain contexts, but the developer can safely exclude because they know their application never calls the API in those contexts. When run on the same corpus of applications, Stowaway, finds 924 underdeclaring applications, which exercise 194 API mappings. Of these mappings only 7 (4%) mappings do not produce a correct permission in any application. Because Stowaway uses API fuzzing, it cannot have any false mappings – every mapping found by them is accompanied by an execution that demonstrates that the call to the API leads to a check for a certain permission. We intersect PScout's 31 extra mappings with Stowaway's mapping and remove any mappings that appear in Stowaway's specification leaving 24 (8%) possibly incorrect mappings. We then manually examine some of the remaining mappings and further remove 4 mappings: 1 mapping has a conditional check to ensure the application holds at least one of the `ACCESS_COARSE_` or `ACCESS_FINE_LOCATION` permissions; and 3 mappings are associated with the `WAKE_LOCK` or the `ACCESS_NETWORK_STATE` permission where the permission requirement depends on the value of an internal state field. From this, we establish that out of the 292 mappings exercised by our corpus of applications, an upper bound of 20 (7%) may actually be incorrect mappings.

5. PERMISSION SYSTEM ANALYSIS

We run PScout on four versions of Android and summarize the results of the extracted permission specifications in Table 1. We chose these versions because these are the Android versions that are predominantly deployed on phones and tablets in the market. We then analyze the extracted specifications to try and answer the four main questions we posed in Section 1.

Is the large number of permissions offered by Android useful? Are any permissions redundant?

As can be seen from Table 1, Android has had at times, anywhere from 75-79 different permissions available to third party application developers. To answer whether the permissions are redundant, we compute the conditional probability of all pairs of Android permissions across their API mappings. We define two types of correlations between permissions: an implicative relationship (i.e. all APIs that check for permission X also check for the permission Y) or a reciprocative relationship (i.e. the checking of either permission by an API means that the other permission will also be checked with a probability higher than 90%).

Out of the 6162 permission pairs in Android 4.0, we found only 14 implicative permission pairs and only 1 reciprocative pair, which we list in Table 2. Only one of the permission pairs, `KILL_BACKGROUND_PROCESSES` and `RESTART_PACKAGES`, is truly redundant – both permissions are checked by the same set of APIs. The Android documentation indicates that the API requiring `RESTART_PACKAGES` has been deprecated and replaced with an API that requires `KILL_BACKGROUND_PROCESS`[2]. Our analysis shows that the new API works with the old deprecated permission, likely for backward compatibility. We also found that the `READ_` and `WRITE_SOCIAL_STREAM` permissions frequently imply one or both of the `READ_` and `WRITE_CONTACT` permissions. This is not a surprise as the Android documentation explains that permission to access contacts is required to access a user's social stream[3]. Several "write" permissions imply their corresponding "read" permission meaning that the APIs always read and modify the corresponding objects. However, since all these permissions enforce access controls for content providers, separate read and write permissions are still required because applications may access the data directly via content provider URIs. We analyzed the source code for the 3 remaining implicative permission pairs to

[2]http://developer.android.com/reference/android/Manifest.permission.html

[3]http://developer.android.com/reference/android/provider/ContactsContract.StreamItems.StreamItemPhotos.html

	Android Version			
	2.2	2.3	3.2	4.0
# LOC in Android framework	2.4M	2.5M	2.7M	3.4M
# of classes	8,845	9,430	12,015	14,383
# of methods (including inherited methods)	316,719	339,769	519,462	673,706
# of call graph edges	1,074,365	1,088,698	1,693,298	2,242,526
# of permission mappings for all APIs	17,218	17,586	22,901	29,208
# of permission mappings for documented APIs only	467	438	468	723
# of explicit permission checks	229	217	239	286
# of intent action strings requiring permissions	53	60	60	72
# of intents ops. w/ permissions	42	49	44	50
# of content provider URI strings requiring permissions	50	66	59	74
# of content provider ops. /w permissions	916	973	990	1417
KLOC/Permission checks	2.1	2.0	2.1	1.9
# of permissions	76	77	75	79
# of permissions required only by undocumented APIs	20	20	17	17
% of total permissions required only by undocumented APIs	26%	26%	23%	22%

Table 1: Summary of Android Framework statistics and permission mappings extracted by PScout. LOC data is generated using SLOCCount by David A. Wheeler.

	Existing	New API	undoc→doc
2.2→2.3	9	40(6%)	0
2.3→3.2	31	25(6%)	2
3.2→4.0	48	56(19%)	212

Table 3: Changes to the permission specification over time. We give the number of existing APIs that acquired a permission requirement and the number of new APIs that require permissions. The percentage in brackets gives the number of new APIs requiring permissions as a fraction of all new APIs introduced between versions. The final column lists the number of undocumented APIs requiring permissions in the previous version that became documented APIs in the new version.

determine the cause of the correlation. `USE_CREDENTIALS` and `MANAGE_ACCOUNTS` are very related, the former allowing the application to use authentication tokens from registered accounts and the latter to manage those accounts. Both `WRITE_HISTORY_BOOKMARKS` / `GET_ACCOUNTS` and `ADD_VOICEMAIL` / `READ_CONTACTS` are also pairs of related permissions, which are checked when accessing the `browser` and `call_log` content providers respectively. `ACCESS_COARSE_` and `ACCESS_FINE_LOCATION` is the only reciprocative pair. While in most cases `FINE` permission is a superset of `COARSE` permission, getting location changes from the `PhoneStateListener` is only allowed if the `COARSE` permission is held.

Summary: While there are small amounts of redundancy illustrated in these 15 pairs, the vast number of Android permissions have very little correlation with any other permission. As a result, we believe there is little redundancy in the Android permission specification.

How many undocumented APIs require permissions and how common is it for applications to use undocumented APIs?

Table 1 gives the total number of APIs that require permissions as well as the number of documented APIs that require permissions. From this, we can see that there are anywhere from 16K-28K undocumented APIs that require permissions across different versions of Android. In addition, 22-26% of the declared permissions may only be checked if an application uses undocumented APIs. For example: `CLEAR_APP_CACHE`, `SET_DEBUG_APP` and `MOUNT_UNMOUNT_FILESYSTEMS` cannot be required if only documented APIs are used. These permissions are generally related to specialized system functionality, which seems to justify why such functionality is not exposed to average application developers. This suggests that if the intent is for developers to only use documented APIs, then Android could export a significantly smaller list of permissions.

From our study of 1,260 Android 2.2 applications in Section 4, we find that only 53(3.7%) applications use undocumented APIs. Out of the 13,811 APIs that those applications use, only 158(1.1%) are undocumented. In contrast, the same applications use 292 API calls that require permissions, out of which 22(7.5%) are undocumented. Thus, applications make very little use of undocumented APIs, but for the undocumented APIs they do use, a significantly larger fraction of those require permissions than for the documented APIs they use. We also have noticed that across versions, a number of APIs requiring permissions that were undocumented have become documented in later versions. The third column of Table 3 shows the number of undocumented APIs that require permissions became documented in the next major version we examined. As can be seen, initially not many undocumented APIs were made documented, but this changed in Android 4.0, which made many previously undocumented APIs documented, possibly in acknowledgment that some undocumented APIs are useful and that the wider Android developer community should be made aware of them. This also helps explain the lower percentage

permission X	permission Y	P(Y|X)	P(X|Y)
KILL_BACKGROUND_PROCESS	RESTART_PACKAGES	1.00	1.00
WRITE_SOCIAL_STREAM	WRITE_CONTACTS	1.00	0.93
READ_SOCIAL_STREAM	READ_CONTACTS	1.00	0.92
USE_CREDENTIALS	MANAGE_ACCOUNTS	1.00	0.73
WRITE_SOCIAL_STREAM	READ_SYNC_SETTINGS	1.00	0.62
WRITE_SOCIAL_STREAM	READ_SOCIAL_STREAM	1.00	0.59
WRITE_CONTACTS	READ_CONTACTS	1.00	0.58
WRITE_SOCIAL_STREAM	READ_CONTACTS	1.00	0.54
WRITE_HISTORY_BOOKMARKS	READ_HISTORY_BOOKMARKS	1.00	0.39
WRITE_HISTORY_BOOKMARKS	GET_ACCOUNTS	1.00	0.30
WRITE_CALENDAR	READ_CALENDAR	1.00	0.17
ACCESS_LOCATION_EXTRA_COMMANDS	ACCESS_COARSE_LOCATION	1.00	0.05
ACCESS_LOCATION_EXTRA_COMMANDS	ACCESS_FINE_LOCATION	1.00	0.05
ADD_VOICEMAIL	READ_CONTACTS	1.00	0.04
ACCESS_COARSE_LOCATION	ACCESS_FINE_LOCATION	0.95	0.90

Table 2: Highly correlated permissions in Android. P(Y|X) denotes the conditional probability computed by taking the percentage of APIs that check for permission X that also check for permission Y.

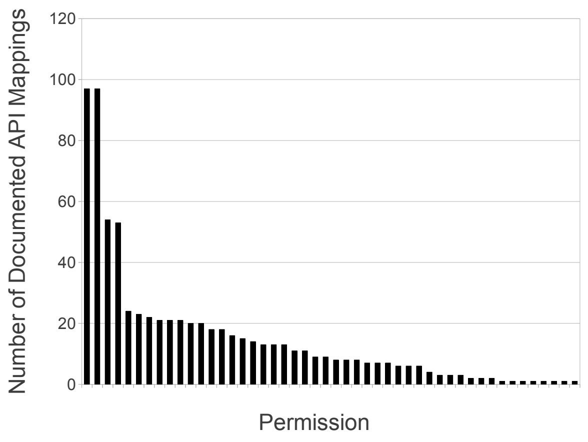

Figure 5: Number of documented APIs that map to a permission in Android 4.0. Each bar represents the number of documented APIs that require a particular permission.

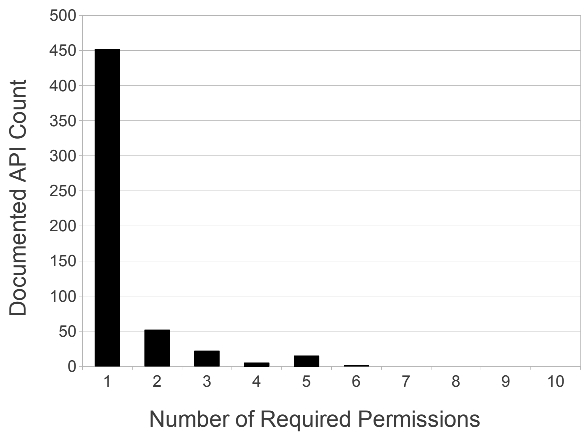

Figure 6: Number of permissions required by a documented API. Each bar represents the number of documented APIs that require that number of permissions.

of permissions that are only used by undocumented APIs in Android 4.0.

Summary: There are many undocumented APIs that require permissions and even some permissions that are only needed if an application is using undocumented APIs. Currently, applications do not commonly make use of undocumented APIs.

How complex and interconnected is the relationship between Android APIs and permissions?

Figure 5 and Figure 6 show histograms of the number of documented APIs that map to a permission and the number of permissions required by an API respectively. Together both of these graphs show that while the permission specification is very broad in size, it is not very interconnected – over 80% of APIs only require at most 1 permission, and very few require more than 2 or 3. Similarly, 75% of permissions have fewer than 20 API calls that require them. We also compute the length of the path in the call graph between API call and the permission check for each mapping. We find that over 60% of the mappings have a path length of less than 5 edges indicating that the permission check happens fairly early on in the processing of most API calls.

Table 4 gives the 5 permissions in Android 4.0 with the highest number of API mappings and the 5 permissions with the highest number of checks. Having a lot of APIs mapping to a permission does not necessarily translate to a widespread functionality for that permission. The top two permissions, `SET_WALLPAPER` and `BROADCAST_STICKY`, are both required for methods in the `Context` class which has 394 subclasses. Most of the documented API mappings for these two permissions are inherited methods from documented subclasses of `Context`. As a result, the number of mappings can sometimes be a function of the object hierarchy rather than functionality. High numbers of API mappings also does not translate into large numbers of permission checks. For example, all APIs using the `SET_WALLPAPER` permission

	# of mappings		# of checks	
Highest # of mappings				
SET_WALLPAPER	97	1,466	1	1
BROADCAST_STICKY	97	2,472	5	7
WAKE_LOCK	54	3,874	4	5
BLUETOOTH	53	1,878	37	67
READ_CONTACTS	24	1,244	29	275
Highest # of checks				
BLUETOOTH	53	1,878	37	67
READ_CONTACTS	24	1,244	29	275
READ_SOCIAL_STREAM	22	1,145	22	163
BLUETOOTH_ADMIN	22	1,362	16	44
READ_SYNC_SETTINGS	21	1,086	12	85

Table 4: Heavily used and checked permissions. This table gives the permissions with the largest number of API mappings and largest number of checks in Android 4.0. In each column, the left number represents mappings and checks for documented APIs only, the right number represents mappings and checks for all APIs.

pass through a single permission check in the `WallPaperManagerSerivce`. Similarly, the vast majority of APIs that need `BROADCAST_STICKY` or `WAKE_LOCK` pass through a small number of permission checks in the `sendStickBroadcast` method and `PowerManagerService` class. The permissions that map to the largest number of APIs tend to be permissions protecting generic system resources. In contrast, a larger fraction of the permissions with a large number of checks, such as `READ_CONTACTS` and `READ_SOCIAL_STREAM`, tend to protect content providers that store private information belonging to the user.

Summary: The permission system is broad but not heavily interconnected. Permissions that have many API mappings tend to protect generic system resources rather than user data and have fewer permission checks.

How has the permission system of Android evolved over time?

From Table 1, even though the amount of code in Android has increased by over 40% from 2.2 to 4.0, there has remained roughly one permission check for every 2KLOC in all versions. Thus, the amount of checking that Android performs to ensure that applications have permissions to access sensitive resources has increase proportionally with the functionality in the OS. However, when we compare the number of documented APIs that require permissions, we see that this has increased by over 54% in Android 4.0, indicating that Android 4.0 exposes more permission-requiring functionality to applications via APIs. This increase can at least be partially explained by examining the Android 4.0 SDK documentation, which highlights key changes involving user profiles and social network sharing on all applications. Such functionality requires access to sensitive content stored by new content providers, which is reflected in the 43% increase in the number of content provider permission checks.

To further explore how the amount of sensitive functionality that is exposed to application has increased over time, Table 3 summarizes how permission requirements have been added to Android APIs over time. As we can see, there have been both existing APIs that have been upgraded to require permissions, new APIs added that require permissions, as well as undocumented APIs that require permissions became documented. In Android 4.0, the LOC increased by 27% while the number of documented APIs only increased by 4%. Recall from Table 1 that the proportion of functionality that requires permissions has remained relatively constant, so with the fewer new APIs in 4.0, a larger percentage of those APIs must require permissions (19% in Android 4.0 versus 6% in previous versions). This indicates there is more permission-requiring functionality behind each new Android 4.0 API.

Initially, we hypothesized that the APIs whose permission requirements changed between versions might have done so because of errors in Android access control policy or due to fundamental changes in the functionality of the APIs. However, when we examined the new paths between the APIs and permission checks that caused the change in permission requirement, we found that the changes were actually often due to subtle and innocuous code changes. For example, in Android 2.2 the `startDiscovery` method in the `BluetoothAdapter` starts bluetooth device discovery and already requires the `BLUETOOTH_ADMIN` permission. Between Android 2.2 and Android 2.3, a call to `getState` was added, which checks if the bluetooth device is on and terminates the function early if it is not. `getState` requires the `BLUETOOTH` permission so as a result, `startDiscovery` also requires the `BLUETOOTH` permission as of Android 2.3. The added functionality was for debugging and does not fundamentally change the behavior of `startDiscovery`, yet results in an additional permission requirement. In another example, between Android 2.3 to Android 3.2, the `resetPassword` method in the `DevicePolicyManagerService` had a call to `checkPasswordHistory` added to it to make sure the user did not reset their password to a recently used password. `checkPasswordHistory` requires the `WRITE_SETTINGS` permission, which allows the application to read or write phone settings, so as a result, `resetPassword` also requires `WRITE_SETTINGS`. However, in Android 4.0, which is supposed to be a merge of Android 2.3 and Android 3.2, the call to `checkPasswordHistory` has been removed, and `resetPassword` no longer needs the `WRITE_SETTINGS` permission. These examples illustrate that changes in the permission requirement over time are often due to arbitrary reasons and the addition or removal of a permission requirement is not often indicative of errors in permission checking or in fundamental changes to API functionality.

We believe that there is a fundamental trade-off between the stability of a permission specification over time and how fine-grain the permission specification is. On one hand, stability of the permission specification is desirable as it means that application developers do not need to update the permissions their applications declare as the underlying OS changes. On the other hand, fine-grain permissions provide better least-privilege protection. For example, combining the `BLUETOOTH_ADMIN` and `BLUETOOTH` permissions would mean that 10% of the APIs that only had the `BLUETOOTH_ADMIN` permission would unnecessarily gain the `BLUETOOTH` permission and 64% of the APIs that only had `BLUETOOTH` permissions would unnecessarily gain the `BLUETOOTH_ADMIN` permission. However, this reduction in least privilege would have prevented the change in permissions needed for `startDiscovery` from Android 2.2 to 2.3. We have found several instances of the same trade-off within Android. While we do not believe that one can have both permission stability and least-privilege together, we believe that awareness of this trade-off may help the design of permission systems in newer mobile OSs such as Windows Phone and B2G.

Summary: Over time, the number of permission checks in Android has remained constant with code size, though the amount of sensitive functionality used by APIs has been increasing. There is a fundamental trade-off between stability of the permission specification and enforcing least-privilege with fine-grain permissions.

6. RELATED WORK

The closest related work is the Stowaway project [14], Bartel et al. [4] and Vidas et al. [21]. The main difference between PScout and previous work is that their focus is to measure the amount of permission overdeclaration in third party applications. As a result, they have varying levels of completeness in the specification they extract to measure the overdeclaration, but none are as complete as PScout. We discuss the specific differences below.

Stowaway extracts an Android permission specification using API fuzzing and as a result is less complete than PScout. For their purposes, this was sufficient since the main purpose of their work was to measure the amount of permission overdeclaration as opposed to extract a complete specification. The authors of Stowaway have made their mapping available so we are able to compare against theirs. For the most part, PScout's specification is a superset of their extracted specification and the rate and amount of overdeclaration we measure in applications also agrees with their results.

Bartel et al. perform a call-graph based analysis on the Android framework that is very similar, but less extensive than PScout's. The main differences are that PScout handles Intent and Content Provider functions whereas Bartel's analysis only infers permission checks on `checkPermission` functions. As a result, while Bartel's mapping is double the size of that reported by Stowaway because they use static analysis, it is still considerably less complete than the mapping produced by PScout, likely because of the missing permission checks.

Vidas et al. extract a permission specification by scanning the Android documentation. As a result, their specification is the least complete of all previous work since the Android documentation is incomplete.

There is a large body of security research on permission-based systems [15, 16] and in Android security [6, 7, 8, 11, 12, 13, 18]. Many techniques are proposed to protect user privacy, detect malware, or certify application security. Batyuk et al. [5] use static analysis to detect privacy leakage in Android applications. AppFence [18] modifies Android OS to anonymize sensitive user data for applications that are not authorized to access it and use taint analysis to prevent applications that are authorized to use sensitive user data from transmitting leaking it. Crowdroid [7] analyzes the pattern of system calls made by applications to detect malware. We feel that having an accurate permission specification for Android, as well as an analysis of that specification, is complementary to work on securing Android and other similar smartphone OSs.

There has been previous work in extracting specifications from programs for the purposes of explicit model checking. Bandera extracts finite state models from Java source code [9] and Lie et al. [19] extract models from cache coherence protocol code. To produce models that are checkable by a model checker, both must abstract details of the implementations and perform size reduction on elements in the code when they extract their models. Other model checkers, such as SLAM [2] and BLAST [17] take a step further and perform automatic abstractions as they check the code. In contrast, the size of the code base PScout is analyzing is far larger than the code base in this previous work.

7. CONCLUSION

We built PScout, a version-independent tool to extract the permission specification and take the first steps to answer some key questions about Android's permission system. One of the challenges with extracting a permission specification from Android is that the permission checks and API calls that lead to them are distributed over an extremely large code base. We find that it is possible to extract an accurate permission specification using light-weight call-graph analysis, augmenting that analysis with domain-specific information to selectively refine parts of that call-graph with flow-sensitive analysis, and using a uniform abstraction for permission checks. Our evaluation of the extracted specification shows that it is more complete when compared to other permission specifications, but still has a low number of false mappings due to infeasible paths.

By using PScout to analyze several major versions of the Android OS we expose some interesting characteristics of the Android permission specification. First, the Android permission system has little redundancy in its set of non-system permissions, but a small subset of the permission can be hidden from most developers since they are only required by undocumented APIs and very few applications use undocumented APIs. Second, the fine-grained permissions cause innocuous code changes to result in churn in the permission requirements of APIs. While a coarser permission set can improve the stability of permission specification and alleviate this churn, it comes at the expense least-privilege protection. From our experience, we believe that PScout can form a basis for more sophisticated static analysis tools to further analyze and understand the implementation and design of smartphone permission systems.

Acknowledgements

The authors would like to thank Adrienne Porter Felt, Phillipa Gill, Mohammad Mannan, David Barrera and the anonymous reviewers for their helpful comments. Kathy is supported by an OGS and a NSERC CGSM scholarship. The work in this paper was also supported in part by the NSERC ISSNet Strategic Network, an NSERC Engage Grant, an NSERC Discovery Grant, an MRI Early Researcher Award and a grant from the Ontario Research Fund.

8. REFERENCES

[1] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie. Short paper: A look at smartphone permission models. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 63–68, Oct. 2011.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, June 2001.

[3] D. Barrera, H. Kayacik, P. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2010.

[4] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Automatically securing permission-based software by reducing the attack surface: An application to Android. Technical report, University of Luxembourg, SNT, 2011. Tech Report.

[5] L. Batyuk, M. Herpich, S. Camtepe, K. Raddatz, A. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In *Proceedings of the 6th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 66–72, Oct. 2011.

[6] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 225–238, June 2008.

[7] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 15–26, Oct. 2011.

[8] A. Chaudhuri. Language-based security on Android. In *Proceedings of the ACM Fourth Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 1–7, 2009.

[9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 439–448, June 2000.

[10] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*, pages 77–101, Aug. 1995.

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 393–407, Oct. 2010.

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX Security Symposium*, pages 21–36, Aug. 2011.

[13] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 235–245, Nov. 2009.

[14] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 627–638, Oct. 2011.

[15] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, pages 7–18, June 2011.

[16] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, pages 22–37, Aug. 2011.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 58–70, Jan. 2002.

[18] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. "These aren't the droids you're looking for": Retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 639–652, Oct. 2011.

[19] D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, pages 192–203, July 2001.

[20] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, page 13. IBM Press, 1999.

[21] T. Vidas, N. Christin, and L. Cranor. Curbing Android permission creep. In *Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011)*, Oakland, CA, May 2011.