Detecting Capability Leaks in Android-based Smartphones

Michael Grace, Yajin Zhou, Zhi Wang, Xuxian Jiang

North Carolina State University
890 Oval Drive, Raleigh, NC 27695
{mcgrace, yajin_zhou, zhi_wang, xuxian_jiang}@ncsu.edu

ABSTRACT

Recent years have witnessed increased popularity and adoption of smartphones partially due to the functionalities and convenience offered to their users (e.g., the ability to run third-party applications). To manage the amount of access given to smartphone applications, Android provides a permission-based security model, which requires each application to explicitly request permissions before it can be installed to run. In this paper, we systematically analyze eight flagship Android smartphones from leading manufacturers, including HTC, Motorola, and Samsung and found out that the stock phone images do not properly enforce the permission model. Several privileged permissions that protect the access to sensitive user data and dangerous features on the phones are unsafely exposed to other applications which do not need to request them for the actual use, a security violation termed *capability leak* in this paper. To facilitate identifying these capability leaks, we take a static analysis approach and have accordingly developed a system called Woodpecker. Our results with eight phone images show that among 13 privileged permissions examined so far, 11 were leaked, with individual phones leaking up to eight permissions. By exploiting these leaked capabilities, an untrusted application can manage to wipe out the user data, send out SMS messages (e.g., to premium numbers), record user conversation, or obtain user geo-locations on the affected phones – all *without* the need of asking for any permission.

1. INTRODUCTION

Recent years have witnessed increased popularity and adoption of smartphones. According to data from IDC [22], smartphone manufacturers shipped 100.9 million units in the fourth quarter of 2010, compared to 92.1 million units of PCs shipped worldwide. This is the first time in history that smartphones are outselling personal computers. The popularity is partially attributed to the incredible functionalities and convenience smartphones offered to end users. In fact, existing mobile phones are not simply devices for making phone calls and receiving SMS messages, but powerful communication and entertainment platforms for web surfing, social networking, location navigating, and online banking etc.

The smartphone popularity and adoption is also spurred by the proliferation of feature-rich devices as well as compelling mobile applications (or simply apps). In particular, these mobile apps can be readily accessed and downloaded to run on smartphones from various *app stores* [5]. For example, it has been reported [21] that Google's Android Market already hosts 150,000 apps as of February, 2011 and the number of available apps has tripled in less than 9 months. Moreover, not only official smartphone platform vendors such as Apple and Google are providing respective app stores to host hundreds of thousands of apps, but also third-party vendors including Amazon are competing in this market by providing separate channels for mobile users to browse and install apps.

Not surprisingly, mobile users are increasingly relying on smartphones to store and handle personal data. Inside the phone, we can find current (or past) geo-location information about the user [7], phone call logs of placed and received calls, an address book with various contact information, as well as cached emails and photos taken with the built-in camera. The type and the volume of information kept in the phone naturally lead to various concerns [16, 15, 25, 37] about the safety of this private information, including the way it is managed and accessed.

To mediate access to various personal information and manage certain advanced phone functions, smartphone platform vendors have explored a number of approaches. For example, Apple uses a vetting process through which each third-party app must be scrutinized before it will be made available in the app store. Google defines a permission-based security model in its Android platform, which requires that any app, before installation, must explicitly request permissions to access certain information and features. In other words, the requested permissions essentially define the capability the user may grant to an Android-based mobile app. By showing the requested permissions to the user before the app can be installed, this permission-based security model allows a user to gauge the app's capability and determine whether or not to install the app in the first place. Due to the central role of the permission-based security model in the Android design, it is critical that the security model is properly enforced in existing Android-based smartphones.

In this paper, we systematically study eight popular Android-based smartphones from leading manufacturers, including HTC, Motorola, and Samsung and are surprised to find out that these stock phone images do not properly enforce the permission-based security model. Specifically, several privileged (or dangerous) permissions that protect access to sensitive user data or features on the phones are unsafely exposed to other applications which do not need to request these permissions for the actual use. For simplicity, we use the term *capability leak* in this paper to represent the situation where an app can gain access to a permission without actually requesting it. Each such situation essentially leads to a violation of the permission-based security model in Android.

To facilitate exposing capability leaks, we have developed a semi-automated system called *Woodpecker*. Woodpecker employs data flow analysis techniques to systematically analyze pre-loaded apps on the phones and for each app explore the reachability of a dangerous permission from a public, unguarded interface. To better examine possible capability leaks, our system distinguishes two different categories. *Explicit capability leaks* allow an app to successfully access certain permissions by exploiting some publicly-accessible interfaces or services without actually requesting these permissions by itself. *Implicit capability leaks* similarly allow an app to obtain unauthorized permissions without requesting them. However, instead of exploiting some public interfaces or services, such an app can acquire or "inherit" permissions from another app with the same signing key (presumably by the same author). Consequently, explicit leaks represent serious security errors as they subvert the permission-based security model Android uses to control access to dangerous features or sensitive data while implicit leaks could misrepresent the capabilities available to an app.

We have implemented a Woodpecker prototype to uncover both types of capability leaks in Android-based smartphones.[1] Our current prototype focuses on 13 representative privileged permissions that protect sensitive user data (e.g., geo-location) or phone features (e.g., sensors like the camera or microphone, or the ability to send SMS messages). We have used our prototype to examine eight popular Android phones: HTC Legend/EVO 4G/Wildfire S, Motorola Droid/Droid X, Samsung Epic 4G, and Google Nexus One/Nexus S. Our results show that among these 13 privileged permissions, 11 were explicitly leaked, with individual phones leaking up to eight permissions.[2] In particular, by exploiting these leaked capabilities, an untrusted app on these affected phones can manage to wipe out the user data on the phones, send out SMS messages (e.g., to premium numbers), record user conversation, or obtain user geo-locations – all *without* asking for any permission.

The rest of this paper is organized as follows: Section 2 describes background information on the Android platform. Section 3 describes our system design for capability leak detection in Android phone images. Section 4 presents its detailed prototype. Section 5 contains the evaluation results from our study of eight Android-based smartphones from leading manufacturers. Section 6 discusses the limitations of our approach and suggests avenues for future improvement. Finally, Section 7 describes related work, and Section 8 summarizes our conclusions.

2. BACKGROUND

In this section, we briefly review key concepts and background about Android, which are essential to our system but may be unfamiliar to some readers. Readers with sufficient background can safely skip this section.

Android [1] is an open source mobile phone platform developed by Google. It is based on Linux but contains its own runtime. Associated with the runtime, Android provides an application framework that contains a wide variety of application programming interfaces (APIs) to facilitate the software development. The framework and third-party apps are mainly written in an Android-specific dialect of Java while some performance-critical parts could be in C.

Each Android app is running inside a Dalvik [6] virtual machine (VM), which is instantiated as a user-level process in Linux. Different apps are running in different Dalvik VMs, isolated from each other.

[1]The reason why we chose to implement Woodpecker in Android is due to its open-source nature and wide adoption. However, the same need for possible capability leak detection exists and similar design principles are applicable for other smartphones (e.g., from Apple).

[2]Since last month (April, 2011), we have been in the process of reporting the discovered capability leaks to the corresponding vendors. So far, Motorola and Google have confirmed all discovered vulnerabilities related to their phones. Meanwhile, we experienced some difficulties with HTC and Samsung. Our experience is similar to others [8], echoing "the seven deadly sins of security vulnerability reporting."[30]

Protection level	Description
`normal`	Low-risk permissions granted to any package requesting them
`dangerous`	Higher-risk permissions that require user confirmation to grant
`signature`	Only packages with the same author can request the permission
`signatureOrSystem`	Both packages with the same author and packages installed in the `/system/` directory tree can request the permission

Table 1: Permission Protection Levels in Android

The Dalvik VM is derived from Java but has been significantly revised (with its own machine opcode and semantics) to meet the resource constraints of mobile phones. Also, when an app is being installed in Android, a unique user identifier (UID) will be assigned to the app. To facilitate sharing information between apps, Android allows multiple apps to share the same UID if the apps are signed with the same developer certificate. As a result, Android relies on the Linux process boundary and this app-specific UID assignment strategy to achieve isolation, preventing a misbehaving or malicious app from disrupting other apps or accessing other apps' files.

Where a regular Linux program only has a single entry point, an Android app can contain multiple entry points. Specifically, each app is composed of one or more different components, each of which can be independently invoked. There are four types of components: *activities*, *services*, *broadcast receivers* and *content providers*. An activity represents part of the visible user interface of an app. A service, much like a Unix daemon, runs in the background for an indefinite period of time. A broadcast receiver, as the name indicates, is a component that does nothing but receive and react to broadcast announcements. A content provider makes a specific set of the app's data available to other apps.

Each Android app is deployed in the form of a compressed package (`apk`). Each `apk` contains a manifest file (`AndroidManifest.xml`) that describes various standard properties about the app, such as its name, the entry points (or interfaces) it exposes to the rest of the system, and the permissions it needs to perform privileged actions. It can also contain an optional `sharedUserId` attribute, which allows two apps if signed with the same developer certificate to share the same UID. Naturally, an Android package will also contain Dalvik bytecode, native code, or both; most presently contain only Dalvik bytecode.

As mentioned earlier, Android defines a permission-based security model [4]. In this model, the principals that have these permissions are apps, not users. Android retains the Linux concepts of user identifier and group identifier, but assigns these identifiers to installed apps rather than users, as only one user is expected to use an Android device. There is a stock set of permissions defined by the Android framework, but developers are free to define additional permissions as they see fit. Each permission has a protection level [3], which determines how "dangerous" the permission is and what other apps may request. Table 1 summarizes the defined protection levels in Android. Note that manufacturers and carriers often customize the Android system to add differentiating features to their products. In that context, the `signature` and `signatureOrSystem` permission protection levels are reserved to define capabilities that only vendor-written software should have access to. Permissions are checked either through annotating entry points defined in the manifest file or programmatically by the Android framework.

To facilitate controlled interaction among apps, Android also provides an inter-process communication (IPC) layer called Binder, which is essentially a message passing interface dealing in parcels of binary data. *Intents* are a special type of parcel that represents either a request to perform an operation or a notification that some event

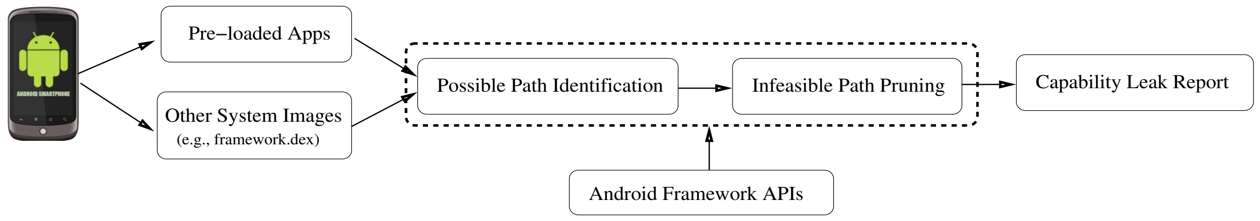

Figure 1: An Overview of Woodpecker

has taken place. In publish-subscribe fashion, Binder routes Intents through the system; during this routing process, it checks any permission annotations, refusing to route Intents from unprivileged senders – or, in some cases, to unprivileged receivers. When an app is contacted via an Intent, one of the defined entry points gets an encapsulated object with information about the Intent content and its sender; it is the permission information carried by this object that can be checked by code running within the receiver.

3. SYSTEM DESIGN

Our goal is to identify *capability leaks*, i.e., situations where an app can gain access to a permission without actually requesting the permission (in its manifest file). Each such situation essentially leads to a violation from the permission-based security model in Android. Since there are some privileged permissions that may not be granted to a third-party user app, we in this work choose to focus on those permissions used by the pre-loaded apps as a part of an Android phone firmware. For simplicity, the terms "permissions," "capabilities," as well as their representative APIs in the Android framework for permission checking are used interchangeably.

Figure 1 shows the high-level overview of our system. Based on the nature of two different kinds of capability leaks (i.e., either explicit or implicit), the system performs two complementary sets of analysis. Specifically, to expose explicit leaks of a capability, our system first locates those (pre-loaded) apps in the phone that have the capability. For each such app, our system further identifies whether a public interface is exposed that can be used to gain access to the capability. (The public interface is essentially the entry point defined in the app's manifest file, i.e., *activity*, *service*, *receiver*, and *content provider*.) In other words, starting from the public interface, there exists an execution path that can reach some use of the capability. If the public interface is not guarded or the execution path does not have a sanity checking mechanism in place to prevent it from being invoked by another unrelated app, we consider the capability leaked. Our system then reports such leaks and further provides evidence that can be used to fashion input to exercise the leaked capability. This evidence includes the value of any variables needed to take the discovered path to the exposed capability, as well as the provenance of any arguments that are passed to it, which can be used to construct a concrete attack.

On the other hand, implicit capability leaks arise from an optional attribute in the manifest file, i.e., "`sharedUserId`." This attribute, if set, causes multiple apps signed by the same developer certificate to share a user identifier. As permissions are granted to user identifiers, this causes all the apps sharing the same identifier to be granted the *union* of all the permissions requested by each app. To detect such leaks in an app that shares a user identifier, our system reports the exercise of an unrequested capability, which suspiciously has been requested by another app by the same author. We stress that an implicit leak requires a certain combination of apps to be installed: an app seeking to gain unauthorized capabilities can only do so if another app, with the same shared user identifier and signing key, is installed to grant the additional permission. In the context of the pre-loaded apps on the phone, we can identify whether such a colluding app exists. However, due to the fact that we cannot rule out the possibility of a colluding app being installed at a later time, its mere absence does not indicate such implicit leak is "safe" or may not occur later.

Assumptions In this work, we consider the scenario where a smartphone user will install an third-party app on the phone, and the author of the third-party app has necessary knowledge of the phone's system image. Moreover, the app aims to maliciously perform some high-privilege activities (e.g., wiping out the user data on the phone or recording the user conversation), whose APIs are protected by permission checks. To do that, the attacker chooses to not request the required permissions to elude detection or these permissions cannot be granted to third-party apps. (Examples include those permissions defined as `signature` or `signatureOrSystem` – Section 2).

Meanwhile, we limit the scope of the attacker's abilities by assuming that the underlying Linux kernel and Binder IPC framework are trusted. Also, we assume that the signing keys to the system image are not available to the attacker. Given these constraints, a malicious app will not be able to directly access the high-privilege APIs. However, as many pre-loaded apps in the system image have the corresponding permissions, if the malicious app can cause one of these apps to invoke the desired API on its behalf, it will have gained actual access to the capability it represents.

3.1 Explicit Capability Leak Detection

As discussed earlier, explicit leaks of a particular capability may occur in any app that has requested it in its manifest file. To detect these leaks, our system analyzes each such app in two steps. The first step, *possible-path identification* (Section 3.1.1), builds a control-flow graph to identify all the possible paths within the program from a well-defined entry point (in the manifest file) to some use of the capability. After identifying these possible paths, the second step, *feasible path refinement* (Section 3.1.2), employs field- and path-sensitive inter-procedural data flow analysis to determine which of these paths are feasible.

3.1.1 Possible Path Identification

Give an app under inspection, our system first extracts the Dalvik bytecode from the app, and then builds a control-flow graph (CFG) to locate all possible paths of execution. Though the CFG construction is a well-studied topic, there are several aspects that are unique in Android and unfortunately make the CFG construction complicated.

The first one comes from indirect control-flow transfer instructions in Dalvik. The Dalvik bytecode targets a hypothetical machine architecture, which does not support most forms of indirect control-flow transfer. In fact, the only indirect transfers in Dalvik's machine language are due to the Java equivalent of pointers: object references. However, object references are rather commonly passed as arguments within an app method, and due to inheritance it is often not possible to unambiguously determine what concrete class a reference represents. Further, as we are analyzing the bytecode, object references will naturally involve type resolution of related objects. In our current prototype, we take a conservative approach. Specifically, when analyzing the Dalvik bytecode for an app, our system maintains a comprehensive class hierarchy. When an ambiguous reference is encountered, we consider all possible assignable classes. This is a straightforward approach, but one that will not introduce any false negatives.[3]

[3]On the other hand, our system can be further improved by integrat-

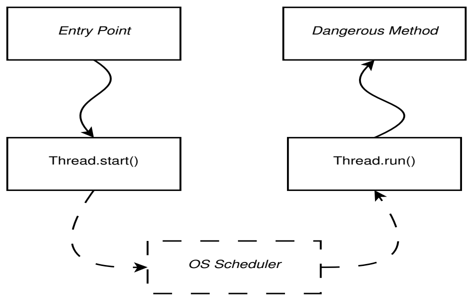

Figure 2: A discontinuity in the control flow introduced by the Android framework.

The second one arises from the event-driven nature in executing an Android-based app, which is also reflected in the app structure. In particular, due to the large number of callbacks used by the Android framework, app execution often passes through the framework to emerge elsewhere in the app. For a concrete example, consider the `java.lang.Thread` class. This class is used to implement native threads, which Android uses in abundance to achieve better UI responsiveness. A developer can simply extend this class, implement the `run()` method, and then call the `start()` method to schedule the thread. However, if we analyze only the code contained within the app, the `run()` method does not appear to be reachable (from `start()`), despite the fact that after the `start()` method is called, control flow goes through the Dalvik VM to the underlying thread scheduler and eventually to the `run()` method. In other words, Android's event-driven nature will unavoidably cause some discontinuity in the CFG construction if we only focus on analyzing the app code (Figure 2). Fortunately, beyond CFG construction, this intervening framework code is of no particular value to our analysis, and its behavior is well-defined in the Android framework APIs. Therefore, we leverage these well-defined semantics to link these two methods directly in the control flow graph, resolving the discontinuity in the process. We have applied this strategy to a number of other callbacks, such as those for Android `Message` queues, timers, and GPS position updates. Some unrelated callbacks may also be safe to ignore. For example, our current prototype does not consider Android UI-related callbacks as they require the user's intervention, which could expose and block the attack.

The third one is that an Android-based app does not necessarily have only one entry point. Instead, rather than a traditional "main method" of some kind, an Android app contains one or more components defined in its manifest file. Each component can potentially define multiple entry points accessible through the Binder IPC mechanism. To take these factors into account, our prototype iterates through each entry point defined in the manifest file to build the CFG (by essentially treating it as a separate program). For each resulting CFG, we can then locate possible paths, each indicating the reachability from a known entry point to a point that exercises a specific permission of interest.

3.1.2 Feasible Path Refinement

The previous step produces control-flow graphs which may represent a tremendous number of potential paths. Among these possible paths, not all of them lead to a dangerous call that exercises the permission of interest, and of those that do, not all are feasible. Therefore, we employ inter-procedural data flow analysis to find paths that are both feasible and result in a dangerous call.

Specifically, we use symbolic path simulation, a path-sensitive data flow analysis technique. The underlying intuition behind this strategy is that a path of program execution can be modeled as a set of program states, each dependent on the last. In order for this set of states to be feasible, each program point (instruction) must follow from the preceding ones. Similar to other data flow analysis techniques, symbolic path simulation implements an iterative algorithm that converges on a fix-point. At each program point, the set of input states are fed through a transfer function (representing the operation performed by that instruction) to produce a set of output states. However, before these output states are used as input states for that program point's successors, we verify that their constraints are consistent. In this way, infeasible paths are not fed forward through the analysis.

ing the latest developments in scalable, comprehensive points-to analysis (Section 6).

Algorithm 1: Capability leak detection

```
Input: entry points, known method summaries
Output: a set of capability leaks
foreach entry point ∈ entry points do
    $worklist$ = initial state: start of the entry point
    $states$ = initial state
    $summaries$ = known method summaries
    foreach $state \in worklist$ do
        remove $state$ from $worklist$
        if state's instruction is a method call then
            if a summary does not exist for the target then
                summarize(target, summaries);
            end
        end
        $worklist += \delta(state) - states$
        $states += \delta(state)$
    end
    if a dangerous-call state is flagged then
        report the state as a capability leak
    end
end
```

Algorithm 1 summarizes our infeasible path reduction. In essence, it is a field- and path-sensitive symbolic simulation algorithm that considers multiple similar concrete paths through a program at once, while condensing methods into parameterized summaries relating their inputs to their outputs. Each state in the analysis encodes the value of data fields with constraints, allowing some similar states to be joined with one another. Particularly, the algorithm operates in the standard fashion for data flow analysis: a worklist is maintained of actively-considered states, and a transfer function (δ) is used to generate new states from a given state. Only new states are added to the worklist, so eventually the algorithm converges on a solution that represents all the feasible states reachable from a given entry point.

Our algorithm has also been enhanced with several optimizations. For example, we accelerate the process by using method summaries to avoid recursively considering the same method-call chains multiple times. To save space, *joining* (rather than simply adding) new states to the worklist and visited-state list make the algorithm scale better both in terms of time and memory. Its actual implementation is cognizant of the value constraints placed on each memory item, and tries to merge very similar states where possible. As an example, if two states are joined that only differ by whether a boolean value is true or false, the resulting state will simply remove any constraint on the boolean value. In this way, fewer states need to be remembered, and fewer successors must be calculated using the transfer function δ.

While the algorithm itself and some of the above optimizations are rather standard, applying the algorithm to Android, however, leads to some unique challenges and benefits. For example, recall that an Android app can define multiple entry points, so we need to produce a separate set of potential paths for each. These paths do not include any executed instructions in the app prior to the entry point, which excludes such code as any constructors that set the initial state of the app. Due to the fact that the entry points in an app can be invoked in any sequence, we opt to take a conservative approach. Specifically, we do not attempt to model this initially-executed code, since any field initializations made in it could have been overridden by another exposed interface. As a result, our algorithm begins by assuming that a field might contain any assignable value. As that field is used along a path of execution, the list of possible values shrinks each time it is used in a way that renders some candidate values impossible.

When reducing infeasible paths, we also face the same type inference problem experienced in the first step. Fortunately, the set of inferences built up by symbolic path simulation naturally mitigates the path explosion caused by our first step. Specifically, object instances can be tracked during the path simulation, and some paths will become infeasible after the system first "guesses" at the object's type somewhere along a path. In addition, certain Dalvik bytecode, especially type-carrying instructions, also help. For instance, the `check-cast` opcode establishes that its operand can be assigned to the supplied type, or an exception is thrown.

In addition, the execution of the algorithm in our case also involves handling Android framework APIs or methods that do not belong to the app. Specifically, the app under inspection may invoke certain APIs that are exported by the Android framework. In our algorithm, the transfer function for a method invocation opcode is a *method summary*, which essentially phrases the method's outputs in terms of its inputs. Without statically analyzing the code for an external method – and all of its dependencies – we cannot build such a summary. Yet analyzing the entire Android framework would easily lead to state explosion and place an enormous burden on our algorithm. To solve this dilemma, we again leverage the well-defined API semantics of the Android framework. Specifically, it contains a remarkably robust set of predefined libraries, which reduces the need for developers to pull in third-party libraries to support their code. By summarizing these built-in classes ahead of time, we can avoid paying the time, space, and complexity costs associated with doing so each time during application analysis. In our prototype, we found that this approach allows us to phrase some functions more succinctly than the algorithm would, as we can trim out unimportant details from the summaries.

During this infeasible path pruning step, we also need to account for explicit permission checks within the identified path. An app might allow any caller to invoke its entry points, yet deny unprivileged callers access to dangerous functionality by explicitly checking the caller's credentials before any dangerous invocations. Such an arrangement would not constitute a capability leak, and so should not be reported. A naïve solution would be to mark any path encountering an interesting permission check as infeasible. However, our approach does not know what kind of dangerous call lies at the end of the path beforehand. Allowing unrelated permission checks to mark whole paths as infeasible would therefore introduce false negatives. Instead, we model the permission system within our artificial method summaries. Explicit permission checks set a flag along their "true" branch; if that path of execution later encounters a corresponding dangerous call, it is not reported as a capability leak.

A side benefit of performing this kind of analysis is that it models all data flow assignments, not just those relating to branch conditions. As a result, we can trace the provenance of any arguments to the dangerous method. With such information, we can characterize the severity of the capability leak. A capability leak that directly passes through arguments from the external caller is obviously worse than one that only allows invocation with constant values, and this design can distinguish between the two. Given that path feasibility is undecidable, our design errs on the side of caution: it will not claim a feasible path is infeasible, but might claim the reverse is true. As a result, this argument information is valuable, as it can be used to generate a concrete test case that verifies the presence of the detected capability leak.

3.2 Implicit Capability Leak Detection

In explicit capability leak detection, we focus on those apps that request permissions of interest in their manifest file. If an app has a `sharedUserId` in its manifest but does *not* request a certain (dangerous) permission, we also need to investigate the possibility of an implicit leak.

To detect implicit capability leaks, we employ much the same algorithm as for explicit leaks. However, some of the inputs to that algorithm must be changed to reflect a basic difference in focus: explicit capability leak detection assumes the caller of an app's exposed API is malicious, while implicit capability leak detection assumes the app itself might be malicious. Accordingly, instead of only starting from the well-defined entry points in the explicit leak detection, we also need to broaden our search to include the app's initialization.

However, modelling the initialization process in an Android app is somewhat complicated. Specifically, there are two kinds of constructors to handle: *Instance* constructors are explicitly invoked in the Dalvik bytecode (with the `new-instance` bytecode operation); *Class* constructors – `static` initialization blocks – are implicitly invoked the first time a class is used. Accordingly, instance constructors are relatively straightforward to handle as they need to be explicitly invoked. However, class constructors are more complicated. In particular, a class constructor may be invoked in a number of scenarios: it is instantiated with the `new` keyword, a `static` member of the class is referenced, or one of its subclasses is likewise initialized. This means that this type of initialization can occur in a variety of orders. In our prototype, we treat all of the relevant instructions as branches, and take into account the class loading order to determine the path feasibility. Also, in our system, we consider a capability to have been implicitly leaked if there is *any* way to exercise it, which is different from explicit capability leak detection. (This has implications in changing method summaries used for pruning infeasible paths – Section 3.1.2.)

Finally, once we have identified that an implicit capability leak exists, we can perform an additional step to determine whether that leak may actually be exercised. In the context of a phone's system image, we can determine the runtime permissions granted to each shared user identifier by crawling the manifest files of all the packages in the image. We union the permissions granted to each application with a given shared user identifier, which yields the set of permissions given to each of them. We report any implicitly leaked permissions contained within that set.

4. IMPLEMENTATION

We have implemented a Woodpecker prototype that consists of a mixture of Java code, shell scripts and Python scripts. Specifically, our static analysis code was developed from the open-source `baksmali` disassembler tool (1.2.6). As discussed in Section 3, our static analysis approach is rather standard. In the following, we mainly focus on the peculiarities of the Android platform as well as the deviations they cause from standard practice.

4.1 Dalvik Bytecode & Manifest Extraction

To detect possible capability leaks in an Android phone, our system first leverages the Android Debug Bridge (`adb`) tool [2] to obtain access the phone's system image, mainly those files in the `/system/app` and `/system/framework` directories. These directories contain all of the pre-installed apps on the device, as well as any dependencies they need to run.

After obtaining the phone image, we then enumerate all pre-installed apps. For each app, our system decompresses the related Android package (`apk`) file to extract its manifest file (`AndroidManifest.xml`), which is then paired with the app's bytecode (either `classes.dex` or its `odex` variant). A standalone script has been developed to extract all the pre-installed apps and disassemble them to extract their bytecode for subsequent analysis. Based on the number of apps installed on the device as well as the complexity or functionality implemented in these apps, this process typically takes on the order of ten minutes per smartphone image.

Based on the extracted app manifest, we further comb through it for two things: requests for any permissions of interest and the optional `sharedUserId` attribute. Apps that are granted related permissions are checked for explicit capability leaks, while those with the `sharedUserId` attribute set are checked for implicit capability leaks. Naturally, we also compute the actual set of permissions granted to each pre-loaded app by combining all the permission requests made with the same `sharedUserId`.

4.2 Generic Control-Flow Graph Construction

Among all the pre-loaded apps on the phone device, we iterate through each app to detect possible capability leaks. As there are tens of dangerous permissions defined in the Android framework, instead of building a specific control-flow graph (CFG) for each permission, we choose to first build a generic CFG to assist our static analysis.

Specifically, we start from each entry point and build the CFG. The generic CFG will be the union of those CFGs from the defined entry points. There is a subtlety between components defined in the manifest file and the actual entry points for CFG construction. In particular, some entry points are standard and can be readily determined by the type of components contained within the app. There are in total four types, and each has a predefined interface to the rest of the system. For instance, any "receiver" defined in the manifest file must subclass `android.content.BroadcastReceiver`. In such cases, inspecting the class hierarchy allows to determine that the "`onReceive(Context, Intent)`" method is an entry point (as per the specification).

Moreover, among these four types, three of them solely take data objects as inputs through their entry points, but services can be different. In particular, Android defines a CORBA-like binding language, the Android Interface Definition Language (AIDL), which allows services to expose arbitrary methods to other apps. `aidl` files are used at compile-time to manufacture Binder stubs and skeletons that encapsulate the necessary IPC functionality. At run-time, the component's `onBind(Intent)` method is called by the system, which returns an `android.os.Binder` object. The methods contained within this object are then exported to callers that have a compatible skeleton class. Since we only analyze the bytecode and do not have access to the original `aidl` files used to define the interface, there is a need to further parse and infer the internal structure of the `Binder` object. Each such object contains an `onTransact()` method that is passed a parcel of data that encodes which method to call. We can then treat this method as an entry point in order to build our CFG. However, once the graph has been built, it is more semantically accurate to treat the embedded method calls in `onTransact()` as entry points for the purposes of our feasible path refinement stage.

From another perspective, Android-based apps essentially expose a set of callbacks to the system instead of a single "main method." Our system leverages the knowledge of how these callbacks are defined in Android to identify them. In addition, the Android framework defines many other callbacks at run-time, which will similarly cause discontinuities in the CFG generation. One example is the previous `Thread.start()->run()` scenario. In our prototype, instead of statically analyzing the entire Android framework, we opt to use knowledge of the framework's semantics to connect the registration of a callback to the callback itself. To automate this process, we provide a boilerplate file that represents knowledge about the framework. This file contains simplified definitions for any explicitly-modelled method in the framework, written in the `dex` format; it is fed into our system alongside the app's code to facilitate CFG construction.

Permission	Capability
ACCESS_COARSE_LOCATION	Access coarse location (e.g., WiFi)
ACCESS_FINE_LOCATION	Access fine location (e.g., GPS)
CALL_PHONE	Initiate a phone call (without popping up an UI for confirmation.)
CALL_PRIVILEGED	Similar to CALL_PHONE, but can dial emergency phone numbers (e.g., 911)
CAMERA	Access the camera device
DELETE_PACKAGES	Delete existing apps
INSTALL_PACKAGES	Install new apps
MASTER_CLEAR	Remove user data with a factory reset
READ_PHONE_STATE	Read phone-identifying info. (e.g., IMEI)
REBOOT	Reboot the device
RECORD_AUDIO	Access microphones
SEND_SMS	Send SMS messages
SHUTDOWN	Power off the device

Table 2: 13 **Representative Permissions (For brevity, we omit the** `android.permission.` **prefix in each permission)**

4.3 Capability Leak Detection

With the generic CFG and the set of entry points, we then aim to identify possible execution paths from one of the entry points to some use of an Android API that exercises a permission of interest. If the path is not protected by the appropriate permission checks and its entry point is publicly accessible, an explicit capability leak is detected. Due to the large number of sensitive permissions defined in the Android framework, our study chooses thirteen representative permissions marked `dangerous`, `signature` or `signatureOrSystem`. These permissions are summarized in Table 2 and were chosen based on their potential for abuse or damage.

For each chosen permission, our first step is to identify the list of related Android APIs that might exercise the permission. However, such a list is not easy to come by. In fact, we found out that though Android's permission-based security model might be comprehensive enough in specifying the permissions required to access sensitive data or features, the available API documentation is incomplete about which APIs a permission grants access to. Specifically, when dealing with various apps in the system image, we encountered numerous permissions not meant for general consumption – and that therefore do not even have formally specified APIs. One example is "`android.permission.MASTER_CLEAR`," which allows an app to perform a factory reset of the smartphone. This permission is marked as `signatureOrSystem`, so only apps included in the system image can request it; it is intended to be implemented by the vendor and only used by the vendor, so none of the APIs listed in the API documentation check this permission.

For each related permission and the associated Android APIs, our next step then reduces the generic CFG to a permission-specific CFG. Within the reduced CFG, we can then apply the Algorithm 1 to locate possible execution paths from an entry point to the associated Android APIs. For each identified path, we further look for the presence of certain permission checks. Our experience indicates that some permission checks are already defined in the manifest file (and thus automatically enforced by the Android framework). However, many others will explicitly check their caller's permissions. In our prototype, we resort to the Android Open Source Project (AOSP) to find explicit permission checks in the framework. There are also some cases that do not fall under the AOSP. For them we have to apply `baksmali` to representative phone images and then manually examine each explicit permission check. Using the previous example of "`android.permission.MASTER_CLEAR`," Android provides an interface, `android.os.ICheckinService`, which declares the `masterClear()` method. The Samsung Epic 4G's factory reset implementation contains a class named `com.android.server.FallbackCheckinService`. This class implements this Android interface, whose `masterClear()` method explicitly checks the "`android.permission.MASTER_CLEAR`" permission.

To facilitate our static analysis, our prototype also includes a fictitious `dangerous` class that has many static permission-associated member fields. Each identified Android API call, if present in an execution path being analyzed, will update the member field related to the associated permission. As a result, we can detect dangerous calls by simply listing the related member fields in this class.

To model the impact a caller's permissions have on whether a dangerous call can succeed, we use another fictitious `permission` class. This class contains a number of member fields and an artificial method definition for `Context.checkCallingPermission()`. This method sets these member fields dependent upon the permission it is called with. In other words, each member field flags whether a path of execution has checked a particular permission. During an explicit capability leak analysis run, we only consider a capability to have been leaked if a state exists that contains a dangerous-call field modification (in `dangerous`) and does not have the corresponding permission-check flag set (in `permission`). Implicit capability leak analysis does not need to be concerned about the value of the permission-check flags. Instead, it is sufficient to have a dangerous call field modification (in `dangerous`).

5. EVALUATION

In this section, we present the evaluation results of applying Woodpecker to eight smartphones from four vendors, including several flagship phones billed as having significant additional bundled functionality on top of the standard Android platform. We first describe our methodology and tabulate our results in Section 5.1. In Section 5.2, we present three case studies: two explicit leaks and one implicit leak. Lastly, Section 5.3 consists of a performance measurement of our system, both in terms of the accuracy of its path-pruning algorithm and its speed.

5.1 Results Overview

In order to assess capability leaks posed in the wild, we selected phones representing a variety of manufacturers and feature sets. Table 3 shows the phone images and their versions we analyzed using Woodpecker. These phones span most of the 2.x version space, and as shown by the app count for each phone image, some are considerably more complex than others.

Vendor	Model	Android Version	# Apps
HTC	Legend	2.1-update1	125
	EVO 4G	2.2.2	160
	Wildfire S	2.3.2	144
Motorola	Droid	2.2.2	76
	Droid X	2.2.1	161
Samsung	Epic 4G	2.1-update1	138
Google	Nexus One	2.3.3	76
	Nexus S	2.3.3	72

Table 3: Experimented Android-based Smartphones

Running Woodpecker on each phone image produces a set of reported capability leak paths. For each reported path, we then manually verify the leak by tracing the path through the disassembled Dalvik bytecode. For explicit capability leaks whose paths seem plausible, we then craft a test application and run it on the actual device, where possible. The results are summarized in Table 4.

After identifying these capability leaks, we have taken considerable time in reporting them to corresponding vendors. As of this writing, Motorola and Google have confirmed all the reported vulnerabilities in the affected phones. HTC and Samsung have been really slow in responding to, if not ignoring, our reports/inquires. Though the uncovered capabilities leaks on the HTC and Samsung phones have not been confirmed from respective vendors, we have developed a test app to exercise and confirm all the discovered (explicit) capability leaks on the affected phones.

We believe these results demonstrate that capability leaks constitute a tangible security weakness for many Android smartphones in the market today. Particularly, smartphones with more pre-loaded apps tend to be more likely in having explicit capability leaks. The reference implementations from Google (i.e., Nexus One and Nexus S) are rather clean and free from capability leaks except only one explicit leak (marked as ✓^{2}in Table 4) by an app `com.svox.pico`. This app defines a receiver, which can be tricked to remove another app, `com.svox.langpack.installer` by any other third-party app.[4] Our data also show that these capability leaks are not very evenly distributed among smartphones – at least for the thirteen permissions we modelled. For example, those smartphones with system images (i.e., Motorola Droid) very close to the reference Android design (i.e., Google Nexus One and Nexus S) seem to be largely free of capability leaks, while some of the other flagship devices have several. Despite this general trend, we caution against drawing any overly broad conclusions, as some devices (e.g., Motorola Droid X) with higher app counts nevertheless contained fewer capability leaks than substantially simpler smartphones (e.g., HTC Legend).

5.2 Case Studies

To understand the nature of capability leaks and demonstrate the effectiveness of our system, we examine three scenarios in depth. These scenarios were selected to illustrate some of the patterns we encountered in practice, as well as how our system was able to handle them.

5.2.1 Explicit Capability Leaks Without Arguments

The simplest scenario, from Woodpecker's perspective, involves an entry point calling a dangerous capability that is not influenced by any arguments. These capabilities tend to have simpler control flows, as there are no arguments to validate or parse. The Samsung Epic 4G's `MASTER_CLEAR` explicit capability leak is of this type, as it ends with the dangerous call `CheckinService.masterClear()`, which functionally performs a factory reset.

To understand how Woodpecker detects this explicit capability leak, we first explain the normal sequences when the `MASTER_CLEAR` capability is invoked. Specifically, the Samsung Epic 4G's phone image has a pre-loaded app, `com.sec.android.app.SelectiveReset`, whose purpose is to display a confirmation screen that asks the user whether to reset the phone. The normal chain of events has another system app broadcast the custom `android.intent.action.SELECTIVE_RESET Intent`, which the `SelectiveResetReceiver` class (defined in the pre-loaded app) listens for. When this class receives such an intent, it opens the user interface screen (`SelectiveResetApp`) and waits for the user to confirm their intentions. Once this is done, the `SelectiveResetService` is started, which eventually broadcasts an intent `android.intent.action.SELECTIVE_RESET_DONE`. The original `SelectiveResetReceiver` class listens for this `Intent` and then calls `CheckinService.masterClear()`.

Our system detects the last part of the above chain starting after the broadcasted intent `android.intent.action.SELECTIVE_RESET_DONE` is received in the same pre-loaded app. In particular, the intent arrives at one entry point defined in the app's manifest file (i.e., the `onReceive(Context, Intent)` method within `SelectiveResetReceiver`), which then executes a rather straightfor-

[4]This `com.svox.pico` app implements a text-to-speech engine that the accessibility APIs use to talk. However, it exports a public receiver interface, `com.svox.pico.LangPackUninstaller` for `android.speech.tts.engine.TTS_DATA_INSTALLED` intents. If this intent is received, this app will blindly remove another app `com.svox.langpack.installer`, which is hard-coded in the implementation.

	HTC						Motorola				Samsung		Google			
Permission	Legend		EVO 4G		Wildfire S		Droid		Droid X		Epic 4G		Nexus One		Nexus S	
	E	I	E	I	E	I	E	I	E	I	E	I	E	I	E	I
ACCESS_COARSE_LOCATION	✓	✓	✓	✓	·	✓	·	·	✓	·	·	·	·	·	·	·
ACCESS_FINE_LOCATION	✓	·	✓	·	·	✓	·	·	✓	·	·	·	·	·	·	·
CALL_PHONE	·	·	·	·	·	·	·	·	·	·	✓	✓	·	·	·	·
CALL_PRIVILEGED	·	·	·	·	·	✓[1]	·	·	·	·	·	·	·	·	·	·
CAMERA	✓	·	✓	·	✓	·	·	·	·	·	·	·	·	·	·	·
DELETE_PACKAGES	✓[2]	·	✓[2]	·	✓[2]	·	✓[2]	·	✓[2]	·	✓[2]	·	✓[2]	·	✓[2]	·
INSTALL_PACKAGES	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
MASTER_CLEAR	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
READ_PHONE_STATE	·	✓	·	✓	·	✓	·	·	✓	·	·	✓	·	·	·	·
REBOOT	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·
RECORD_AUDIO	✓	·	✓	·	✓	·	·	·	·	·	·	·	·	·	·	·
SEND_SMS	✓	·	✓	·	✓	·	·	·	·	·	·	·	·	·	·	·
SHUTDOWN	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·
Total	6	2	8	2	4	4	1	0	4	0	3	2	1	0	1	0

Table 4: Capability Leak Results of Eight Android-based Smartphones (E: explicit leaks; I: implicit leaks)

ward `Intent`-handling code sequence:

1. Determines that the `Intent` it just received is a `android.intent.action.SELECTIVE_RESET_DONE` operation;
2. Gets the `CheckinService` that contains the master clear functionality;
3. Checks whether it was retrieved successfully;
4. Calls `CheckinService.masterClear()` in a worker thread.

Since `CheckinService.masterClear()` takes no arguments, no additional dataflow analysis needs be performed to characterize the capability leak.

In our experiments, we found other capability leaks of the same nature, including the `REBOOT` and `SHUTDOWN` leaks on the HTC EVO 4G, which involve a receiver listening for a new custom `Intent` and then immediately exercising the related functionalities. Also, on the same phone, we found a new vendor-defined capability `FREEZE` exposed by a system app, which disables the phone's touchscreen and buttons until the battery is removed. In those cases, there is literally no control flow involved, making these capability leaks trivial to exploit.

5.2.2 *Explicit Capability Leaks With Arguments*

Looking beyond simple imperative capability leaks, we consider more complicated cases that involve arguments-taking capabilities. For example, Android's SMS API consists of three methods, each of which takes five or six arguments. The HTC phones have an explicit leak of this capability that entails significant preprocessing of these arguments, so we examine it next.

On these phones, the general `com.android.mms` messaging app has been extended to include a non-standard service, `com.htc.messaging.service.SmsSenderService`, which is used by other vendor apps to simplify sending SMS messages. This service can be started with an `Intent` that contains a number of additional data key-value pairs, known as `Extras`. Each `Extra` contains some data about the SMS to be sent, such as the message's text, its call-back phone number, and the destination phone number etc.

The `SmsSenderService` service processes these fields in its `onStart(Intent, int)` entry point, ensuring that the mandatory key-value pairs exist, including the message body and destination phone number. If they do, the `Intent` is bundled into a `Message` and sent to the `SmsSenderService$ServiceHandler` class via the Android message-handling interface. This interface is designed to allow different threads of execution to communicate using a queue of `Messages`. The typical paradigm uses a subclass of `android.os.Handler` to poll for new `Message` objects, using a `handleMessage(Message)` method. Such `android.os.Handler` objects also expose methods to insert `Messages` into their queue, such as `sendMessage(Message)`.

When building possible paths and pruning infeasible paths, our system will diligently resolve the super- and sub-class relationships that bracket the message-passing code. In this case, the initial `SmsSenderService$ServiceHandler.sendMessage(Message)` call fully specifies the class that `sendMessage(Message)` will be called upon, but `SmsSenderService$ServiceHandler` does not contain a definition for that method. Looking to its superclass, `android.os.Handler`, Woodpecker finds an artificial method definition of the appropriate signature. This definition in turn calls the `android.os.Handler.handleMessage(Message)` method, which is extended by the `SmsSenderService$ServiceHandler` class. In this case, our design has no difficulty resolving these relationships, because the first call fully specifies the `SmsSenderService$ServiceHandler` class. This type information is then carried forward through the call chain as a constraint on the arguments to each call, as a class' methods are associated with an object instantiating that class via an implicit argument (the `this` keyword).

Ultimately, the app execution flow will reach `SmsManager.sendMultipartTextMessage()`, a method that exercises the dangerous `SEND_SMS` permission. The arguments by this point have been transformed: the destination address remains the same, but the call-back number may not have been provided by the `Intent`'s data, and the message body might have been chunked into SMS-sized pieces if it is too long. When processing this execution path, Woodpecker reports this path as feasible and thus exposing the exercised permission `SEND_SMS`. Since the exercised capability took a number of arguments, our system also reports the provenance of each related argument to the Android API, which allows for straightforwardly linking the API arguments back to the original `Intent` passed to the entry point at the very beginning. In other words, by simply including a premium number in the intent, the built-in app will start sending SMS messages to this premium number!

Our experience indicates most capability leaks we detected are of this form. For example, the explicit leak of `CALL_PHONE` capability in Samsung Epic 4G involves passing a component a "technical assistance" phone number, which it calls after considerable processing. Similarly, all the tested HTC phones export the `RECORD_AUDIO` permission, which allows any untrusted app to specify which file to write recorded audio to without asking for the `RECORD_AUDIO` permission.

5.2.3 *Implicit Capability Leaks*

Explicit leaks seriously undermine the permission-based security model of Android. Implicit leaks from another perspective misrepresent the capability requested by an app. In the following, we choose one representative implicit leak and explain in more detail.

Specifically, the HTC Wildfire S has a built-in MessageTab app, `com.android.MessageTab`, which uses the `CALL_PRIVILEGED` capability (marked as ✓[1]in Table 4) without declaring it in its manifest. This MessageTab app is intended to manage the phone's SMS messages, allowing the user to review sent messages and send new ones. For the sake of convenience, this app links messages sent to contacts with the appropriate contact information, allowing the user to dial contacts directly through a "contact details" screen. However, this app does not declare the correct permissions to call phone numbers, as it only requests SMS-related permissions: neither the `CALL_PHONE` nor `CALL_PRIVILEGED` permission occur in its manifest. On the other hand, MessageTab does declare a `sharedUserId` attribute: "`android.uid.shared`." This user identifier is used by a number of core Android apps, including `com.android.htcdialer` – which has both phone-dialing permissions.

When analyzing this app, Woodpecker reports an implicit leak in the `com.android.MessageTab.ContactDetailMessageActivity2` activity component. Specifically, this component has a `onResume()` method – an entry point called when the activity is displayed on the screen. In this case, it is used to instruct on how to build a list of contacts to display on the screen, by calling `com.htc-.widget.HtcListView.setOnCreateContextMenuListener()` with a callback object (`ContactDetailMessageActivity2$3`). When the user long-presses one of these contacts, that callback object's `onCreateContextMenu()` method is called. This method then calls `ContactDetailMessageActivity2.addCallAndContactMenuItems()` to make the contacts' context menus. A call to a helper method, `android.mms.ui.MessageUtils.getMakeCallDirectlyIntent()`, builds the `Intent` to send to dial a contact. This helper method builds the actual `android.intent.action.-CALL_PRIVILEGED Intent`, which will be broadcasted when the user clicks on the contact. From the disassembled code, the `addCallAndContactMenuItems()` method also registers an `ContactDetailMessageActivity2$MsgListMenuClickListener` object as a callback for the click-able contact. This object's `onMenuItemClick(MenuItem)` method is then called, which takes the `Intent` associated with the contact and calls `com.android.internal-.telephonyITelephony.dialWithoutDelay(Intent)` with it, which immediately dials a phone number.

Note that this implicit capability leak traversed a number of callbacks that either require user intervention or are very visible to the user. These callbacks would normally not be considered useful for an explicit capability leak, which assumes a malicious caller. However, as implicit capability leaks assume that the app itself may be malicious, our algorithm simply reports them by not making such value judgments when considering possible execution paths.

5.3 Performance Measurement

Next, we evaluate the performance of our prototype, in terms of both the effectiveness of its path pruning algorithm and the amount of time it takes to process a smartphone's system image.

To measure how well Woodpecker's path pruning algorithm eliminates infeasible paths, we consider its output from the experiments with a single permission, `android.permission.SEND_SMS`. In particular, we run only the possible-paths portion of the algorithm (i.e., with no pruning) and identify how many entry points *might* contain a path to a dangerous capability. Our results show that for each phone, Woodpecker will report more than 8K possible paths. This surprisingly large number is due to the conservative approach we have taken in resolving an ambiguous reference to assignable classes. Fortunately, our re-run of the full system by pruning the infeasible paths immediately brings the number to the single digits. Specifically, our system only reports capability leaks in the HTC phones, especially 2, 3, 2 for the HTC Legend, EVO 4G, and Wildfire S respectively. Among the reported leaks, we then manually verify the correctness of the pruned paths. The results show they are all valid with no false positives. Note that the presence of one single path is sufficient to leak

Vendor	Model	Processing Time	# Apps
HTC	Legend	3366.63s	125
	EVO 4G	4175.03s	160
	Wildfire S	3894.37s	144
Motorola	Droid	2138.38s	76
	Droid X	3311.94s	161
Samsung	Epic 4G	3732.56s	138
Google	Nexus One	2059.47s	76
	Nexus S	1815.71s	72

Table 5: Processing Time of Examined Smartphone Images

the related capability. We do not measure false negatives due to the lack of ground truth in the tested phone images. However, because of the conservative approach we have been taking in our prototype, we are confident in its low false negatives.

For the processing time, we measure them directly by running our system multiple times over the tested smartphone images. We analyze each image ten times on an AMD Athlon 64 X2 5200+ machine with 2GB of memory and a Hitachi HDP72502 7200 rpm hard drive. The mean of these results are summarized in Table 5. Each phone image took at most a little over an hour to process. We believe the average time ($\sim$ 51.0 minutes) per image to be reasonable given the offline nature of our tool, which has not yet been optimized for speed.

6. DISCUSSION

Our system has so far uncovered a number of serious capability leaks in current smartphones from leading manufacturers. Given this, it is important to examine possible root causes and explore future defenses.

First of all, capability leaks essentially reflect the classic confused deputy attack [20] where one app is tricked by another into improperly exercising its privileges. Though one may easily blame the manufacturers for developing and/or including these vulnerable apps on the phone firmware, there is no need to exaggerate their negligence. Specifically, the permission-based security model in Android is a capability model that can be enhanced to mitigate these capability leaks. One challenge however is to maintain the integrity of those capabilities when they are being shared or opened to other unrelated apps. In other words, either the capability-leaking app needs to ensure that it will not accidently expose its capability without checking the calling app's permission or the underlying Android framework will diligently mediate app interactions so that the integrity will be maintained to not leak the capability. However, the interaction among apps are usually application-specific, and their semantic information could be hard for the Android framework to obtain.

Second, to avoid unsafely exposing capabilities, we can also develop a validator tool and release it together with the Android SDK. Note that such a validator tool needs to accommodate various ways on how the permission model is implemented in Android. Specifically, Android uses string identifiers to represent permissions, and permission information can be encoded in either the app's manifest or code, which indicates that the permission model cannot be considered type-safe. Accordingly, conventional Java source code analysis tools are not aware of the impact permissions have on program execution. To the best of knowledge, we are not aware of any other tool that is designed to identify potential capability leaks within an app.

Woodpecker represents our first step towards such a validator tool for capability leak detection. Though it has identified serious capability leaks in current Android phones, it still has a number of limitations that need to be addressed. For example, other than tightening the underlying implementation and incorporating latest development of accurate, scalable points-to analysis [10, 31, 32], our system needs to expand the coverage to accommodate native code. (Our prototype only works for Dalvik bytecode, not the actual machine code.) Also, there is a lack of support in our current system when dealing with one advanced class of capability leaks: *chained capability leaks*. To illustrate, consider three apps A, B, and C. App C has a CALL_PHONE capability, which it safely exposes to B by defining a new permission, say MY_CALL_PHONE. The new permission is acquired by B. For a chained leak to occur, B opens up the new permission unsafely to A. As a result, there is a call chain A->B->C, which could leak the CALL_PHONE capability. Moreover, as the new permission MY_CALL_PHONE can be arbitrary and specific to an particular app implementation, there is a need to explore innovative ways to extend our current prototype to accommodate such chained capability leaks.

Finally, our study only examines capability leaks among pre-loaded apps in the phone firmware. We also expect the leaks could occur among third-party user apps. Note that phone images are relatively homogeneous and static with usually a somewhat infrequent update schedule. Capability leaks, especially explicit ones, on phone images are of great interest to malicious third parties. Implicit leaks, on the other hand, appear to be relatively rare, which we assume are more software engineering defects than a real security threat. However, for third-party apps, implicit leaks could constitute collusion attacks that directly undermine the app market model. Specifically, app markets do not report the actual permissions *granted* to an app. Instead they report only the permissions an app requests or embodied in the manifest file. As a result, a cohort of seemingly innocuous apps could conspire together to perform malicious activities and the user may not be informed of the true scope of their permissions within the system. Meanwhile, we hypothesize that explicit leaks in user-installed apps may be less common and useful, as an app must have both a sizable installed base and unwittingly expose some interesting functionality in order for an attacker to derive much benefit from exploiting the leaked capabilities. In future work, we plan to apply Woodpecker to assess the threat posed by capability leaks in user apps.

7. RELATED WORK

Smartphones have recently attracted considerable attention, especially in the context of privacy. Accordingly, much work has been devoted to analyzing smartphone apps, either statically or dynamically. For example, TaintDroid [16] applies dynamic taint analysis to monitor information-stealing Android apps. Specifically, by explicitly modeling the flow of sensitive information through Android, TaintDroid raises alerts when any private data is going to be transmitted from the device. A follow-up work [17] developed a Dalvik decompiler ded to statically uncover Java code from the Dalvik bytecode in popular free Android apps. The uncovered Java code is then fed into existing static analysis tools to understand or profile the app behavior. Woodpecker is different from these efforts with its unique focus on statically analyzing pre-loaded apps in smartphone firmware to uncover possible capability leaks.

From another perspective, researchers have also developed static analysis tools for privacy leaking detection. For example, PiOS [15] is a representative example, which constructs a control-flow graph for an iOS app and then looks for the presence of information-leaking execution through that graph. Specifically, PiOS tries to link sources of private information to network interfaces. In comparison, Woodpecker was developed for the Android platform and thus needs to overcome platform-level peculiarities for the control-flow construction and data flow analysis (Section 3). Most importantly, Woodpecker has a different goal in uncovering unsafe exposure of dangerous capability uses, including both explicit and implicit ones. In the same vein, work by Chaudhuri et al. [11, 19] formalizes data flow on Android so that a data flow policy can be formally specified for an Android app, which can then be checked against the app code to ensure compliance. A SCanDroid system [19] has been accordingly developed to extract such specifications from the app's manifests that accompany such applications, and check whether data flows through the app are consistent with the specification. Note that SCanDroid requires accessing the app's Java source code for the analysis, which is not available in our case for capability leak detection.

On the defensive side, TISSA [37] argues for a privacy mode in Android to tame information-stealing apps. Kirin [18] attempts to block the installation of apps that request certain combinations of permissions with deleterious emergent effects. A development of that system, Saint [28], empowers the app developer to specify additional constraints on the assignment of permissions at install-time and their use at runtime. Apex [26] modifies the permission framework to allow for selectively granting permissions and revoking permissions at runtime. MockDroid [9] allows privacy-sensitive calls to be rewritten to return "failure" results. In the .NET framework, Security by Contract [13] allows an application's behavior to be constrained at runtime by a contract. Such contract-based systems might represent a defense against implicit capability leaks, though none of these share the same goal of exposing the capability leaks in smartphone firmware.

As discussed earlier, capability leaks essentially reflect the confused deputy attack [20]. Other researchers also warn of similar attacks in Android [14, 12, 27]. For example, Davi *et al.* [12] show a concrete confused deputy attack against the Android Scripting Environment. QUIRE [14] allows apps to reason about the call-chain and data provenance of requests, which could be potentially helpful in mitigating this attack. Nils [27] manually analyzed the HTC Legend's system image looking for possible permission abuses. In comparison to these efforts, our work develops a semi-automated system to systematically detect such capability leaks. In fact, only manual effort required by Woodpecker comes from verifying the detected leaks. Moreover, note that some Android malware such as Soundcomber [29] were developed by requesting certain Android permissions. Our research shows that these requests could be potentially avoided as the permissions might have already been leaked (e.g., RECORD_AUDIO).

More generally, a number of systems that target desktop apps have been developed to detect system-wide information flow or confine untrusted app behavior. For example, TightLip [35] treats a target process as a black box. When the target process accesses sensitive data, TightLip instantiates a sandboxed copy, gives fuzzed data to the sandboxed copy and runs the copy in parallel with the target for output comparison and leak detection. Privacy Oracle [24] applies a differential testing technique to detect the correlation or likely leaks between input perturbations and output perturbations of the application. Also, system-level approaches such as Asbestos [33], HiStar [36], Process Coloring [23], and PRECIP [34] instantiate information flow at the process level by labeling running processes and propagating those labels based on process behavior. While we expect some of these approaches will be applicable on resource-constrained mobile phone environments, they are more focused on detecting information leaks instead of capability leaks (and their applicability to the smartphone setting still remains to be demonstrated).

8. CONCLUSIONS

In this paper, we present a system called Woodpecker to examine how the Android-essential permission-based security model is enforced on current leading Android-based smartphones. In particular, Woodpecker employs inter-procedural data flow analysis techniques to systematically expose possible capability leaks where an untrusted app can obtain unauthorized access to sensitive data or privileged actions. The results are worrisome: among the 13 privileged permissions examined so far, 11 were leaked, with individual phones leaking up to eight permissions. These leaked capabilities can be exploited to wipe out the user data, send out SMS messages (e.g., to premium numbers), record user conversation, or obtain the user's geo-location data on the affected phones – all *without* asking for any permission.

9. REFERENCES

[1] Android. http://www.android.com/.

[2] Android Debug Bridge. http://developer.android.com/guide/developing/tools/adb.html.

[3] Android Permission Protection Levels. http://developer.android.com/reference/android/R.styleable.html#AndroidManifestPermission_protectionLevel.

[4] Android Security and Permissions. http://developer.android.com/guide/topics/security/security.html.

[5] Apple App Store. http://www.apple.com/iphone/apps-for-iphone/.

[6] Dalvik. http://sites.google.com/site/io/dalvik-vm-internals/.

[7] IPhone Stored Location in Test Even if Disabled. http://online.wsj.com/article/SB10001424052748704123204576283580249161342.html.

[8] Vulnerability in HTC Peep: Twitter Credentials Disclosure. http://seclists.org/fulldisclosure/2011/Feb/49.

[9] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: trading privacy for application functionality on smartphones. In *HotMobile '11: Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, May 2011.

[10] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 555–565, November 2009.

[11] A. Chaudhuri. Language-Based Security on Android. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–7, February 2009.

[12] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 3th Information Security Conference*, October 2010.

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

[14] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[15] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, February 2011.

[16] 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 USENIX Symposium on Operating Systems Design and Implementation*, pages 1–6, February 2010.

[17] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[18] 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*, pages 235–245, February 2009.

[19] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. http://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf.

[20] N. Hardy. The Confused Deputy, or Why Capabilities Might Have Been Invented. In *ACM Operating Systems Review*, volume 22, pages 36–38, 1988.

[21] J. Hildenbrand. 150,000 apps in Android Market, tripled in 9 months. http://www.androidcentral.com/150k-apps-android-market-tripled-9-months.

[22] IDC. Android Rises, Symbian 3 and Windows Phone 7 Launch as Worldwide Smartphone Shipments Increase 87.2% Year Over Year. http://www.idc.com/about/viewpressrelease.jsp?containerId=prUS22689111.

[23] X. Jiang, A. Walters, D. Xu, E. H. Spafford, F. Buchholz, and Y.-M. Wang. Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, 2006.

[24] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 279–288, 2008.

[25] K. Mahaffey and J. Hering. App Attack-Surviving the Explosive Growth of Mobile Apps. https://media.blackhat.com/bh-us-10/presentations/Mahaffey_Hering/Blackhat-USA-2010-Mahaffey-Hering-Lookout-App-Genor
pdf.

[26] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332, April 2010.

[27] Nils. The Risk you carry in your Pocket. https://media.blackhat.com/bh-ad-10/Nils/Black-Hat-AD-2010-android-sandcastle-slides.pdf.

[28] M. Ongtang, S. E. McLaughlin, W. Enck, and P. D. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference*, pages 340–349, December 2009.

[29] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, pages 17–33, February 2011.

[30] R. Siles. The Seven Deadly Sins of Security Vulnerability Reporting, 2010. http://blog.taddong.com/2010/08/seven-deadly-sins-of-security.html.

[31] M. Sridharan and R. Bodík. Refinement-Based Context-Sensitive Points-To Analysis for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, pages 387–400, June 2006.

[32] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 32–41, January 1996.

[33] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and Event Processes in the Asbestos Operating System. *ACM Transactions on Computer Systems*, 25, December 2007.

[34] X. Wang, Z. Li, J. Y. Choi, and N. Li. PRECIP: Practical and Retrofittable Confidential Information Protection Against Spyware Surveillance. In *Proceedings of the 16th Network and Distributed System Security Symposium*, 2008.

[35] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping Applications from Spilling the Beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, 2007.

[36] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[37] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST 2011)*, June 2011.