MIT Libraries | DSpace@MIT

MIT Open Access Articles

Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Bagheri, Hamid, et al. "Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification." Proceedings of FM 2015: Formal Methods, edited by Nikolaj Bjørner and Frank de Boer, vol. 9109, Springer International Publishing, 2015, pp. 73–89.

As Published: http://dx.doi.org/10.1007/978-3-319-19249-9_6

Publisher: Springer Nature America, Inc

Persistent URL: https://hdl.handle.net/1721.1/121239

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

Detection of Design Flaws in the Android Permission Protocol through Bounded Verification

Hamid Bagheri[1,2], Eunsuk Kang[1], Sam Malek[2], and Daniel Jackson[1]

[1] Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
[2] Department of Computer Science, George Mason University

Abstract. The ever increasing expansion of mobile applications into nearly every aspect of modern life, from banking to healthcare systems, is making their security more important than ever. Modern smartphone operating systems (OS) rely substantially on the permission-based security model to enforce restrictions on the operations that each application can perform. In this paper, we perform an analysis of the permission protocol implemented in Android, a popular OS for smartphones. We propose a formal model of the Android permission protocol in Alloy, and describe a fully automatic analysis that identifies potential flaws in the protocol. A study of real-world Android applications corroborates our finding that the flaws in the Android permission protocol can have severe security implications, in some cases allowing the attacker to bypass the permission checks entirely.

1 Introduction

Modern mobile devices provide a framework for multiple applications to interact with each other by exporting and invoking APIs. From a security and privacy perspective, some of the resources shared through the APIs may be considered more critical than others; for example, an ability to send a text message is more dangerous than an ability to change the ringtone on the phone. Therefore, a mechanism that can be used by the developer to control access to critical resources is essential.

Popular operating systems such as Android, iOS, and Windows Phone implement a *permission-based* model for controlling the types of resources that each application is allowed to access. In this model, a developer protects a critical resource inside an application by assigning an explicit permission, which must be obtained by any application that wishes to access the resource. Permissions are typically granted to an application at the discretion of the end user, who makes a decision based on the perceived trustworthiness of the application.

In recent years, researchers have identified a number of flaws in the permission mechanisms that lead to serious security and privacy breaches [1,2,3,4,5,6]. The typical manner in which these problems are discovered involves a careful

scrutiny by security experts, sometimes long after these devices are released. Many issues are overarching design flaws that require system-wide reasoning—not easily attainable through conventional analysis methods such as testing and static analysis, which are more suited for detecting bugs in individual parts of the system.

Just as techniques in formal methods have proven practical in assessing the security of network protocols [7], we believe that building a formal model of a permission protocol and performing a rigorous analysis can identify potential vulnerabilities and candidate fixes. This paper, unlike prior studies of Android security (including ours [8]) that leverage code analyses to check a particular application for vulnerabilities, instead focuses on modeling and analyzing the Android permission protocol for design flaws. Our model is written in Alloy [9], a language based on a first-order relational logic, with an analysis engine that performs bounded verification of models. As far as we are aware, our work is the first that describes an *automated* analysis of the Android permission protocol.

Through an analysis of our model, we identified a number of vulnerabilities in the protocol that allow a malicious application to entirely bypass permission checks. In particular, we performed a study of a vulnerability that has not been studied in the security literature before—called the *custom permission vulnerability.* To confirm that an abstract attack scenario identified during the analysis is indeed realistic, we demonstrated the attack on concrete Android applications across different versions of Android. Through our study, we show that the custom permission vulnerability is widespread, and that many popular applications are, in fact, susceptible to this type of attacks.

The rest of the paper is structured in the following way. We begin by giving a brief background on Android and motivating why securing its permission protocol can be a challenging task (Section 2). We then describe a formal model of the permission protocol in Alloy (Section 3) and an automated security analysis of the model (Section 4). We present an experiment to demonstrate the feasibility and prevalence of the custom permission vulnerability in existing Android applications (Section 5). Finally, we discuss the related work (Section 6) and conclude with future work (Section 7).

2 Background and Motivation

An *application* is the primary unit of functionality in Android: A typical device is constantly running numerous applications to support the user's needs, such as a messaging service, a mail client, a navigation application, just to name a few.

The success of Android is in part due to its flexible framework for cross-application communication and sharing. Each application is organized into a set of *components*, which export APIs to other applications, thus enabling reuse of functionality across multiple project and software vendors. For example, the developer of a navigation application may encapsulate its map search functionality into an individual component, and provide it as a service to the rest of the

device. There are four types of components: *service*, *activity*, *broadcast receiver*, and *content provider*, each serving a different purpose.

A potential downside to the open-ended nature of the Android framework is an increased risk for security and privacy breaches. Some components handle information that is considered particularly critical, and so freely sharing these components without discretion may lead to undesirable consequences for the user. For example, the navigation application may not want to release map search histories as part of a component API, since a rogue application could use these data to extrapolate the user's travel pattern for a malicious purpose.

Android uses a permission-based mechanism to control how applications interact with each other. Before an application can access a component, it must be granted an explicit *permission* to do so by the user. Each permission is associated with a *protection level*, which indicates the trustworthiness of an application that may be granted this permission. There are three types of protection levels: (1) *normal*, meaning the permission is granted to every application, (2) *dangerous*, granted only at the discretion of the device user, and (3) *signature*, granted only to applications from the same developer[3]. A runtime engine monitors every invocation of an API operation and ensures that the calling application has the permission to perform that operation.

An Android device contains a number of built-in permissions for basic features, such as sending a text message, turning on GPS, and accessing the Internet. In addition, Android allows a third-party application to define *custom permissions* and selectively control access to its components. Typically, permissions are granted to an application at the time of its installation; however, a special type of permissions called *URI permissions* may be temporarily granted and revoked during the lifetime of an application.

The goal of the Android permission protocol is to prevent any *unauthorized access*; that is, each application should be able to access only those components that it is granted permissions for, and no more. Ensuring that the system achieves this goal, however, is a challenging task, especially since it can be difficult to predict all the ways in which a malicious application may attempt to misuse the system. An attack may involve performing a complex but obscure sequence of operations that would unlikely be encountered during normal usage scenarios. Identifying such attacks requires system-wide reasoning, and cannot be easily achieved by conventional analysis methods such as testing and static analysis, which are more suited at detecting defects in individual parts of the system.

Motivated by this challenge, we explored an approach to analyzing the security of the Android permission protocol by constructing a formal model and performing an automated analysis of the model. Two key elements that distinguish our approach from previous studies of Android security are as follows:

– **System-wide dynamic reasoning:** By modeling the behavior of Android in terms of architectural-level operations (such as installing or removing an application) executed over a sequence of discrete time steps, we are able to

[3] A fourth protection level, *signature/system*, also exists but is rarely used, and so, for the purpose of our discussion, will be grouped into *signature*.

perform system-wide reasoning that would be difficult to achieve using static analysis or testing. For example, our analysis can explore all possible orders in which applications are installed and check whether a particular ordering could be exploited by an attacker (which, in fact, turned out to be the key to an actual attack that involved custom permissions).
- **Concretization:** The result of the analysis, performed on an *abstract* model, is used to guide an implementation-level analysis that checks a *concrete* Android application for the presence of a vulnerability.

This approach demonstrates a potential synergy between model-based and code analysis techniques for an *end-to-end* security analysis: A system-level reasoning is first performed on a high-level model of the system, generating information about potential vulnerabilities, each of which can be confirmed for presence in the implementation using techniques such as static analysis, testing, or inspection.

3 Android Permission Model

In this section, we describe a formal model of the Android permission protocol in Alloy [9], a specification language based on a first-order relational logic. Alloy is suitable for this modeling task because (1) its flexible core allows one to model and integrate different aspects of a system, and (2) its backend tool, the Alloy Analyzer, provides an automated analysis for checking assertions and generating counterexamples. However, our approach does not prescribe the use of a particular formalism, and other languages may well be suitable.

Our model is based on the official documentation on Android permissions from Google [10]. Android is a large and complex operating system, and modeling it in its entirety would be infeasible. Thus, we focused on the parts of Android that are relevant to the permission mechanism—how permissions are granted and maintained, and how they constrain the behavior of an application. As a result, other aspects of Android (such as intents) are omitted from this model.

One of the challenges that we encountered during our modeling task was due to the fact that some of the key aspects of the Android permission protocol are *under-specified* in the official documentation. For example, the document fails to describe what happens to the permissions that have already been granted when the application that defines those permissions is uninstalled. To avoid over-specification (and possibly ruling out counterexamples), we deliberately left the corresponding parts of the model under-specified. This was possible because Alloy supports *partial* modeling: It allows parts of the system to be left unspecified, allowing the Alloy Analyzer to explore all alternative behaviors.

Figure 1 shows an abridged version of the model in Alloy[4], divided into three parts: (1) the architecture of an Android device (lines 4-19), (2) the Android per-

[4] The Alloy keyword sig introduces a *signature*, which defines a set of elements in the universe. A signature may contain one or more *fields*, each introducing a relation that maps the elements of the signature to the field expression; for example, field protectionLevel in Permission is a binary relation that maps each Permission object to its protection level (line 25). The keyword extends creates a subtyping relationship

mission scheme (lines 21-26), and (3) system operations that modify or depend on the permissions (lines 28-66).

3.1 Permissions

An Android *device* consists of a number of interacting *applications*, each containing zero or more *components* that may export services to other applications. The set of applications running on a device may change over time as new applications are installed and existing ones are removed. We model the dynamic aspect of the system by using a standard Alloy idiom in which an execution is represented as a sequence of time steps, and each mutable object is associated with a different state in each time step [9]. To do this, we introduce a set of totally ordered elements as signature Time, and add it as the last column of relations that are considered mutable[5]; for example, the field apps uses Time to keep track of the installed applications at each time step (line 6).

An application may use permissions to control access to its components by other applications. Each permission object, shown on line 25, is associated with a name and a protection level, which can take one of the three values: Normal, Dangerous, and Signature (in order of increasing criticality). Permissions can be assigned to an application at two different levels. Each component may be guarded by at most one permission (represented by the field guard on line 17), which must be acquired by an application before being able to access the component. In addition, an application may be assigned its own guard (line 13), which is imposed on every one of its components; when both the application and one of its components have a guard, the component-specific permission takes the priority.

Note that the type of the field guard in both Application and Component is PermName. In other words, the guard does not contain information about the protection level that is intended for the component being accessed. As discussed later in the section, this turns out to be a design flaw in Android that can be exploited by a malicious application for unauthorized access.

In addition to a set of built-in permissions that are available by default on Android, an application developer may create one or more *custom permissions* to protect an application-specific component (lines 7-8). For example, each Android device contains a built-in permission called android.permission.INTERNET, controlling which applications are allowed to use the built-in component that provides Internet access. A third-party navigation application may provide its map search capability as a service to other applications, and define a custom permission called com.myapp.perm.SEARCH_MAP to control its access.

A *content provider* is a type of storage component containing one or more database tables that are identified by *URIs* (line 19)[6]. By default, obtaining a

---

between two signatures; an abstract signature has no elements except those belonging to its extensions, and one sig introduces a signature that contains only one element.

[5] The ordering library in Alloy imposes a total order on an input signature (line 1).

[6] Other types of components—service, activity, and broadcast receiver—can be treated equally as far as permissions are concerned, and are omitted from Figure 1.

```
1  open util/ordering[Time]
2  sig Time {}
3
4  /* Android architecture */
5  one sig Device {
6    apps: Application -> Time,          // currently installed applications
7    builtinPerms: set Permission,       // permissions built into Android
8    customPerms: Permission -> Time }  // currently active custom permissions
9  sig Application {
10   declaredPerms: set Permission,      // custom permission declarations
11   usesPerms: set PermName,            // permissions it intends to use
12   grantedPerms: Permission -> Time,  // permissions currently granted
13   guard: lone PermName,
14   components: set Component }
15 sig Component {
16   app: Application,
17   guard: lone PermName }
18 sig URI {}  // points to a table inside a content provider
19 sig ContentProvider in Component { paths: set URI }
20
21 /* Permission objects */
22 sig PermName {}  -- permission name
23 abstract sig ProtectionLevel {}
24 one sig Normal, Dangerous, Signature extends ProtectionLevel {}
25 sig Permission { name: PermName, protectionLevel: ProtectionLevel }
26 sig URIPermission in Permission { uri: URI }
27
28 /* Invocation operation */
29 pred invoke[t, t': Time, caller, callee: Component] {
30   caller.app + callee.app in Device.apps.t
31   canCall[caller, callee, t]
32   noChanges[t, t'] }
33 pred canCall[caller, callee: Component, t: Time] {
34   guardedBy[callee] in (caller.app.grantedPerms.t).name }
35 fun guardedBy[c: Component]: PermName {
36   {p: PermName | (p = c.guard) or (no c.guard and p = c.app.guard) } }
37 pred noChanges[t, t': Time] {
38   Device.apps.t' = Device.apps.t
39   Device.customPerms.t' = Device.customPerms.t
40   all a : Application | a.grantedPerms.t' = a.grantedPerms.t }
41
42 /* Install operation */
43 pred install[t, t': Time, app: Application] {
44   app not in Device.apps.t
45   Device.customPerms.t' = Device.customPerms.t + newCustomPerms[t,app]
46   grantPermissions[t', app]
47   all a : Application - app | a.grantedPerms.t' = a.grantedPerms.t
48   Device.apps.t' = Device.apps.t + app }
49 fun newCustomPerms[t: Time, app: Application]: set Permission {
50   {p: app.declaredPerms | p.name not in (Device.customPerms.t).name} }
51 pred grantPermissions[t: Time, app: Application] {
52   app.grantedPerms.t.name = app.usesPerms
53   app.grantedPerms.t in Device.customPerms.t + Device.builtinPerms }
54
55 /* Uninstall operation */
56 pred uninstall[t, t': Time, app: Application] {
57   app in Device.apps.t
58   Device.apps.t' = Device.apps.t - app
59   Device.customPerms.t' = Device.customPerms.t - app.declaredPerms
60   all a : Application - app | a.grantedPerms.t' = a.grantedPerms.t }
61
62 /* Event trace definition */
63 fact traces {
64   all t: Time - last | let t' = t.next |
65     some app: Application, c1,c2: Component |
66       install[t, t', app] or uninstall[t, t', app] or invoke[t, c1, c2] }
```

Fig. 1: A snippet of the Alloy model of the Android permission protocol.

permission on a content provider grants access to all of its tables. To allow more fine-grained control, Android provides a special type of permissions called *URI permissions* (line 26), which can be used to grant access to a particular URI inside a content provider.

Finally, an application specifies its intent to access a component by including the name of the associated permission as one of its *uses-permissions* (line 11). When an application is installed, the device determines the set of permissions that should be granted to the application using usesPerms.

3.2 System Behavior

Three types of operations relevant to the Android permission scheme are described in the Alloy model: invoking a component, which succeeds only when the calling application has the appropriate permission, and installing and uninstalling an application, which may modify the custom permissions on the device.

Invoke Operation The operation of a component invoking another component is expressed as predicate invoke (lines 29-32), which evaluates to true if and only if caller successfully invokes callee between time steps t and t'. The predicate is, in turn, defined as a conjunction of three constraints: both caller and callee must belong to some application on the device (line 30), caller must have the permission to access callee (31), and no changes are made to the active permissions during the invocation (32).

The predicate canCall defines what it means for caller to be able to invoke callee at time step t (lines 33-34); that is, caller must possess the permission that guards callee[7]. Note that callee may be guarded by no permission at all (i.e., guardedBy may return an empty set), in which case canCall is trivially satisfied; in other words, a component without a guard can be accessed by any other component.

Recall that a component's guard is simply the name of a permission, and so its protection level, by design, plays no role in determining whether caller should be allowed to invoke callee. While not explicitly stated in the Android documentation, this design decision relies on one critical assumption: If an application possesses a permission to access a component with a certain protection level, then it must have been authorized by the user to do so during its installation. However, as our analysis will reveal, this assumption is false: It is possible for a malicious application to obtain a permission to a component with a high protection level (e.g., dangerous), even though the authorization was intended for a lower protection level (e.g., normal). Section 4 describes this attack in detail.

Install Operation The first constraint in install describes the precondition for the operation: app must not already exist on the device at time t (line 44). The four constraints that follow describe the effect of the operation on the device:

[7] Keywords + and in are union and subset operators, respectively.

– If app declares its own custom permissions, they are added to the device, except those that already exist on the device at time t; function newCustom-Perms describes exactly those new permissions to be added (lines 49-50).
– Every permission that app requests in its usesPerms is granted to the new application by the device (lines 51-53).
– The permissions granted to other applications on the device are unaffected.
– Finally, app is added to the set of existing applications on the device.

Note that the process of granting a permission through the user's approval is implicit in this model; grantPermissions simply sets the granted permissions to those in the application's usesPerms (line 52), without describing how a decision about each permission is made. This modeling choice reflects the rather coarse-grained nature of Android permissions: Unless an application is granted *every* one of its uses-permissions, it will not be installed on the device (i.e., the user has no ability to selectively grant permissions[8].). In other words, the details of how permissions are granted are not relevant to our analysis, because the effect of installation is always the same: Each installed application will possess all of the permissions that it requests.

Uninstall Operation This operation removes the specified application app from the device, as well as all of its associated custom permissions. The permissions granted to every other application remains the same during the operation.

Trace Definition The fact[9] traces defines the behavior of the system as a set of traces that it may produce (lines 62-66). Conceptually, a trace is a sequence of time steps, where between each pair of adjacent steps, t and t', one or more of the system operations takes place[10]. Given this definition, a satisfying instance of the model found by the Alloy Analyzer will correspond to exactly one of the possible traces of the system.

Other Parts Due to limited space, Figure 1 omits details about other aspects of the permission protocol that are present in the full Alloy model, including: different types of components (beside content providers), dynamic allocation and checking of URI permissions, and application signatures. The complete model is available online at our project site[11].

4 Analysis

In this section, we describe an automated analysis to check whether the Android permission protocol, as specified in our model, satisfies its goal of preventing unauthorized access.

[8] While outside the scope of our analysis, previous studies have pointed this out as a major source of usability and privacy issues in Android [1].
[9] An Alloy *fact* is a constraint that holds for every satisfying instance of the model.
[10] This trace definition precludes stuttering, as we did not deem it necessary for this model; however, an operation that represents *noop* could be added to allow it.
[11] `http://sdg.csail.mit.edu/projects/android`

```
1  assert NoUnauthorizedAccess {
2    all t, t' : Time, callee, caller : Component |
3      invoke[t, t', caller, callee] implies authorized[caller,callee,t] }
4
5  // True iff caller is authorized to invoke callee
6  pred authorized[caller,callee: Component, t: Time] {
7    let pname = guardedBy[callee],
8      grantedPerm = caller.app.grantedPerms.t & name.pname,
9      requiredPerm =
10       (callee.app.declaredPerms + Device.builtinPerms) & name.pname |
11         some pname implies
12           equalOrHigher[grantedPerm.protectionLevel,
13                         requiredPerm.protectionLevel] }
```

Fig. 2: Assertions on the Android permission protocol.

An Alloy *assertion* is used to state a property that the model is expected to satisfy. When prompted to check an assertion, the Alloy Analyzer explores all possible behaviors of the system and finds a counterexample, if any, that corresponds to a violation of the assertion. The analysis is *exhaustive but bounded* up to a user-specified scope on the size of the domains: If there is a counterexample within the scope, the analyzer is guaranteed to find it, but absence of a counterexample does not imply the validity of the assertion. In practice, many system flaws can be demonstrated with a small number of objects [11], and if desired, the user can iteratively re-analyze the model with larger scopes to gain further confidence.

An important security property of Android is that every component invocation is *authorized*; that is, when a component invokes another component, the caller must have been granted the permission that was declared by the developer to protect the callee.

This property is formally specified as Alloy assertion NoUnauthorizedAccess in Figure 2. Predicate authorized describes what it means for component caller to be authorized to invoke callee. Its definition relies on two different types of permission: grantedPerm represents the permission that is granted to caller during its installation; requiredPerm, on the other hand, represents the custom permission that was declared specifically to guard callee. Then, caller is considered authorized to invoke callee only if the protection level of grantedPerm is equal to or higher than that of requiredPerm.

4.1 Custom Permission Vulnerability

Analysis When prompted to check the assertion, the Alloy Analyzer returns a counterexample trace that demonstrates how a design flaw in Android may lead to a violation of the property. The analysis was performed with a scope of 5 on the size of each domain, and took approximately 4 seconds to complete[12].

[12] The analysis was performed on a Mac OS X machine with 1.8 GHz Intel Dual Cores and 4GB of RAM.

A visualization of the counterexample is shown in Figure 3. In this trace, Application0 declares a custom permission (Permission1) to guard its component (labeled victim) with the protection level of Signature, meaning that only those applications that share the same signature should be able to access it. A separate, malicious application, Application1, bypasses the signature requirement by exploiting a design oversight in Android: Namely, it allows multiple applications to define custom permissions with the same name, but without a clear specification of which one should take precedence when they have different protection levels.

To carry out this type of attack, Application1 declares its own custom permission (Permission0) with the same name as Permission1 but with the lowest protection level, Normal. The attack comprises of the following three operations:

– Step (a): Application1 is installed before Application0, activating its custom permission (Permission0) with the Normal protection level on the device.
– Step (b): Application0 is installed, but a custom permission with the same name is already active, and so Permission1 is ignored. As a result, Application1 continues to hold the same permission that it was granted in Step (a).
– Step (c): The malicious component inside Application1 is able to access victim, despite not having the same signature as Application0.

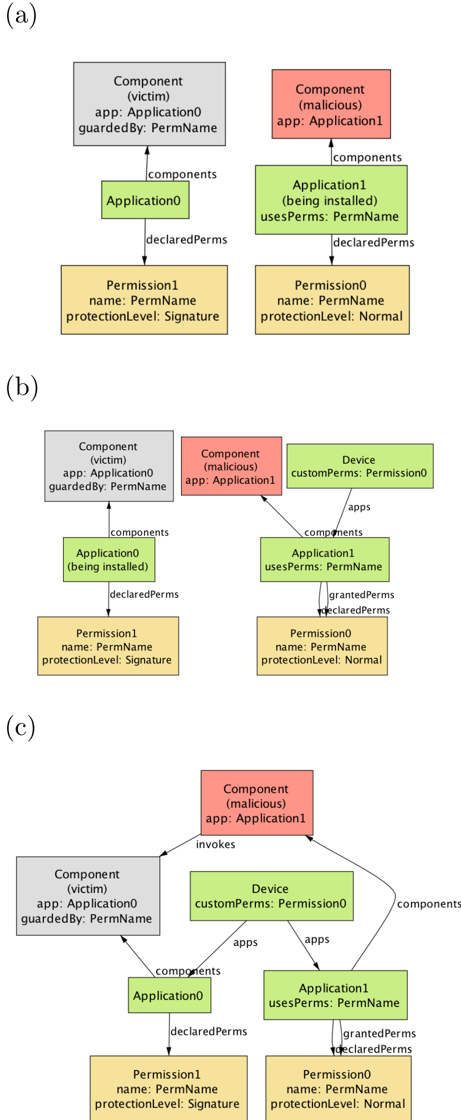

Fig. 3: A counterexample showing an unauthorized access of component victim by malicious Application1 through a custom permission misuse.

Evaluating a Fix One potential fix to this flaw is to disallow multiple applications that define a custom permission

with the same name from simultaneously existing on the device. In our Alloy model, this fix can be expressed by adding the following constraint to the install operation from Figure 1:

```
1  // can't install if a declared perm is named the same as existing one
2  no p : app.declaredPerms | p.name in (Device.customPerms.t).name
```

Re-analyzing the assertion NoUnauthorizedAccess, however, reveals another counterexample. This scenario begins in the same way as the one in Figure 3, where a malicious application (App1) defines its own custom permission with the same name as another permission, but with a lower protection level. Furthermore, another malicious application (App2) that uses this permission is installed. In the next step, App1 is uninstalled, and its associated custom permission is removed from the device. However, Android fails to revoke the same permission from applications that use it (namely, App2), resulting in a dangling permission. When the victim application (App0) is installed, App2 is still able to access the victim component, but with the lower protection level that was defined by App1.

This demonstrates that simply disallowing an installation of applications with duplicate permissions is not sufficient. The uninstall operation must also be amended to ensure that granted permissions are revoked when an application that declares those permissions is uninstalled. This can be done by modifying the constraint on line 60 in Figure 1 as follows:

```
1  all a: Application - app |
2    a.grantedPerms.t' = a.grantedPerms.t - app.declaredPerms
```

4.2 Other Vulnerabilities Found

Our analysis revealed two other types of vulnerabilities in the permission protocol. Due to limited space, we only briefly discuss them here, and refer the reader to our project site for more detail.

URI Permission Flaw A malicious application can obtain a URI permission to a part of a content provider that it is not authorized to access. This vulnerability is due to another flaw in the Android permission protocol: granted URI permissions are not revoked when the associated content provider is uninstalled, leaving dangling permissions that can be exploited for a similar type of attack as in Section 4.1.

To our knowledge, this vulnerability with URI permissions is a previously unknown one. However, further study revealed that the vulnerability exists up to Android version 2.3.7; in newer devices, the URI permissions are revoked during uninstallation, disallowing the attack. Our analysis detected this as a counterexample because the model, reflecting the current Android documentation, was deliberately under-specified with respect to the effect of uninstallation on URI permissions.

Improper Delegation A malicious application may be able to *indirectly* invoke a component, without having a permission to do so, by interacting with a third component that possess the permission. This vulnerability has been identified as the *permission re-delegation* attack in previous work by Felt and her colleagues [12]; our analysis was able to automatically rediscover it.

5 Experiments

A rigorous analysis of a formal model, such as the one described in Section 4, can be used to identify potential flaws at the design level, but by itself does not form a complete security analysis of the system. Instead, the formal analysis must be complemented with a systematic analysis of the concrete system to confirm whether those flaws can lead to *realistic* vulnerabilities, and subsequently attacks.

In this section, we present an experimental study to answer the following two research questions:

- **RQ1:** Can the flaws identified in our formal analysis of Android permission protocol cause an actual attack with serious security consequences?
- **RQ2:** How susceptible are real-world Android applications to security attacks that are due to these flaws in Android permission protocol?

In particular, we focus on the custom permission vulnerability in Section 4.1, as it has not been previously studied in the literature[13]. To address RQ1, we developed demonstrative applications that represent postulated malicious behaviors in the generated counterexample in Figure 3, and observe whether the permission requirement could be bypassed as in the scenario. For RQ2, we performed a study on hundreds of real-world Android applications and quantitatively measured the prevalence of the security vulnerability due to the flaws found in Android permission protocol.

5.1 Demonstration of the Attack

To test the feasibility of the Alloy counterexample in Figure 3, we developed a skeletal address book application that corresponds to the victim application in the trace (cf. Application0 in Fig. 3). Figure 4(a) partially shows an Android manifest file[14] for this application. It defines a custom permission, named ADBOOK_READ, with the *signature* protection level (lines 2–3). This permission is then specified as a guard (in line 7) to protect access to the AddrBookProvider component (lines 4–9), which stores the content of the address book.

[13] The URI permission vulnerability is omitted since it exists only on an outdated version of Android, and the improper delegation flow has already been studied in [12].

[14] A manifest file contains, among other things, declarations of uses and custom permissions for an application.

As declared in its manifest, the AddrBook application does not grant access to its data to any other application. It is thus expected that only applications that explicitly request the ADBOOK_READ permission and are signed with the same signature will be allowed to read the address book contents.

Next, we developed an application that represents postulated malicious behaviors in the Alloy counterexample. Figure 4(b) shows part of the manifest file implementation for MalApp (corresponding to Application1 in Fig. 3). Similar to the address book application, it declares the ADBOOK_READ permission, albeit with a lower protection level, *normal*. It further includes a *uses-permission* element to declare that it requires the self-declared custom permission (lines 15–16). The MalActivity component, which represents the malicious component in the counterexample, then simply sends a query to the AddrBookProvider component.

```
1  //(a) Address book ------------------------------
2  <permission android:name="com.example.ADBOOK_READ"
3    android:protectionLevel="signature" />
4  <application android:label="AddressBook">
5    <provider android:name=".AddressBookProvider"
6     android:authorities=".AddressBookProvider"
7     android:readPermission="com.example.ADBOOK_READ"
8     <!--android:grantUriPermissions="true"-->
9     >
10   </provider>
11 </application>
12 //(b) Custom permission vulnerability-------------
13 <permission android:name="com.example.ADBOOK_READ"
14   android:protectionLevel="normal" />
15 <uses-permission android:name=
16   "com.example.ADBOOK_READ" />
17 <application android:label="MalApp">
18   <activity
19    android:name=".MalActivity"
20    android:label="MalApp" >
21    <intent-filter>
22     <action android:name="MAIN" />
23     <category android:name="LAUNCHER" />
24    </intent-filter>
25   </activity>
26 </application>
```

Fig. 4: Snippets of the demonstrative applications that represent the counterexample scenarios shown in Fig. 3.

The two applications were signed with different keys to reflect a real scenario, where they would be from different developers. We then installed and executed them, according to the counterexample, on two versions of the Android SDK—2.3.7 and 4.4.4—under the Genymotion[15] emulator. We repeated the experiments with different combinations of protection levels for AddressBook and MalApp. In all cases, we observed that MalApp was successfully able to access the content of the address book, confirming the feasibility of the attack.

5.2 Prevalence of the Vulnerability

To estimate the prevalence of this vulnerability among real Android applications, we examined 1,500 applications collected from two repositories: (1) popular free applications from Google's Play Store[16] and (2) open-source applications from the F-Droid repository[17].

An application is at risk of containing a custom permission vulnerability if (1) it defines a custom permission used to protect a component API and (2) it does not implement an *additional,* dynamic check to ensure that the calling application is authorized to access the API. We constructed a custom static analysis

[15] www.genymotion.com
[16] http://play.google.com/store/apps
[17] https://f-droid.org/

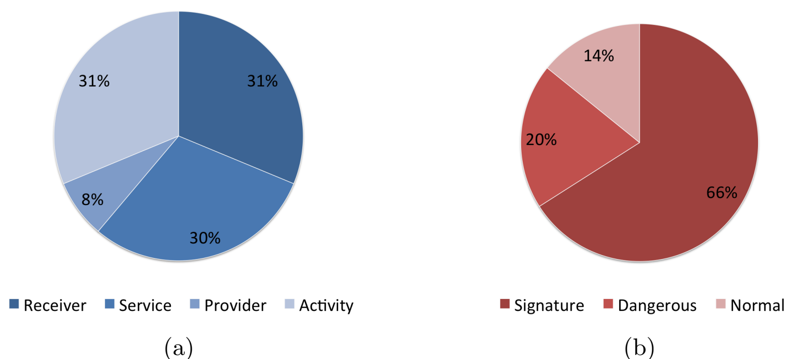

Fig. 5: (a) Frequency of component types protected by custom permissions; (b) Categorization of custom permissions based on their protection levels.

tool to check these two conditions. For each application, our tool decompiles the related Android package file to extract its manifest file. It then pairs the manifest file with the corresponding application's bytecode to perform the following checks:

- **Permission:** The tool checks the manifest file for any declaration of custom permissions, and whether those permissions are actually used to guard components.
- **Dynamic enforcement:** There is a programmatic but limited method for an application to protect itself against the custom permission attack. If it knows a whitelist of trusted calling applications, then it can implement a dynamic check to reject calls from unknown applications (however, it may not be possible to construct such a list for an open-ended application that is designed to interact with many applications). The tool analyzes the bytecode for the presence of this optional check by searching for the use of built-in Android functions such as *getCallingUid*, which returns the caller's information.

Results The total numbers of custom permissions defined within the apps for our Google Play and F-Droid test sets are 536 and 171, respectively. 201 (47.26%) of the apps in our Google Play test set define at least one custom permission, whereas this number is just 67 (6.42%) for the F-Droid repository. The average number of custom permissions per app for those that define at least one custom permission is 2.64. Out of the apps that define custom permissions, 116 (57.71%) apps in the case of Google Play and 45 (67.16%) in the case of F-Droid use those permissions to protect their components. Just under 5% of all the apps in our test set perform the dynamic check.

According to Figure 5(a), about 61% of the components protected by custom permissions are of type Service or Broadcast Receiver. This is important because the lack of a visible user interface in these types of components promotes possibilities for a stealthy permission re-delegation attack [12]. More than 85% of

custom permissions are defined at signature or dangerous protection levels that regulate access to critical APIs, as shown in Figure 5(b).

The results show that custom permissions are widely used by real-world Android applications to guard critical APIs. Most developers do not perform any additional check to ensure that incoming APIs are from trusted callers, suggesting that they may be unaware of the custom permission vulnerability, despite its potential for security breaches.

6 Related Work

The custom permission vulnerability in Section 4.1 was first described in a blog post by an independent security researcher [13]. However, despite its potential security consequences, the vulnerability has not received widespread attention among Android developers; as revealed by our study in Section 5, a significant number of Android applications are still vulnerable to this attack. To our knowledge, the vulnerability has not been studied in the academic literature.

We are aware of two previous works that describe a formalization of the Android permission protocol. Shin and his colleagues encoded a formal model of the protocol in Coq and proved a set of security properties using its interactive theorem proving facility [14]. The main difference between their work and ours is in the kind of analysis performed. A successful Coq proof provides a stronger theoretical guarantee than an Alloy analysis, which is bounded to finite domains in the universe. On the other hand, the Alloy Analyzer is capable of generating counterexamples, which we found tremendously helpful for identifying the vulnerabilities in the system. Even though the properties proven were similar to ours, their analysis failed to identify the custom permission vulnerability, because the definition of the installation operation in their model is over-constrained — their model prevents an application from being installed if it declares a permission that already exists on the device, ruling out behavior that would have revealed the attack.

Fragkaki et al. describe a logical formalization of a permission model similar to the one used in Android [15]. However, they only performed an informal analysis of the model, and did not identify the custom permission vulnerability.

Most of the previous works in Android security involve performing manual inspection or program analysis to identify a particular vulnerability in Android applications [2,4,6,8,16,17,18,19]. Two previous projects deal specifically with permission vulnerabilities in Android. Felt and her colleagues performed a study of existing applications for permission usage and discovered that many of them are "overprivileged" (i.e., given more permissions than they need) [1]. However, their study does not consider custom permissions. In a separate work, Felt et al. describe a type of attack called *permission re-delegation*, and show that many existing Android applications are vulnerable to this type of attack [12].

A number of static analysis tools, such as ComDroid [16], Epicc [17], FlowDroid [19], have been developed to detect a flow of malicious data within an application or between multiple applications. However, these tools do not deal with permission-related vulnerabilities.

More recently, we developed COVERT [8], an approach for compositional analysis of Android inter-application vulnerabilities. COVERT uses static analysis techniques to extract a formal model of Android apps. It then performs the analysis for inter-application vulnerabilities in a modular way, permitting the results of such analyses to be composed to support incremental verification of apps as they are installed, updated, and removed.

These research efforts are mainly focused on analyzing a *particular* application (or a set of apps, in case of COVERT) by extracting relevant security behaviors from it. In contrast, our work focuses on analyzing the *general* underlying Android permission protocol itself, and identifying *design flaws* that may be applicable to all Android applications.

7 Conclusion

In this paper, we presented a formal model of the Android permission protocol in Alloy, and an automated analysis that identified a number of flaws in the protocol that cause serious security vulnerabilities. We also performed a study of one of the vulnerabilities and showed that it is prevalent among many existing Android applications.

It is notable that *underspecification* of the Android permission protocol was essential; it allowed us to avoid specifying aspects of behavior that were not clear in the documentation, and led to the discovery of vulnerabilities that had eluded an earlier analysis of the very same protocol by others (which, due to the use of a theorem prover based on a functional language, had not supported underspecification).

While this paper has focused on the analysis of Android, we believe that our approach can be applied to other types of mobile devices that rely on permissions, such as iOS and Windows Phone. By building a precise model of the permission mechanism and subjecting it to exhaustive analysis, the device designer may be able to discover potential vulnerabilities, instead of relying solely on manual scrutiny by security experts.

We plan to further explore the synergy between formal analysis of a high-level system model and implementation-level techniques, as mentioned in Section 2. We are currently working on an end-to-end security analysis framework that combines a model-based detection of system-level attacks with a suite of static analysis tools that can identify particular types of vulnerabilities; our target domains include web security, mobile devices, and system-of-systems. We believe that our work in this paper presents a first step towards this goal.

Acknowledgment

This work was supported in part by awards D11AP00282 from the US Defense Advanced Research Projects Agency, H98230-14-C-0140 from the US National Security Agency, HSHQDC-14-C-B0040 from the US Department of Homeland Security, and CCF-1252644 from the US National Science Foundation.

References

1. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: 18th ACM Conference on Computer and Communications Security (CCS). (2011) 627–638
2. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on android. In: Proceedings of the 13th international conference on Information security (ISC). (2010)
3. Pandita, R., Xiao, X., Yang, W., Enck, W., Xie, T.: Whyper: Towards automating risk assessment of mobile applications. In: Proceedings of the 22Nd USENIX Conference on Security. SEC'13, Berkeley, CA, USA, USENIX Association (2013) 527–542
4. Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic detection of capability leaks in stock android smartphones. In: Proceedings of the 19th Annual Symposium on Network and Distributed System Security. (2012)
5. Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A stealthy and context-aware sound trojan for smartphones. In: Proc. of 18th Annual Network and Distributed System Security Symposium (NDSS). (2011)
6. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: Proc. of USENIX. (2011)
7. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM Comput. Surv. **41**(4) (October 2009) 19:1–19:36
8. Bagheri, H., Sadeghi, A., Garcia, J., Malek, S.: Covert: Compositional analysis of android inter-app permission leakage. IEEE Transactions on Software Engineering (2015)
9. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. Second edn. MIT Press (2012)
10. Google: Android system permissions. `http://developer.android.com/guide/topics/security/permissions.html`
11. Andoni, A., Daniliuc, D., Khurshid, S., Marinov, D.: Evaluating the small scope hypothesis. `http://sdg.csail.mit.edu/pubs/2002/SSH.pdf`
12. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission redelegation: Attacks and defenses. In: 20th USENIX Security Symposium. (2011)
13. Mark Murphy: Vulnerabilities with custom permissions. `http://commonsware.com/blog/2014/02/12/vulnerabilities-custom-permissions.html` (2014)
14. Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A formal model to analyze the permission authorization and enforcement in the android framework. In: IEEE International Conference on Privacy, Security, Risk and Trust. (2010) 944–951
15. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing android's permission system. In: 17th European Symposium on Research in Computer Security (ESORICS). (2012) 1–18
16. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services. MobiSys '11, New York, NY, USA, ACM (2011) 239–252
17. Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Traon, Y.L.: Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In: Proceedings of the 22nd USENIX Security Symposium, Washington, DC (August 2013)

18. Enck, W., Gilbert, P., Chun, B.g., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proc. of USENIX OSDI. (2011)
19. Arzt, S., Rasthofer, S., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014). (2014)