https://doi.org/10.1007/s00165-017-0445-z
BCS © 2017
Formal Aspects of Computing (2018) 30: 525–544

Formal Aspects of Computing

A formal approach for detection of security flaws in the android permission system

Hamid Bagheri[1], Eunsuk Kang[2], Sam Malek[3], and Daniel Jackson[2]

[1] Department of Computer Science and Engineering, University of Nebraska, Lincoln, NE, USA
[2] Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA
[3] School of Information and Computer Sciences, University of California, Irvine, Irvine, CA, USA

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.

Keywords: Android; Permission protocol; Alloy; Verification.

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.

Correspondence and offprint requests to: H. Bagheri. E-mail: bagheri@unl.edu.

In recent years, researchers have identified a number of flaws in the permission mechanisms that can lead to serious security and privacy breaches [FCH+11, DDSW10, PXY+13, GZWJ12a, SZZ+11, EOMC11a]. 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.

In this paper, we provide a comprehensive description of a formal analysis of the permission protocol implemented in the Android framework that was first introduced in our prior work [BKMJ15]. Just as techniques in formal methods have proven practical in assessing the security of network protocols [WLBF09], 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 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 [Jac12], 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 three types of vulnerabilities in the protocol that allow a malicious application to entirely bypass permission checks: (1) *URI permission vulnerability*, in which a malicious application can procure a URI permission to a part of a content provider[1] that it is not authorized to access. This vulnerability was previously unknown, and our further study confirmed the existence of the vulnerability on various versions of the Android framework up to version 2.3.7. (2) *Improper delegation vulnerability*, in which a malicious application without being granted a required permission can indirectly access a protected resource, via interacting with another component that possesses the required permission. Our analysis automatically rediscovered this vulnerability that has previously been revealed by Felt and her colleagues [FWM+11]. (3) *Custom-permission vulnerability*, in which a malicious application can illegally access the resources of another application, protected by a custom permission. This vulnerability is rooted in a design flaw that our automated analysis discovered in the Android permission protocol: If two applications declare the same custom permission, whichever application is installed first is the one whose definition is used. This flaw has been described earlier, albeit in a blog post [Mar14]. Yet, despite its significance, neither the vulnerability nor its security consequences have never been discussed in the security literature before. To confirm that an abstract attack scenario identified during the formal 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.

This paper describes several new non-trivial extensions to the preliminary version of our work described in [BKMJ15]. (1) We have expanded our Alloy model of the Android framework to capture the newly added dynamic aspects of the Android permission protocol. More specifically, in the latest release of the Android framework (i.e., version 6.0 or Marshmallow), the static nature of the Android permission protocol has been changed. In other words, an application now may obtain permissions from the user not only during its installation, but during its runtime as well. Additionally, the user may choose to revoke a previously granted permission from the application at any time she wishes. (2) We report on new experiments to assess the scalability of our permission protocol analysis through measuring how variation in size of the finite domains affects the computational time required for bounded verification of the model. On top of these technical contributions, the paper provides an in-depth description of the bounded verification of the Android permission protocol, a detailed discussion on the Alloy language and its analyzer that makes the automated analysis possible, and a revamped discussion of this work in the context of related research.

The rest of this 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 (Sect. 2). We then describe a formal model of the permission protocol in Alloy (Sect. 3) and an automated security analysis of the model (Sect. 4). We present an experiment to demonstrate the feasibility and prevalence of the custom permission vulnerability in existing Android applications (Sect. 5). Finally, we discuss the related work (Sect. 6) and conclude with future work (Sect. 7).

[1] Content provider components provide storage capabilities to other components.

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 [BGS+16]. 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 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.[2] A runtime engine inside the OS 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 be unlikely to 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-level 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 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).
- *Guided implementation-level analysis* While rigorous analyses of a formal, yet *abstract*, model of the permission protocol helps identify potential flaws at the system level, one more step is needed to form an end-to-end security analysis of the system. In particular, the formal analysis must be backed with an implementation-level analysis of the concrete system to confirm whether the flaws identified can result in *realistic* exploits with serious security consequences. The result of the formal analysis is used at this step to guide the implementation-level analysis.

[2] 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*.

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 [Jac12], a specification language based on a first-order relational logic. Our model is based on the official documentation on Android permissions from Google [Goo]. 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.

In the rest of this section, we first provide a brief overview of Alloy, and then describe a formal model of the permission protocol in Alloy.

3.1. Alloy overview

Alloy is a lightweight formal specification language based on a first-order relational logic with transitive closure [Jac12]. 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.

Figure 1 shows an abridged version of the model in Alloy, divided into three parts: (1) the architecture of an Android device (lines 3–21), (2) the Android permission scheme (lines 22–27), and (3) system operations that modify or depend on the permissions (lines 28-66).

In Alloy, essential data types that collectively define the vocabulary of a model are introduced using *signatures* (`sig`). 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 26). A field may be associated with an optional *multiplicity* operator, including `set` (*zero or more*) and `lone` (*at most one*); by default, if no multiplicity is specified, *exactly one* is assumed. The keyword `extends` creates a subtyping relationship 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.

The other essential constructs of the Alloy language include: *facts*, *predicates*, *functions* and *assertions*. Facts (`fact`) are formulas that take no arguments, and define constraints that must always hold. Predicates (`pred`) are named logical formulas used in defining parameterized and reusable constraints that are evaluated to be either true or false. Functions (`fun`) are parameterized expressions; a function, similar to a predicate, can be invoked by instantiating its parameters, but it instead returns a relational expression. An assertion (`assert`) is a formula describing an expected property of the model; when prompted, the Alloy Analyzer will attempt to generate a counterexample that refutes the assertion.

In addition, the Alloy language provides a small core of logical and relational operators. The dot (.) and tilde (∼) operators denote a relational join of two relations and the transpose operation over a binary relation, respectively. For example, given a particular permission object `p` and its field `protectionLevel`, the join expression `p.protectionLevel` yields the protection level that is associated with that permission. The transitive closure (ˆ) of a relation is the smallest enclosing relation that is transitive; the reflexive-transitive closure (*) is the smallest enclosing relation that is both transitive and reflexive.

The Alloy Analyzer is a constraint solver that supports automatic analysis of models written in Alloy. It can be used to find a satisfying instance representing a possible scenario in the specified system, or a counterexample to a given assertion. The analysis process involves a translation of an Alloy specification into a Boolean formula in conjunctive normal form (CNF), which is then solved using an off-the-shelf SAT solver.[3] The Alloy Analyzer is a *bounded* verifier; it exhaustively explores all possible behaviors of a system, but only up to a user-specified *scope* on the size of domains. Thus, the lack of a counterexample detected by the analyzer does not necessarily imply that the assertion is valid, as there may exist a counterexample beyond the given scope. However, for increased confidence in its result, the user may perform the analysis again with a larger scope.

We will introduce additional details of the Alloy language as necessary to present our formal model of the Android permission protocol. For further information about Alloy, we refer the interested reader to [Jac12].

3.2. 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 [Jac12]. To encode this idiom, we introduce a set of totally ordered elements as signature `Time`, and add it as the last column of relations that are considered mutable;[4] for example, the field `apps` uses `Time` to keep track of the installed applications at each time step (line 5).

An application may use permissions to control access to its components by other applications. Each permission object, shown on line 26, 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 18), 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 (line 7). 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 21).[5] By default, obtaining a 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 27), which can be used to grant access to a particular URI inside a content provider.

[3] Alloy relies on a finite model finder called Kodkod [TJ07] as an intermediate language before translation to CNF.

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

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

```
open util/ordering[Time]
sig Time {}
/* Android architecture */
one sig Device {
  apps: Application -> Time,          // currently installed applications
  builtinPerms: set Permission,       // permissions built into Android
  customPerms: Permission -> Time     // currently active custom permissions
}
sig Application {
  declaredPerms: set Permission,      // custom permission declarations
  usesPerms: set PermName,            // permissions it intends to use
  grantedPerms: Permission -> Time,   // permissions currently granted
  guard: lone PermName,
  components: set Component
}
sig Component {
  app: Application,
  guard: lone PermName
}
sig URI {}  // points to a table inside a content provider
sig ContentProvider in Component { paths: set URI }
/* Permission objects */
sig PermName {}  -- permission name
abstract sig ProtectionLevel {}
one sig Normal, Dangerous, Signature extends ProtectionLevel {}
sig Permission { name: PermName, protectionLevel: ProtectionLevel }
sig URIPermission in Permission { uri: URI  }
/* Invocation operation */
pred invoke[t, t': Time, caller, callee: Component] {
  caller.app + callee.app in Device.apps.t
  canCall[caller, callee, t]
  noChanges[t, t']
}
pred canCall[caller, callee: Component, t: Time] {
  guardedBy[callee] in (caller.app.grantedPerms.t).name
}
fun guardedBy[c: Component]: PermName {
  {p: PermName | (p = c.guard) or (no c.guard and p = c.app.guard) }
}
pred noChanges[t, t': Time] {
  Device.apps.t' = Device.apps.t
  Device.customPerms.t' = Device.customPerms.t
  all a : Application | a.grantedPerms.t' = a.grantedPerms.t
}
/* Install operation */
pred install[t, t': Time, app: Application] {
  app not in Device.apps.t
  Device.customPerms.t' = Device.customPerms.t + newCustomPerms[t,app]
  grantPermissions[t', app]
  all a : Application - app | a.grantedPerms.t' = a.grantedPerms.t
  Device.apps.t' = Device.apps.t + app
}
fun newCustomPerms[t: Time, app: Application]: set Permission {
  {p: app.declaredPerms | p.name not in (Device.customPerms.t).name}
}
pred grantPermissions[t: Time, app: Application] {
  app.grantedPerms.t.name = app.usesPerms
  app.grantedPerms.t in Device.customPerms.t + Device.builtinPerms
}
/* Uninstall operation */
pred uninstall[t, t': Time, app: Application] {
  app in Device.apps.t
  Device.apps.t' = Device.apps.t - app
  Device.customPerms.t' = Device.customPerms.t - app.declaredPerms
  all a : Application - app | a.grantedPerms.t' = a.grantedPerms.t
}
/* Event trace definition */
fact traces {
  all t: Time - last | let t' = t.next |
    some app: Application, c1,c2: Component |
      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

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`.

3.3. 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.

Using the standard Alloy idiom for modeling dynamic system behavior, we encode each operation as a predicate that describes a relationship between the states of the system before (`t`) and after (`t'`) the operation takes place. Conceptually, one can regard the operation as having successfully taken place for a particular set of parameters if the corresponding predicate evaluates to true over those same parameters. A predicate is a conjunction of constraints, some of which may correspond to preconditions for the operation, while others describe its postconditions or frame conditions (which are used to ensure that only the intended parts of the system are modified); like in Z [WD96], no distinction is made between these conditions in Alloy.

Invoke operation The operation of a component invoking another component is expressed as predicate `invoke` (lines 29–33), 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 34–36); that is, `caller` must possess the permission that guards `callee`.[6] 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 47). The four constraints that follow describe the effect of the operation on the device:

- 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 `newCustomPerms` describes exactly those new permissions to be added (lines 53–55).
- Every permission that `app` requests in its `usesPerms` is granted to the new application by the device (lines 56–59).
- 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 49), without describing how a decision about each permission is made. This modeling choice reflects the rather coarse-grained nature of Android permissions:

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

```
/* How permissions are granted in Android 6 */
pred grantPermissions[app : Application, t : Time] {
  let normalPerms = allDevicePerms[t] & protectionLevel.Normal,
    signaturePerms = allDevicePerms[t] & protectionLevel.Signature {
      app.grantedPerms.t.name in app.usesPerms
      // in Android version 6.0 or higher,
      // only Normal and Signature permissions are granted during install
      // (Signature perms are granted only if the app's signature matches that of
      //  the other app that declares those permissions)
      app.grantedPerms.t in
        normalPerms +
        { s : signaturePerms | let declaringApp = declaredPerms.s & Device.apps.t |
          declaringApp.signature = app.signature }
    }
}
/* Returns all permissions available on the device at t */
fun allDevicePerms[t : Time] : set Permission {
  Device.builtinPerms + Device.customPerms.t
}
/* Operation for requesting dangerous permissions during runtime */
pred requestPerms[t, t': Time, app: Application, perms : set PermName] {
  // perms must appear in the manifest
  perms in app.usesPerms
  // perms must not already have been granted
  no perms & (app.grantedPerms.t).name
  // perms should have protection level of "dangerous"
  all pn : perms | some p :  allDevicePerms[t] & name.pn | p.protectionLevel = Dangerous
  // once the user grants the permission, then they are added to the app's grantedPerms
  app.grantedPerms.t' = app.grantedPerms.t + allDevicePerms[t] & name.perms
  // frame conditions
  Device.customPerms.t' = Device.customPerms.t
  Device.apps.t' = Device.apps.t
  // no changes to other apps
  all a : Application - app | a.grantedPerms.t' = a.grantedPerms.t
}
/* Operation for revoking previously granted permissions */
pred revokePerms[t, t' : Time, app: Application, perms: set PermName] {
  // perms must already have been granted
  perms in (app.grantedPerms.t).name
  // remove the permissions
  app.grantedPerms.t' = app.grantedPerms.t - allDevicePerms[t] & name.perms
  // frame conditions
  Device.customPerms.t' = Device.customPerms.t
  Device.apps.t' = Device.apps.t
  // no changes to other apps
  all a : Application - app | a.grantedPerms.t' = a.grantedPerms.t
}
/* New event trace definition */
fact traces {
  all t: Time - last | let t' = t.next |
    some app: Application, c1,c2: Component, perms: set PermName |
      install[t, t', app] or uninstall[t, t', app] or invoke[t, c1, c2] or
      requestPerms[t,t',app,perms] or revokePerms[t,t',app,perms]
}
```

Fig. 2. A snippet of the Alloy model describing dynamic granting and revocation of 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.[7]) 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 As a precondition, the specified application (app) must already exist in the device (line 62). Then, the operation removes the application from the device, as well as all of its associated custom permissions (lines 63-64). The permissions granted to every other application remains unchanged during the operation.

[7] While outside the scope of our analysis, previous studies have pointed this out as a major source of usability and privacy issues in Android [FCH+11].

```
assert NoUnauthorizedAccess {
  all t, t' : Time, callee, caller : Component |
    invoke[t, t', caller, callee] implies authorized[caller,callee,t]
}
// True iff caller is authorized to invoke callee
pred authorized[caller,callee: Component, t: Time] {
  let pname = guardedBy[callee],
    grantedPerm = caller.app.grantedPerms.t & name.pname,
    requiredPerm =
      (callee.app.declaredPerms + Device.builtinPerms) & name.pname |
        some pname implies
          equalOrHigher[grantedPerm.protectionLevel,
                        requiredPerm.protectionLevel]
}
```

Fig. 3. Assertions on the Android permission protocol

Trace definition The fact `traces` defines the behavior of the system as a set of traces that it may produce (lines 68–72). 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.[8] Note that the operation parameters are represented as quantified variables (`app`, `t`, and `t'`) without specifying their actual values; in the ensuing analysis, the Alloy Analyzer attempts to automatically generate values for these parameters that allow a certain sequence of operations to take place. A satisfying instance of the model found by the analyzer, then, corresponds to exactly one of the possible traces of the system.

Other parts 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.[9]

3.4. Extension for dynamic permission assignment

Since the initial conception of our model, various parts of the Android OS have evolved over a series of updates, including its permission protocol. In the latest version of Android (6.0), an application may obtain permissions from the user not only during its installation, but during its runtime as well. This increased flexibility allows the application to carry out an execution even if it lacks access to some of the permissions that it wishes to use (albeit with a reduced level of functionality). Additionally, the user may choose to revoke a previously-granted permission from the application at any time she wishes. In this section, we describe an extension to our initial model in Fig. 1 to capture this dynamic aspect of the new permission protocol.

The Alloy snippet in Fig. 2 describes the two major features of the permission protocol in Android 6.0: requesting and revoking a permission. In the new protocol, permissions that potentially involve a user's private data—i.e., those with the *dangerous* protection level—are no longer granted at install time. To reflect this change, predicate `grantPermissions` from Fig. 1 is modified to ensure that only *normal* and *signature* permissions are granted to an application at this time (Fig. 2, lines 10–13). Recall that signature-level permissions are still granted to the application only if it shares a secret signature with the one that declares this permission.

During its execution, an application may request access to one or more dangerous permissions to be granted by the user; in our model, this operation is represented by `requestPerms`. The first three predicate constraints describe the precondition for the operation; namely, that each requested permission must have been declared in the application manifest file, and must not already be accessible to the application (lines 22–27). As a result of the operation, all of the requested permissions (`perms`) are added to the existing set of permissions granted to the application (line 29). The remaining constraints define a frame condition to ensure that the permissions for the other applications remain unchanged during this operation (lines 30–34).

[8] 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.
[9] http://sdg.csail.mit.edu/projects/android.

In the new protocol, the user is given the ability to revoke previously granted permissions from an application at any desired time. The definition of `revokePerms`, which models this operation, is straightforward; it requires that the input permissions have already been granted to the application (line 39) as a precondition, and simply removes them from the permission set of the application (line 41). Again, the behavior of the human user is implicit in this model; when the predicate evaluates to true over a certain set of permissions, `perms`, this conceptually corresponds to the user having selected those permissions to be removed through the Android UI.

Finally, we modify the trace definition to allow, at every time step, `requestPerms` or `revokePerms` to be performed on an application (`app`) in addition to the three existing operations. Note that as a result of this new trace definition, the overall behavior of the system, as represented by the number of possible execution traces, has also been expanded; an ensuing analysis would now need to explore additional traces where permissions are granted and revoked throughout the lifetime of an application, not just during its (un)installation.

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.

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 [ADKM], 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 Fig. 3. 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`.

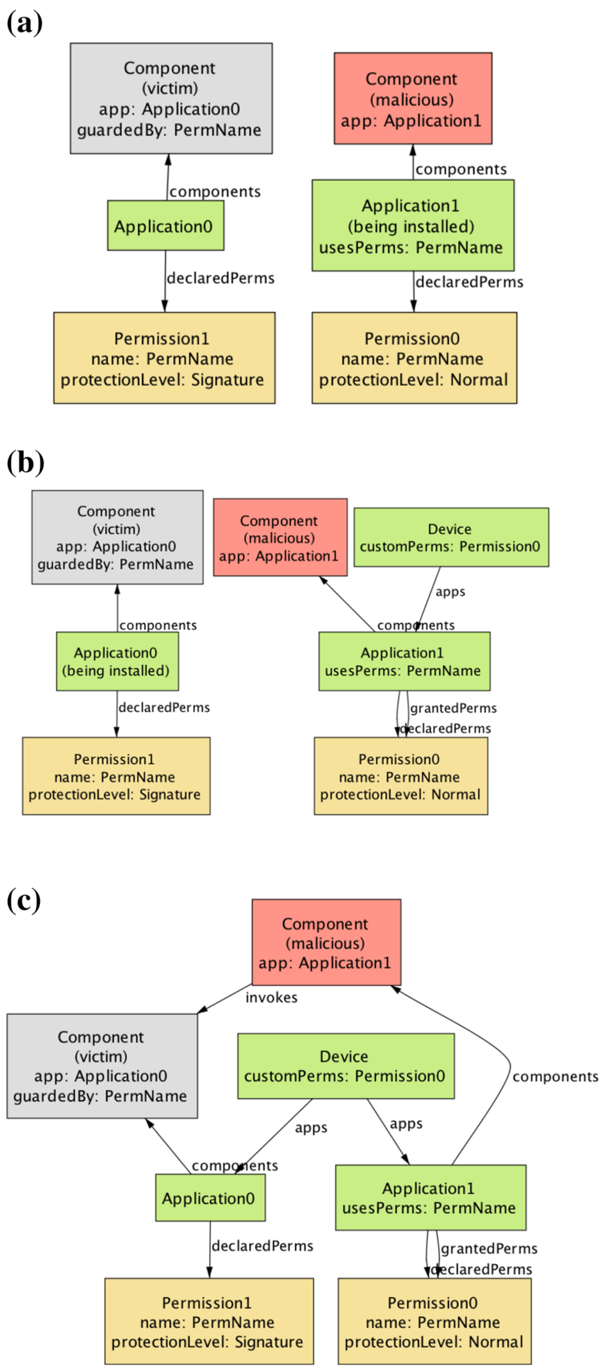

Fig. 4. A counterexample showing an unauthorized access of component `victim` by malicious `Application1` through a custom permission misuse

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.

A visualization of the counterexample is shown in Fig. 4. 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`.

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 Fig. 1:

```
// can't install if a declared perm is named the same as existing one
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 Fig. 4, 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 65 in Fig. 1 as follows:

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

With this modification, the analyzer no longer returns any further counterexample to the given assertion.

4.2. Other vulnerabilities found

Our analysis revealed two other types of vulnerabilities in the permission protocol, which we briefly discuss here.

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 Sect. 4.1.

To our knowledge, this vulnerability with URI permissions is a previously unknown one. However, our further study revealed that the vulnerability exists only up to Android version 2.3.7 (and prior ones); 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 holds the permission. This vulnerability has been identified as the *permission re-delegation* attack in previous work by Felt and her colleagues [FWM+11]; our analysis was able to automatically rediscover it.

4.3. Re-analysis with the dynamic extension

The analysis that led to the detection of the above vulnerabilities was performed on our original model of the Android permission protocol. After extending the model with the details of the dynamic permission assignment in Android 6.0 (as described in Sect. 3.4), we re-analyzed the model against the same security properties.

Our analysis, however, did not reveal any new counterexamples besides the ones already reported. To understand the reason behind the lack of new vulnerabilities, we studied the *unsatisfiable cores* that the Alloy Analyzer produced during the analyses of the original and extended model. An unsatisfiable core is a subset of constraints in a logical specification that together lead to a contradiction; a core is said to be *minimal* if removing at least one of the constraints from the core renders the resulting specification satisfiable. Within the context of assertion checking in Alloy, a minimal core represents the constraints that prevent a counterexample from being generated (within the given scope); in other words, they form an explanation for why the system may satisfy the given property.

The Alloy Analyzer has an ability to produce a minimal unsatisfiable core when it fails to detect any counterexample to an assertion.[10] Once the fix to the custom permission vulnerability is applied to the original model (Sect. 4.1), the analysis of the assertion `NoUnauthorizedAccess` no longer produces any counterexample; instead, it extracts a subset of the constraints in the model as an unsatisfiable core, including the following (line 58 from Fig. 1):

```
pred grantPermissions[t: Time, app: Application] {
  ...
   app.grantedPerms.t in Device.customPerms.t + Device.builtinPerms
}
```

Intuitively, this constraint says that the permissions that are granted to an application must exist as a custom or built-in permission on the device; without this constraint, the analysis would return a counterexample where an application is able to obtain an arbitrary permission that the device does not know about.

When an analysis of the same assertion is performed on the Android 6.0 permission model (Fig. 2), it also produces a core that contains a subset of the constraints from the model, including the following:

```
1  pred grantPermissions[app : Application, t : Time] {
2    let normalPerms = allDevicePerms[t] & protectionLevel.Normal,
3      signaturePerms = allDevicePerms[t] & protectionLevel.Signature {
4        ...
5        app.grantedPerms.t in
6          normalPerms +
7          { s : signaturePerms | let declaringApp = declaredPerms.s & Device.apps.t |
8            declaringApp.signature = app.signature }
9      }
10 }
11 pred requestPerms[t, t': Time, app: Application, perms : set PermName] {
12   ...
13   app.grantedPerms.t' = app.grantedPerms.t + allDevicePerms[t] & name.perms
14   ...
15 }
```

The first constraint (lines 5–8) states that the permissions granted to an application during installation are limited to normal and signature-level ones on the device. The second constraint (line 13) ensures that the permissions granted dynamically by the user must also be the ones that already exist on the device. In other words, an application in Android 6.0 is not able to obtain any more permissions than it was already able to in the previous versions of Android, thus suggesting that the dynamic permission mechanism does not introduce any new vulnerabilities.

[10] This ability, in turn, leverages the minimal core extraction technique implemented as part of Kokod [TCJ08].

```
 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. 5. Snippets of the demonstrative applications that represent the counterexample scenarios shown in Fig. 4

5. Experiments

A rigorous analysis of a formal model, such as the one described in Sect. 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* attacks.

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

- **RQ1:** Can the flaws identified in our formal analysis result in an actual attack with serious security consequences?
- **RQ2:** How susceptible are real-world Android applications to security attacks that are due to these flaws?
- **RQ3:** What is the performance and scalability of our formal analysis approach?

In particular, we focus on the custom permission vulnerability in Sect. 4.1, as it has not been previously studied in the literature.[11] To address RQ1, we developed demonstrative applications that represent postulated malicious behaviors in the generated counterexample in Fig. 4, 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. For RQ3, we ran the analysis over increasing bounds on the size of the top-level type signatures, such as applications and components, and measured the time it took for execution.

5.1. Demonstration of the attack

To test the feasibility of the Alloy counterexample in Fig. 4, we developed a skeletal address book application that corresponds to the victim application in the trace (cf. `Application0` in Fig. 4). Figure 4a partially shows an Android manifest file[12] 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.

[11] The URI permission vulnerability is omitted since it exists only on an outdated version of Android, and the improper delegation flaw has already been studied in [FWM+11].

[12] 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 4b shows part of the manifest file implementation for `MalApp` (corresponding to `Application1` in Fig. 4). 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, representing the malicious component in the counterexample, then simply sends a query to the `AddrBookProvider` component.

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 Alloy counterexample, on two versions of the Android SDK—2.3.7 and 4.4.4—under the Genymotion[13] 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 1500 applications collected from two repositories: (1) popular free applications from Google's Play Store[14] and (2) open-source applications from the F-Droid repository.[15]

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 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 Fig. 6a, 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 makes stealthy permission re-delegation attacks more likely [FWM+11]. More than 85% of custom permissions are defined at signature or dangerous protection levels that regulate access to critical APIs, as shown in Fig. 6b.

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.

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

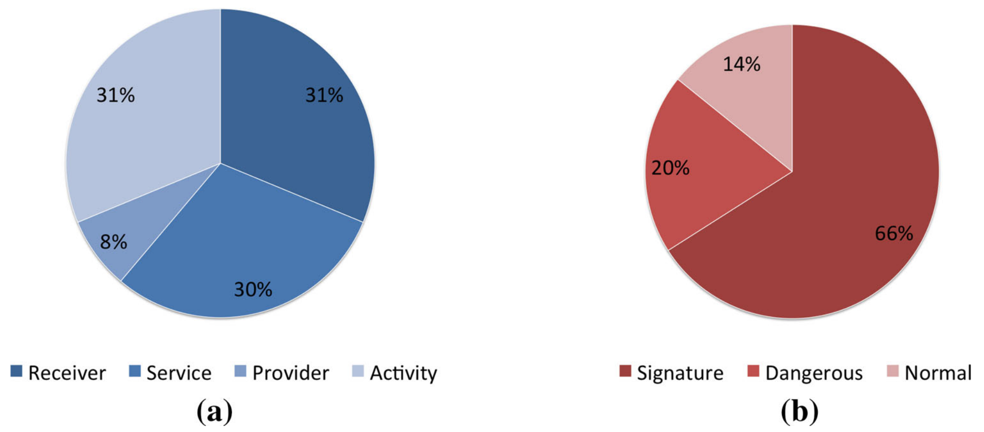

Fig. 6. a Frequency of component types protected by custom permissions; **b** Categorization of custom permissions based on their protection levels

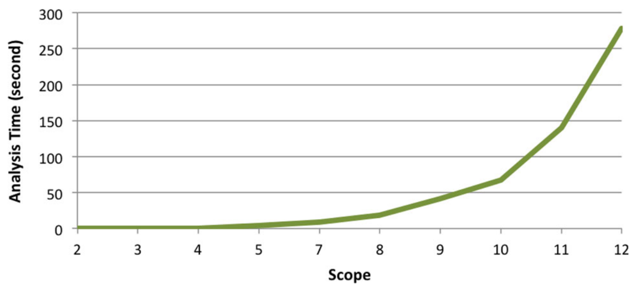

Fig. 7. Average analysis times over an increasing bound (scope) for top-level signatures

5.3. Bounded verification performance

We evaluated the scalability of our permission protocol analysis. Specifically, we measured how variation in size of applications and components, among others, affects the computational time required for bounded verification of the model. As discussed in Sect. 3.1, the completely automated analysis relies on constraint solving over finite relational domains. It thus must be given an explicit scope to bound the number of top-level data type signatures, such as applications, components, permissions and time steps.

Figure 7 shows the analysis time over an increasing bound for top-level signatures, with SAT4J as the SAT solver. All analyses were performed on a Mac OS X machine with 1.8 GHz Intel Dual Cores and 8GB of RAM. As expected, the results show that the analysis time increases as we increased the bounds for top-level signatures. In the worst case, this increase could be exponential, because the number of possible scenarios that must be explored also grows exponentially. The counterexample scenario found corresponds to the scope of 3; we performed additional analysis up to the scope of 12 without discovering any more counterexamples. In all cases, the analysis took under 5 minutes, confirming that an automated verification of the permission protocol implemented in one of the most widely used operating systems is feasible.

These results do not necessarily imply that no other counterexample scenarios violating the checked properties exist beyond the maximum scope in which we performed the analysis. We did not continue our analysis beyond the scope of 12, because we concluded that (1) 5 minutes is a reasonable cutoff for a constraint solver to generate a satisfying instance, and (2) it would be unlikely to find a counterexample that involves a larger number of objects in the permission system. However, we believe that the bounds we used were large enough to capture many design flaws in the Android permission protocol that could cause serious security vulnerabilities.

6. Related work

Over the past few years since the inception of the Android framework, its security has received a lot of attention, due mainly to the popularity of Android as a platform of choice for mobile devices, as well as increasing reports of its vulnerabilities [DDSW10, CFGW11, EOMC11a, EGC+11, GZWJ12a]. We provide an overview of the most notable related research in this area and examine them in the light of our research.

Formal Approaches There exists a related line of research that focuses on formalization of the Android permission protocol. Fragkaki et al. proposed a logical formalization of a permission model similar to the one used in Android [FBJS12]. However, they only performed an informal analysis of the model, and did not identify the custom permission vulnerability. Armando et al. described a formal model of the Android framework and a verification technique based on the *history expressions* process algebra [ACM12]. However, the security analysis of the framework model is left for future work. Chaudhuri also proposed a formal language to describe Android applications and a type system to reason about inter-component data flows [Cha09]. Extending this research work, Fuchs et al. built on the calculus proposed by Chaudhuri to implement SCanDroid, a provably sound static analyzer of data flows to detect permission inconsistencies between applications that could possibly allow malicious access to sensitive information [FCF09]. However, it has never been evaluated over real-world applications.

Along the same line, Bugliesi et al. [BCS13] developed π-Perms, a formal calculus that models a subset of the Android communication architecture and the associated permission mechanism. They then formally defined the notion of safety against the privilege escalation attack. However, similar to the other work we studied, they failed to identify the custom permission vulnerability. More recently, Smith and Coglio [SC15] used the ACL2 theorem prover to develop a formal model of a subset of the Android framework. The authors then leveraged the formal abstractions in verifying the correctness of a sample Android application. Verification using the ACL2 theorem prover requires significant user efforts, whereas in this paper we present a fully automatic analysis of the Android permission protocol using the Alloy Analyzer.

These prior efforts did not identify the flaws that we were able to generate using our approach, either because (1) they did not perform a sufficient amount of analysis, or (2) their analysis targeted different types of properties (e.g., ones about the behavior of a specific application, rather than general properties about the Android permission protocol).

The work that is closest to ours is the one by Shin and his colleagues, where they encoded a formal model of the protocol in Coq and proved a set of security properties using its interactive theorem proving facility [SKFT10]. 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, in a Coq-based approach, the existence of a potential flaw is only suggested by a failed attempt at producing a proof, requiring further human effort to identify the flaw. In comparison, the Alloy Analyzer is capable of automatically producing counterexamples, which we found tremendously helpful for identifying the vulnerabilities.

Android Permissions A large body of the previous work in Android security involves performing manual inspection or program analysis to identify a particular vulnerability in Android applications [SBGM17]. The most closely related research efforts to ours are those that deal specifically with permission vulnerabilities in Android [FCH+11, HBM17, PFNW12, SGS+16, RCE13, LLW+12, BSGM15, EOMC11b, RCT+14, LBKT14, CJD+13, WGZ+13, GZWJ12b]. Among others, 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) [FCH+11]. In a separate work, Felt et al. described a type of attack called *permission re-delegation*, and showed that many existing Android applications are vulnerable to this type of attack [FWM+11]. Kirin provides a set of practical security rules based solely on the permissions requested by Android applications to prevent unsafe combination of permissions that may lead to insecure data flows [EOM09]. Woodpecker targets identifying vulnerabilities in the standard configurations of Android smartphones, i.e., pre-loaded applications, that may lead to permission leaks [GZWJ12b]. These research efforts, similar to many others we studied, mainly focus on application-level analyses, and do not consider analysis of the Android framework nor incorporate a formal verification technique.

Other research efforts concentrate on the analysis of advertisement (ad) libraries that are linked and shipped with Android applications. Since the required permissions of ad libraries are merged into a hosting application's permissions, it is challenging for users to distinguish the permissions requested by the embedded ad libraries from those actually used by the application. Two previous projects deal specifically with such permission tangling issues. AdRisk approaches the problem by decoupling the embedded ad libraries from the host applications and examining the potential unsafe behavior of each library that could potentially result in privacy issues [GZJS12]. AdDroid similarly separates privileged advertising functionality from host applications [PFNW12]. However, these studies do not consider custom permissions.

A number of static analysis tools, such as ComDroid [CFGW11], Epicc [OMJ+13], CHEX [LLW+12], and FlowDroid [ARB+14], have been developed to detect a flow of malicious data within an application or between multiple applications. Among others, Lu et al. [LLW+12] studied security challenges of component hijacking in Android, and developed a tool, called CHEX, to detect such vulnerabilities through static analysis. COVERT [BSGM15] provides a compositional approach for analysis of Android inter-application vulnerabilities. COVERT uses static analysis techniques to extract a formal model of Android applications. 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 applications as they are installed, updated, and removed. More recently, we developed Separ [BSJM16] that provides a proactive scheme for formal synthesis of Android inter-component security policies. It relies on a constraint solver to synthesize potential exploits, from which fine-grained security policies are derived. Such fine-grained, yet system-specific, policies can then be enforced at run time, allowing end-users to safeguard the apps installed on their device from inter-component vulnerabilities.

These static analysis tools are mainly focused on analyzing a *particular* application (or a set of applications, 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. To the best of our knowledge, this study is the first investigation of the Android framework that describes an entirely automated analysis of the Android permission protocol, supported with a formally precise yet bounded relational logic analyzer.

The custom permission vulnerability in Sect. 4.1 was first described in a blog post by an independent security researcher [Mar14]. However, despite its potential security consequences, the vulnerability has not received widespread attention among Android developers; as revealed by our study in Sect. 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.

7. Conclusion

In this paper, we presented a method where a formal model of the Android permission protocol is developed that allows automatic, formal analysis of the protocol to identify potential flaws in permission mechanisms. We then conducted an experimental study to confirm that in reality the potential flaws can lead to serious security vulnerabilities. We also performed a rigorous assessment to estimate how prevalent one of these vulnerabilities in existing Android applications. Finally, we assessed the scalability of the formal analysis approach, the results of which corroborate that the approach is viable for real 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 Sect. 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.

Acknowledgements

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, FA95501610030 from the Air Force Office of Scientific Research, and CCF-1252644 and CCF-1618132 from the US National Science Foundation.

References

[ACM12] Armando A, Costa G, Merlo A (2012) Formal modeling and reasoning about the android security framework. In: Palamidessi C, Ryan MD (eds) Trustworthy global computing, number 8191 in Lecture Notes in Computer Science. Springer, Berlin, pp 64–81. https://doi.org/10.1007/978-3-642-41157-1_5

[ADKM] Andoni A, Daniliuc D, Khurshid S, Marinov D. Evaluating the small scope hypothesis. http://sdg.csail.mit.edu/pubs/2002/SSH.pdf

[ARB+14] Arzt S, Rasthofer S, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P (2014) 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)

[BCS13] Bugliesi M, Calzavara S, Spanò A (2013) Lintent: towards security type-checking of android applications. In: Beyer D, Boreale M (ed) Formal techniques for distributed systems, number 7892 in Lecture Notes in Computer Science. Springer, Berlin, pp 289–304. https://doi.org/10.1007/978-3-642-38592-6_20

[BGS+16] Bagheri H, Garcia J, Sadeghi A, Malek S, Medvidovic N (2016) Software architectural principles in contemporary mobile software: from conception to practice. J Syst Softw 119:31–44

[BKMJ15] Bagheri H, Kang E, Malek S, Jackson D (2015) Detection of design flaws in the android permission protocol through bounded verification. In: Proceedings of the 2015 international symposium on formal methods (FM), volume 9109 of Lecture Notes in Computer Science. Springer, Berlin, pp 73–89

[BSGM15] Bagheri H, Sadeghi A, Garcia J, Malek S (2015) Covert: compositional analysis of android inter-app permission leakage. IEEE Trans Softw Eng 41(9):866–886

[BSJM16] Bagheri H, Sadeghi A, Jabbarvand R, Malek S (2016) Practical, formal synthesis and automatic enforcement of security policies for android. In: Proceedings of the 46th IEEE/IFIP international conference on dependable systems and networks (DSN), pp 514–525

[CFGW11] Chin E, Felt AP, Greenwood K, Wagner D (2011) 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, pp 239–252

[Cha09] Chaudhuri A (2009) Language-based security on android. In: Proceedings of programming languages and analysis for security (PLAS'09), pp 1–7

[CJD+13] Chen KZ, Johnson NM, D'Silva V, Dai S, MacNamara K, Magrino TR, Wu EX, Rinard M, Song DX (2013) Contextual policy enforcement in android applications with permission event graphs. In: NDSS, San Diego, CA

[DDSW10] Davi L, Dmitrienko A, Sadeghi A-R, Winandy M (2010) Privilege escalation attacks on android. In: Proceedings of the 13th international conference on Information security (ISC).

[EGC+11] Enck W, Gilbert P, gon Chun B, Cox LP, Jung J, McDaniel P, Sheth AN (2011) Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of USENIX OSDI

[EOM09] Enck W, Ongtang W, McDaniel P (2009) On lightweight mobile phone application certification. In: Proceedings of the 16th ACM conference on Computer and communications security, Chicago, IL. ACM, pp 235–245

[EOMC11a] Enck W, Octeau D, McDaniel P, Chaudhuri S (2011) A study of android application security. In: Proceedings of USENIX.

[EOMC11b] Enck W, Octeau D, McDaniel P, Chaudhuri S (2011) A study of android application security. In: Proceedings of the 20th USENIX conference on security, SEC'11, San Francisco, CA. USENIX Association, pp 21–21

[FBJS12] Fragkaki E, Bauer L, Jia L, Swasey D (2012) Modeling and enhancing android's permission system. In: 17th European symposium on research in computer security (ESORICS), pp 1–18

[FCF09] Fuchs AP, Chaudhuri A, Foster JS (2009) Scandroid: Automated security certification of android applications

[FCH+11] Felt AP, Chin E, Hanna S, Song D, Wagner D (2011) Android permissions demystified. In: 18th ACM conference on computer and communications security (CCS), pp 627–638

[FWM+11] Felt AP, Wang HJ, Moshchuk A, Hanna S, Chin E (2011) Permission re-delegation: attacks and defenses. In: 20th USENIX security symposium

[Goo] Google. Android system permissions. http://developer.android.com/guide/topics/security/permissions.html

[GZJS12] Grace MC, Zhou W, Jiang X, Sadeghi AR (2012) Unsafe exposure analysis of mobile in-app advertisements. In: Proceedings of the fifth ACM conference on security and privacy in wireless and mobile networks, WISEC '12, Tucson, AZ. ACM, pp 101–112

[GZWJ12a] Grace M, Zhou Y, Wang Z, Jiang X (2012) Systematic detection of capability leaks in stock android smartphones. In: Proceedings of the 19th annual symposium on network and distributed system security

[GZWJ12b] Grace MC, Zhou Y, Wang Z, Jiang X (2012) Systematic detection of capability leaks in stock android smartphones. In: NDSS, San Diego, CA

[HBM17] Hammad M, Bagheri H, Malek S (2017) Determination and enforcement of least-privilege architecture in android. In: 2017 IEEE international conference on software architecture (ICSA), pp 59–68

[Jac12] Jackson D (2012) Software abstractions: logic, language, and analysis, 2nd edn. MIT Press, Cambridge

[LBKT14] Li L, Bartel A, Klein J, Traon YL (2014) Automatically exploiting potential component leaks in android applications. In: Proceedings of the 13th international conference on trust, security and privacy in computing and communications, Beijing, China, pp 388–397

[LLW+12] Lu L, Li Z, Wu Z, Lee W, Jiang G (2012) Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the ACM conference on computer and communications security (CCS)

[Mar14] Murphy M (2014) Vulnerabilities with custom permissions. http://commonsware.com/blog/2014/02/12/vulnerabilities-custom-permissions.html

[OMJ+13] Octeau D, McDaniel P, Jha S, Bartel A, Bodden E, Klein J, Traon YL (2013) 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

[PFNW12] Pearce P, Felt AP, Nunez G, Wagner D (2012) AdDroid: privilege separation for applications and advertisers in android. In: Proceedings of the 7th ACM symposium on information, computer and communications security, ASIACCS '12, Seoul, Republic of Korea. ACM, pp 71–72

[PXY+13] Pandita R, Xiao X, Yang W, Enck W, Xie T (2013) Whyper: towards automating risk assessment of mobile applications. In: Proceedings of the 22nd USENIX conference on security, SEC'13, Berkeley, CA, USA. USENIX Association, pp 527–542

[RCE13] Rastogi V, Chen Y, Enck W (2013) AppsPlayground: automatic security analysis of smartphone applications. In: Proceedings of the 3rd ACM conference on data and application security and privacy, CODASPY '13, San Antonio, TX. ACM, pp 209–220

[RCT+14] Ravitch T, Creswick ER, Tomb A, Foltzer A, Elliott T, Casburn L (2014) Multi-app security analysis with FUSE: statically detecting android app collusion. In: Proceedings of the 4th program protection and reverse engineering workshop, PPREW-4, New Orleans, LA. ACM, pp 4:1–4:10

[SBGM17] Sadeghi A, Bagheri H, Garcia J, Malek S (2017) A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. IEEE Trans Softw Eng 43(6):492–530

[SC15] Smith E, Coglio A (2015) Android platform modeling and android app verification in the acl2 theorem prover. In: Proceedings of the 7th international conference on verified software: theories, tools, and experiments, VSTTE'15, pp 183–201

[SGS+16] Schmerl B, Gennari J, Sadeghi A, Bagheri H, Malek S, Camara J, Garlan D (2016) Architecture modeling and analysis of security in android systems. In: Software architecture. Springer, Cham, pp 274–290

[SKFT10] Shin W, Kiyomoto S, Fukushima K, Tanaka T (2010) A formal model to analyze the permission authorization and enforcement in the android framework. In: IEEE International conference on privacy, security, risk and trust, pp 944–951

[SZZ+11] Schlegel R, Zhang K, Zhou X, Intwala M, Kapadia A, Wang X (2011) Soundcomber: a stealthy and context-aware sound trojan for smartphones. In: Proceedings of 18th annual network and distributed system security symposium (NDSS)

[TCJ08] Torlak E, Chang FS-H, Jackson D (2008) Finding minimal unsatisfiable cores of declarative specifications. In: FM 2008: formal methods, 15th international symposium on formal methods, Turku, Finland, May 26–30, 2008, proceedings, pp 326–341

[TJ07] Torlak E, Jackson D (2007) Kodkod: a relational model finder. In: Tools and algorithms for the construction and analysis of systems, 13th international conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24–April 1, 2007, Proceedings, pp 632–647

[WD96] Woodcock J, Davies J (1996) Using Z. Specification, refinement, and proof. Prentice Hall, Upper Saddle River

[WGZ+13] Wu L, Grace M, Zhou Y, Wu C, Jiang X (2013) The impact of vendor customizations on android security. In: Proceedings of the 2013 ACM SIGSAC conference on computer and communications security, CCS '13, Berlin, Germany. ACM, pp 623–634

[WLBF09] Woodcock J, Larsen PG, Bicarregui J, Fitzgerald J (2009) Formal methods: practice and experience. ACM Comput Surv 41(4):19:1–19:36

Received 31 March 2016

Accepted in revised form 19 October 2017 by Frank de Boer, Nikolaj Bjorner, and Andrew Butterfield

Published online 7 November 2017