Resolving the Predicament of Android Custom Permissions

Güliz Seray Tuncay, Soteris Demetriou, Karan Ganju, Carl A. Gunter
University of Illinois at Urbana-Champaign
{tuncay2, sdemetr2, kganju2, cgunter}@illinois.edu

***Abstract*—Android leverages a set of *system permissions* to protect platform resources. At the same time, it allows untrusted third-party applications to declare their own *custom permissions* to regulate access to app components. However, Android treats custom permissions the same way as system permissions even though they are declared by entities of different trust levels. In this work, we describe two new classes of vulnerabilities that arise from the 'predicament' created by mixing system and custom permissions in Android. These have been acknowledged as serious security flaws by Google and we demonstrate how they can be exploited in practice to gain unauthorized access to platform resources and to compromise popular Android apps. To address the shortcomings of the system, we propose a new modular design called Cusper for the Android permission model. Cusper separates the management of system and custom permissions and introduces a backward-compatible naming convention for custom permissions to prevent custom permission spoofing. We validate the correctness of Cusper by 1) introducing the first formal model of Android runtime permissions, 2) extending it to describe Cusper, and 3) formally showing that key security properties that can be violated in the current permission model are *always* satisfied in Cusper. To demonstrate Cusper's practicality, we implemented it in the Android platform and showed that it is both effective and efficient.**

I. Introduction

Android's permission model forms the security basis for the critical operations that can be performed on the platform by the apps. In a nutshell, the main purpose of this model is to regulate access to platform and app resources, which is achieved by utilizing a set of security labels, called permissions. In order to protect the platform resources (e.g., microphone, Internet etc.), the platform uses *system permissions*, which are a predefined set of permissions introduced by the platform itself. The permission model also provides the platform with finer-grained security as a means to protect Inter-Process Communication (IPC) between different app or system components. Specifically for this purpose, Android introduces *custom permissions*: these are application-defined permissions which allow developers to regulate access to their app components by other apps. In fact, the use of custom permissions is very common among third-party applications. According to our study on the top free apps on the Google Play Store, 65% of the apps define custom permissions, while 70% request them for their operation.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23210
www.ndss-symposium.org

Unfortunately, design flaws and vulnerabilities in custom permissions can completely compromise the security of IPC, inevitably leading to exploits on third-party apps and the platform itself. Previous work has consistently found custom permissions to be problematic [30], [28], and as a response to these studies, Google made an effort to address the identified problems by releasing bug fixes. However, similar vulnerabilities still exist even after Google patched this initial wave of vulnerabilities. In this work, we present two classes of attacks that exploit the vulnerabilities in custom permissions to get unauthorized access to platform and app resources. With one of our attacks, a malicious app can bypass the user interaction requirements for acquiring dangerous system permissions on Android versions that support runtime permissions and stealthily access high-risk platform resources (e.g., camera, microphone etc.). With our other attack, a malicious app can escalate its privileges to gain elevated access to the protected components of other apps. We further demonstrate how an adversary can utilize the aforementioned vulnerabilities to target high profile apps with millions of downloads, such as *CareZone* and *Skype*, and access sensitive user data (e.g., medical conditions, insurance information) and functionalities (e.g., VoIP calls). We have officially reported these attacks to Google, which acknowledged them as severe flaws that need to be addressed in the next versions of Android.

In our investigation of the Android permission model and its respective source code, we observed that *there is no separation of trust between system and custom permissions in the Android framework*, which leads to the manifestation of permission vulnerabilities; we call this failure to distinguish custom permissions in the system, the 'predicament' of custom permissions. First, system and custom permissions are currently insufficiently isolated and they receive the same kind of treatment from Android, which opens up opportunities for malicious apps to utilize custom permissions to obtain unauthorized access to platform resources. Second, there is currently no enforced naming convention for when declaring custom permissions—apps are allowed to declare custom permissions with *any* name they desire. This creates a confused deputy problem where a privileged app's protected resources can be utilized by unauthorized apps that possess different custom permissions declared with the same name as of the ones used by the privileged app to protect its resources. In order to systematically address these problems, we propose a design and corresponding implementation which we call *Cusper*. Cusper decouples the handling of custom permissions from

system permissions to prevent an adversary from escalating their privileges and stealthily acquiring system resources. Additionally, Cusper implements an OS-level naming convention for custom permissions to prevent custom permission spoofing. This is backward-compatible with existing apps and enables the system to properly identify custom permissions according to the developer signature of their definer apps.

To prove the correctness of Cusper, one could employ traditional analysis methods such as testing and static analysis. However, these are typically insufficient since they depend on the analyst determining all possible test cases, a challenging endeavor. In contrast, formal methods can be leveraged to build models and verify that key properties are never violated in a proposed system. In fact, previous work has already used the latter approach to formally model the Android permission model [20], [21], [29]. Unfortunately, the proposed formal models are outdated since they correspond to the *install time* permission model of Android. In order to verify the correctness of Cusper, we build the first formal model of the Android *runtime* permission model using the Alloy specification language. Specifically, we model the data abstractions for permission-related structures and the behavior of system operations (e.g., install, uninstall, update) that concern permissions for the runtime model, which significantly differ in terms of both abstractions and behavior from the previous model. We found that the original permission model violates two fundamental security properties regarding access to app and platform resources: 1) there should be no unauthorized component access, and 2) there should be no access to high-risk ('dangerous') platform resources without user's consent. We leverage our formal model to demonstrate the existence of vulnerabilities that violate these invariants in the original permissions model and show that, in contrast, Cusper *always* satisfies them. Finally, to illustrate Cusper's practicality, we implement it in Android and show that it effectively resolves the identified vulnerabilities while incurring a negligible overhead. Our contributions can be summarized as follows:

1) We identify severe—as acknowledged by Google—vulnerabilities in custom permissions and demonstrate how they can be exploited in practice to obtain unauthorized access to critical platform and app resources.
2) We study the identified vulnerabilities in detail to understand the design flaws leading to their occurrence, and based on our observations, we propose a modular design, Cusper, which aims to systematically eradicate and prevent custom permission vulnerabilities.
3) We implement the first formal model of the Android runtime permission model, and formally prove the correctness of Cusper.
4) We implement Cusper on Android and show that it resolves the identified vulnerabilities effectively and efficiently.

The rest of the paper is organized as follows: Section II covers the background information. Section III presents our analysis on the use of custom permissions by the most popular apps on Google Play. Section IV describes our attacks that exploit the custom permission vulnerabilities. Section V presents our design and implementation of Cusper. Section VI presents our Alloy formal model of Android permissions. Section VIII discusses related work on Android permissions. Finally, we conclude the paper in Section IX.

II. Background

In this section, we will cover the background on Android permissions, IPC on Android, and formal verification using Alloy.

A. Android Permissions

In Android, each app runs as a separate Linux user within its assigned sandbox with limited access to resources to ensure the integrity of the system and the apps. When an app wishes to use a resource outside its sandbox, it has to conform to Android's permission model and explicitly request it.

Permission Essentials. Android associates permissions with protection levels depending on their severity. There are currently three protection levels in Android: *normal*, *signature*, and *dangerous*. Normal permissions are used to protect resources (e.g., Internet) that constitute very little risk to user's privacy or the operation of other apps; whereas dangerous permissions are associated with very high risk operations (e.g., accessing a user's contact list). Signature permissions protect private resources of apps, where the requester can be granted the permission only if it is signed with a matching certificate to that of the definer of the permission of interest. Additionally, Android has permission groups that cluster permissions based on their utility [4].

Before Android 6.0 (API level 23), all permissions were granted at application installation time. However, starting with version 6.0, Android adopted the runtime permission model, where dangerous permissions are granted to an app at runtime by the user the first time they are used by this app, and the user is given the ability to revoke these permissions to apps at any time. Normal and signature permissions are still granted at installation and cannot be revoked by the user. According to the runtime model, if a dangerous permission in a permission group is granted to an app, all the dangerous permissions in that group will also be granted (if explicitly requested by the app) in order to minimize user's effort.

Custom Permissions. Third-party apps are allowed to define new permissions on Android. These permissions, called *custom permissions*, are used to protect an app's own resources from others. In order to define a new custom permission, an app must provide a permission name and can optionally include a permission group to which this permission belongs and a description regarding the utility of the permission in its manifest as shown in the lines 1 - 6 of Listing 1. Additionally, in order to request a permission, an Android app needs to declare the use of the permission by referring to it with its name, as shown in line 8 of Listing 1. Furthermore, Android allows applications to create custom permissions dynamically via the use of the `addPermission()` API method. In order for this method to work successfully, apps need to declare permission trees in their manifest file which state the domain name under which the dynamic permissions will be created.

Although it is suggested that reverse domain name notation should be used for custom permission names, there is currently

Listing 1: Creating and requesting custom permissions

```
<permission <!--Create a custom permission-->
  android:name="com.example.PERM_NAME"
  android:protectionLevel="normal"
  android:permissionGroup=
    "android.permission-group.STORAGE">
</permission>
<!-- Request a custom permission -->
<uses-permission android:name="com.example.PERM_NAME" />
```

no naming convention enforced by the system for custom permissions and apps can use any name they desire when creating new custom permissions. One exception is that Android does not allow two different permissions to coexist on the same device if they have the same name; hence, installation of an app which defines a permission with a name that belongs to an existing permission on the device will be denied by the system. Conventional use case for custom permissions is for apps to define custom permissions with the signature protection level so that only the apps that are signed with the same certificate (e.g., apps that belong to the same developer) can utilize the definer app's resources.

B. Inter-Component Communication

The Android operating system relies on IPC (also called ICC on Android) in order to achieve re-usability. Here, we present the details on how IPC is achieved and protected on Android.

App Components. Android apps can comprise of four components: activities, services, broadcast receivers, and content providers. Each of these components (except content providers) can be an entry point to the apps. Activities are components that present the user with a graphical user interface to perform a single task. Services run in the background to perform long-running operations with no user interface. Broadcast receivers are used to receive broadcast messages from the system or the other apps. Content providers allow storing and sharing data between apps via a relational database interface. Communication between components is achieved through the Intent mechanism on Android. Intents are asynchronous messages between components in the same app or different apps that are used for activation of components.

IPC Security. Android ensures the security of IPC by utilizing its permission model. When two components communicate, both the caller and the callee can require the other party to hold certain permissions for a successful communication. More specifically, this protection can be achieved through the use of custom permissions, which are used by app developers to restrict access to their components. For example, if a component is protected with a signature permission, only the apps that are signed with the same certificate as that of the component owner can access it. Both third-party app developers and the system rely on the correct operation of custom permissions for their security; hence, it is of paramount importance to ensure the security of custom permissions.

C. Formal Verification via Alloy

Alloy is a declarative specification language that is used to model the behavior and structural constraints of complex systems [1]. It provides a modeling tool called Alloy Analyzer that operates based on first-order (i.e., predicate) logic and can be used to analyze formal models created with the Alloy language.

Alloy Language. Statements in Alloy can be interpreted both from object-oriented (OO) programming paradigm and from set theory perspectives. *Signature* is a declaration of a schema, which defines the vocabulary of the model. It is similar to the concept of a class in OO paradigm and to a set in set theory. It can consist of several *fields*, which are equivalent to fields in OO paradigm and to relations from a set theoretical perspective. *Facts* are global constraints to the model that are always supposed to hold. *Predicates* define parametrized constraints, which can be interpreted as operations that can be performed in the model. *Functions* are expressions with declaration parameters and they return a result based on the parameters. *Assertions* are assumptions made on the model and they can be validated via the Alloy analyzer. Additionally, Alloy allows using multiplicity keywords as quantifiers in quantified constraints: `all` (universal quantifier), `some` (existential quantifier), `lone` (zero or one), `one` (exactly one), `no` (zero). Also, it is possible to use `some` (or `set` interchangeably), `lone`, and `one` for field declarations in signatures to indicate the number of elements a field can take and also for signature declarations to indicate the number of elements that can belong to the set of the signature.

Alloy Analyzer. The Alloy Analyzer tool performs only finite scope checks on the models. The analysis is sound since it can never return false positives and is complete up to a scope as the tool will never miss any counterexamples that are equal or smaller than the specified scope. As in traditional model checking, Alloy models are infinite, that is, the specification dictates how the components of a system should behave without any restrictions on their quantity. The analyzer also provides automated analysis by allowing automatic generation of examples that satisfy a given model as well as counterexamples to claims (i.e., assertions) that are expected to hold in the model.

III. Use of Custom Permissions

Custom permissions provide security to IPC that apps harness for their operation. They are utilized by app developers to restrict access to components as per the sensitivity of the protected resource. In this section, we investigate the prevalence of custom permissions among the top free apps on Google Play and showcase two high-profile apps that we selected to launch attacks on by exploiting the vulnerabilities in custom permissions.

A. Prevalence

We collected 50 top free apps from each of the Google Play Store categories (with some failures in collection) and in total analyzed 1308 apps to identify statistics regarding the use of custom permissions. As can be seen in Table I, 65% of the apps in our dataset declare custom permissions (statically or dynamically) and 70% of them request custom permissions. Additionally, 89% of all the permissions created by these apps are of protection level signature (see Table II), which indicates

TABLE I: Apps at risk due to custom permissions

Usage	Number of Apps
Create Static Custom Permissions	834 (64%)
Create Dynamic Custom Permissions	50 (3%)
Create Custom Permissions	847 (65%)
Request Third-party Permissions	919 (70%)
Total number of apps in dataset	1308 (100%)

TABLE II: Protection Levels of Custom Permissions

Permission Protection Level	Number of Permissions
Signature Permissions	1203 (89%)
Dangerous Permissions	14 (1%)
Normal Permissions	40 (2%)
Signature or System Permissions	57 (4%)
Total Number of Permissions	1350 (100%)

that app developers typically use custom permissions to allow other apps to utilize their protected components only if they are signed by the same developer or company.

This analysis shows that custom permissions are commonly utilized by app developers. Vulnerabilities in their use create risks for the security of the platform and key apps. To illustrate this we have identified ways we can launch attacks on the platform to obtain any system resource (e.g., camera, microphone) without user consent and on apps to stealthily access their protected components and data. Apps that utilize custom permissions are potentially susceptible to the attacks that target their protected components when they are installed on Android 6.0 or newer. This currently includes more than 50% of all Android devices [3] with a growing user base. This is a widespread security concern: just the apps in our dataset have been downloaded on average 25 million times.

B. Case Studies

In this section, we present case studies where sensitive data or resources of popular Android apps have been leaked through the custom permission vulnerabilities. In order to obtain the vulnerable components that leak resources, we conduct manual analysis which requires going through the decompiled application code and crafting attacks specifically for the app in study. As an example, to attack a vulnerable activity component, we go through the decompiled code to find the Java file for the activity itself and if it is not obfuscated, we proceed to inspect the source code to identify whether the intent it expects is of a particular format. Finally, using this information, we create attack apps that exploit the existing custom permission vulnerabilities and stealthily activate the component of interest with the appropriate intent. We will explain the details of how this attack works in Section IV-B.

CareZone is a medical Android app produced by a company with the same name. It has 1,000,000+ downloads and has a 4+ rating on the Google Play Store. The app allows users to store medical-related information such as health background (e.g., blood type, medical conditions, allergies etc.), medication lists, medical contacts and their addresses, calendar events, insurance information, and photos or health files in an organized manner. It also features calendars to track appointments as well as to-do and notification lists for tracking tasks. All of this medical data and meta-data is stored in a single content provider which has been exported and is protected by a signature permission. There are no other dynamic checks on accessing the content provider. Our attack is able to bypass the signature requirements and read the entire content provider, which gives us access to the aforementioned sensitive data without the user's explicit consent or knowledge. The fact that all this data was stored in a single content provider seems to reflect the developer's implicit reliance on the security guarantees provided by the platform.

Skype is an Android app by Microsoft that allows users to make voice and video calls over the Internet. It has 500,000,000+ downloads and a 4+ rating. Skype has an activity which can be invoked to start the call functionality to any telephone number. This activity is protected using a signature permission which, once bypassed, would allow the adversary to invoke calls to a specified person or number. This could have many implications. For example, it could be used as part of a suite of other spying capabilities. Our attack is able to spoof the original permission and launch Skype calls without the user's knowledge through this protected activity.

IV. Attacks

Custom permissions play an important role in enabling reusability on the Android platform by providing security to IPC; hence, any threat to their proper operation can result in the compromise of the security of the apps and the platform itself. In this section, we discuss two types of custom permission vulnerabilities we identified on Android: 1) custom permission upgrade and 2) confused deputy. By exploiting these vulnerabilities, an app can bypass user consent screens for granting/denying permissions to obtain high-risk system resources and can also gain unauthorized access to protected components of other apps. We reported these to Google which acknowledged both of them as severe vulnerabilities. Given its real-world implications and its prevalence, we believe that the predicament of custom permissions constitutes a current and notable security risk worth addressing.

Threat Model. We consider an adversary that has the ability to crawl app markets (e.g., Google Play Store) to download victim apps of interest, reverse engineer them by utilizing several tools [7], [12], [10], and analyze the Android manifest files and source code of these apps to observe the cases where custom permissions are used to protect app components. The adversary can build and distribute on app markets a set of malicious apps that exploit the custom permission vulnerabilities of Android to launch attacks on the victim apps and on the platform.

A. Custom Permission Upgrade Attack

Android runtime permission model (supported by Android 6.0 and onward) requires user's approval for granting apps permissions of protection level `dangerous`. This attack enables a malicious app to completely bypass the user consent screen and automatically obtain *any* `dangerous` system permissions [15].

In particular, there are 24 dangerous permissions in 9 permission groups [2] on the current version of the Android platform (7.0), which protect access to high-risk system resources (e.g., storage, contacts, location, camera, microphone, sms, sensors etc.). Our attack illustrates how an adversary can

gain unfettered access to *all* high-risk system resources that are protected by these permissions without the user's consent.

Attack Overview. First, the adversary creates an app that includes in its manifest file a custom permission declaration with the protection level `normal` or `signature` and sets this custom permission to be a part of a system permission group (e.g., storage, camera etc.). Then, they update the definition of this custom permission so that the protection level is changed to `dangerous` and proceed to push an update to their app on the respective app market. Here, this update can be pushed to all the app users after the app reaches a target user base. In addition, specific user groups can be targeted via the use of push services (e.g., Google Cloud Messaging (GCM) [11], which is used by 94% of the apps in our database that utilize custom permissions) that allow sending update notifications, and via enterprise app stores (e.g., Appaloosa [18]) that enable enforced targeted updates. The expectation is that since the custom permission is of level `dangerous`, the user will be prompted at runtime to make a decision on whether to grant or deny this permission in the runtime permission model. However, the malicious app automatically gets granted the permission. In addition, since the runtime permission model grants dangerous permissions on a group basis, the app also automatically obtains all the other requested dangerous permissions of the system permission group that the original permission belongs to. Same procedure can be followed to attack *any* system permissions group; hence, the adversary can silently obtain *all* system permissions simultaneously. Requesting dangerous permissions in the Android manifest constitutes no problems for the adversary, as permission requirements of an app are not directly presented to users at installation since Android 6.0. Hence, the user will be completely unaware that all these system permissions are granted to the app.

Internals of the Attack. Android does not treat custom permissions any differently than system permissions. As we will describe in more detail in Section VI-B, granting of any permission is handled according to the permission's protection level and the SDK level of the requesting app on the runtime model. Normal and signature permissions are always granted as install time permissions. For legacy apps (SDK level $<$23), dangerous permissions are still install time permissions; whereas for new apps they are granted at runtime. In case a legacy app gets *upgraded* to SDK level 23 or more, the system also "upgrades" the granted dangerous permissions from install time to runtime permissions and automatically grants them if they were not manually revoked by the user through the permission settings (indicated by the `FLAG_PERMISSION_REVOKE_ON_UPGRADE` flag being set for the permission). However, the system does not consider other cases where a change in the definition of a permission can mistakenly trigger the same *permission upgrade* operation. In the case of our attack, when a custom permission declaration is modified by an app update such that the protection level changes from normal or signature to dangerous, the system wrongfully treats this case as *an app upgrade*, and tries to also upgrade the existing permission to a runtime permission even though the update operation did not change the app's SDK level. Since Android does not allow users to revoke normal or signature permissions, the aforementioned flag will never be set for the existing install permission. Hence, the dangerous permission will be granted automatically without any user consent. Evidently, this violates a key security principle that should always hold in the Android runtime permission model: *no dangerous runtime permission should be granted without user interaction.*

Note that, the problem here is that the system does not consider the special cases that can happen in the case of custom permissions. If a *system* permission is being upgraded from an install to a runtime permission for an app, this can only mean that the app is being upgraded to SDK level 23 or more. However, when a *custom* permission is upgraded for an app, this can indicate either that the legacy app is being upgraded, or that the permission definition is being changed from normal or signature to dangerous. Currently, the system is not equipped with the ability to distinguish between these two cases for custom permissions as it cannot even distinguish system permissions from custom permissions. To make things worse, the system allows a third party developer to declare a custom permission as a part of a system permission group. Thus, the adversary can not only get a dangerous custom permission silently granted, but they can further get access to all system dangerous permissions in the same group with any granted dangerous custom permission.

B. Confused Deputy Attack

In this attack, the adversary exploits the lack of naming conventions for custom permissions on Android to launch an attack on a victim app that utilizes custom permissions to protect its components [13]. To do this, the adversary counterfeits the custom permissions of the victim app by reusing their names in her own permission declarations and takes advantage of the system's inability to track the true origin of permissions to access protected components of the victim app.

Attack Overview. In this attack, the adversary's goal is to get the operating system to grant their apps a signature custom permission of a victim app that is signed by a different key than that of the adversary and therefore obtain unauthorized access to the components protected by this signature permission.

In order to achieve this, the adversary develops two applications: 1) a definer attack app which spoofs the custom permission of the the victim app by reusing the same permission name but changing the protection level to `dangerous`, 2) a user attack app which only requests this permission in its manifest file. The reason adversary needs two apps to carry out this particular attack is that Android currently does not allow two applications that declare a custom permission with the same name to coexist on the same device. Hence, the adversary's app cannot simultaneously exist on the device along with the victim app if it declares a permission with the same name to the one used by the victim. However, the adversary can divide their attack into two different apps, one that spoofs the custom permission as long as the victim app is not installed on the device, and a second one that only requests this permission and is able to coexist with the victim. The definer attack app needs to be installed first by the user, and this should be followed by the installation of the user attack app. After the spoofed permission of the definer attack app is granted to the user attack app at runtime, the definer attack app can be uninstalled

by the user or updated by the app developer (for all users or targeted to a specific group by using services like GCM or Appaloosa) to remove the custom permission definition so that the victim can be installed afterwards. After the installation of the victim app, the user attack app is able to launch an attack on the victim to freely access victim's signature-protected components even though it is not signed with the same app certificate as the victim. Google acknowledged this as a high-severity attack since it bypasses operating system protections that isolate application data from other applications.

Note that there can be many ways for an adversary to get the user to install two applications on their device. For instance, the app developer can use in-app advertisements and links to direct the user to app stores to download their other app (e.g., Facebook and Messenger). Another effective way would be to utilize a common Android app development practice called plug-in architectures [8], [5], which on demand unravel new features to the user in the form of new apps in order to foster re-usability and save storage space by unlocking features only if they are necessary. An example to apps using this architectures is Yoga Guru [19], which unlocks users new yoga exercises—as part of new apps—only after making progress with the set of exercises they currently have.

Internals of the Attack. During the installation or the update of an app, if a permission definition is removed from the system due to this operation, the system iterates over the existing apps to readjust their granted permissions. The desired outcome of this behavior is that all the undefined permissions should be revoked to the remaining apps after an uninstallation or an update. However, instead of immediately revoking an undefined permission to the remaining apps, the system instead revokes it only if a new permission with the same name is being redeclared. Even though this initially seems unharmful since the granted permission is rendered useless until it is redefined, this behavior is what enables our attack. Once the permission name is recycled by the introduction of a new signature permission, due to the mismatch of signatures, the system attempts to revoke this permission to the adversary; however, it mistakenly only revokes install permissions and fails to do so for runtime permissions. This, in turn, leaves the undefined runtime permissions granted to the app. Since Android utilizes only the names of permissions during permission enforcement, it cannot differentiate between two distinct permissions with the same declared name. Hence, the app holding a "dormant" dangerous permission gains unauthorized access to components protected with a signature permission with the same name. Evidently, this violates a key security principle that should always hold in the Android permission model: *there should be no unauthorized component access*.

Note that again the framework developers seem to disregard the peculiarities and corner cases created by custom permissions. Currently, a third-party app cannot define a custom permission using the name of an existing system permission. It was, however, possible for them to use the name of a new system permission that were to be defined in the next version of the OS, to hijack the system permissions [32]. Google's fix to this security vulnerability was that the system would always take the ownership of permissions defined by itself; hence, spoofing attacks on system permissions should not succeed anymore as the platform is treated as the main principal to define/remove system permissions under any circumstance. However, a similar approach cannot be applied to custom permissions as the system cannot make a decision regarding the ownership of a permission between two apps that define the same custom permission. Hence, we not only need to identify whether a permission is system or custom, but in the latter case, we also need a way to identify its origin and treat it as a different permission in case there are other permissions with the same name. It is worth noting that whether a permission is custom or system cannot be determined solely based on its name as even custom permissions can currently use system prefixes (e.g. `android.permission`) and system apps can create permissions with any name without being forced to use a system prefix (e.g., browser permissions).

V. Cusper

System permissions are defined by the platform—a privileged principal—whereas custom permissions are defined by apps—less privileged principals. The former kind typically protects system resources while the latter is utilized to protect inter-component communication between apps. The fact that the system treats them the same, results in severe security vulnerabilities as the ones we discovered (Section IV). Note that other vulnerabilities might also exist or might manifest in the future because of this non-separation between the two classes of permissions. Ideally, we need a new design which will allow us to achieve a clean separation of trust between the system and custom permissions. This way, the system will have to handle the two cases differently avoiding logic errors and at the same time, any potential vulnerabilities in third party app custom permissions will not allow privilege escalation, which can enable exploits of system permissions and platform resources. However, such a new design needs to be carefully constructed to be practical. In fact, it needs to be as simple as possible to be adopted in practice, and backward compatible. A complete redesign of the Android permission model would require non-trivial modifications to the Android framework while thousands of apps relying on custom permission would be immediately affected. Instead, in our work, we introduce two main design principles which can easily be incorporated into the current design of Android permissions, require no changes to the existing apps, and can guarantee a separation of trust eliminating the threat of privilege escalation in permissions, without breaking the operation of system and third-party components that rely on permissions. These design principles are: (a) decoupling of system and custom permissions; (b) new naming scheme for custom permissions. We implement these in our system that we call Cusper.

A. Isolating System from Custom Permissions

Currently, Android does not maintain distinct representations for system and custom permissions, that is, the system does not track whether a permission originated from the system or from a third-party app. Due to this reason, both types of permissions are also granted and enforced in the same fashion. As we have shown in Section IV, this is problematic as it allows apps to use custom permissions to gain unauthorized access to system permissions. For example, a malicious app can declare a custom permission and assign it to a system

permission group. This behavior is allowed by Android since it does not differentiate between the two permission types. Thus, when the custom permission is granted, the app automatically gains access to the system permissions in the same group, essentially elevating its privileges from a permission defined by a low trust principal to permissions defined by the platform. In our system, we *never allow custom permissions to share groups with system permissions*. Additionally, the fact that Android internally treats all permissions the same way is an important limitation with security repercussions: platform developers tend to overlook the existence of custom permissions when handling permissions. The *custom permission upgrade attack* is an example of that. To overcome this, in our system, *system and custom permissions have distinct representations in the platform*. By doing this, we can differentiate between the two types of permissions during granting as well as enforcement and apply different strategies depending on the type of permissions.

Implementation. In order to decouple the two permission kinds, one could create separate object representations and data structures. This would require a complete redesign of the Android permission implementation throughout the Android framework which we think is impractical. Alternatively one could use existing fields in the current permission representation in Android which can give us information on the source of a permission. `BasePermission` class has a `sourcePackage` field that indicates the originating package of a permission. For system permissions defined in the platform manifest, this field is set to `android`, for system permissions defined in system packages, it *usually* starts with `com.android`, and for custom permissions it is the package name of the defining third-party app. However, the package name itself cannot be used to identify whether a package is system or third-party, as there are already system apps with package names not starting with `com.android` (e.g., browser) and even third-party apps can have package names starting with the system prefixes (`com.android` etc.). Hence, `sourcePackage` is not a reliable identifier of whether a permission is custom or system.

Instead, a both practical and robust approach, would be to extend the object representation of a permission with an additional member variable, indicating whether this permission is a custom permission. In Cusper, we implement this by augmenting the `BasePermission` and the `PackageParser.Permission` classes. The value of the new variable is assigned when an app's manifest is parsed (`PackageParser.java`) during installation or upgrade. If the app under investigation is untrusted (as indicated by its non-platform signature), we mark its permissions as custom. When parsing an untrusted app's manifest, we further check whether the app developer assigned a custom permission to a system permission group. In this case, we ignore the assignment, which results in the permission having no group. Moreover, if the app declares a custom permission group, we ensure it does not use a system permission group prefix (`android.permission-group`). In essence, we thwart the vulnerability while ensuring that even if future vulnerabilities manifest, there will be no escalation to system permissions.

After doing this, we can now track the creation of custom permissions by third-party apps. In order to particularly thwart the *Custom Permission Upgrade*, when a custom permission—which we can now effectively and efficiently differentiate from system permissions—is created with the protection level normal or signature (i.e., install permission), we simply set the `FLAG_PERMISSION_REVOKE_ON_UPGRADE` flag so that the permission will not be granted automatically if it is later updated to be a dangerous (runtime) permission.

B. Naming Conventions for Custom Permissions

Android allows third-party apps from different developers to declare permissions with the same name. The current solution is to never allow two permission declarations with the same name to exist on the device. While this sounds effective, it is unfortunately unable to stop the second attack we demonstrated: a definer app A declares a permission and another app B gets the permission granted. When the first app A is uninstalled and a victim app C comes in declaring and using the same permission to protect its components, it is vulnerable to confused deputy attacks from app B. We solve this problem by introducing an *internal* naming convention: we enforce that *all custom permission names are internally prefixed with the source id of the app that declares it*. Note that we do not expect app developers to change their practices. Custom permissions are still declared with their original names in the manifest files of apps to allow backward compatibility. However, in our system, the custom permission names are *translated* to `source_id : permission_name`. Thus, even if permission revocation such as in the above attack scenario fails, the attack will be rendered ineffective. This is because, as far as our system is concerned, the granted permission to app B will be an entirely different permission than the one app C uses to protect its components.

Choosing the appropriate source id is not straightforward. Consider for example using an app's package name as the `source_id`. This introduces two main problems. First, repackaged apps distributed on third-party application markets could use the package name of an app distributed on Google Play. Thus, the repackaged app could take the role of the *definer attack app* (see Section IV) and instigate a confused deputy attack. This is possible since the repackaged app and the victim app share the same package name and a permission created by the repackaged app cannot be distinguished from the one created by the victim if they share the same permission name.Second, using the package name as the `source_id` might break the utility of `signature` custom permissions for some use cases. For example, developers that have a set of applications which utilize each other's components, commonly use signature permissions to protect the components of their apps from others. Since the installation order cannot be determined in advance, each app in the set has to declare the same permission (i.e., same name and protection level) in their manifest to make sure this permission will be created in the system. If permissions are prefixed with their declarer app's package name, then the system will treat them as different permissions. Therefore, any attempted interaction will be wrongfully blocked.

In Cusper, we instead use the app's signature as the source id to prefix permission names. In the case of a repackaged app, assuming the malicious developer does not possess the private keys of the victim app developer, the declared permission will

be a different permission in the system than the victim's declared permission. Moreover, utility is preserved since custom permissions with the signature level will be treated as the same permission as long as they come from the same developer, which is exactly the purpose. Note that the same scheme can also be utilized for permission tree names.

Lastly, the official suggestion to Android app developers which declare custom permissions, is to use names that follow the reverse domain name paradigm (similar to the one for package names). However, Android does not enforce this naming convention. Even though it will ignore a permission declaration with the exact same name as an existing permission, it allows third-party apps to use a system permission name prefix (e.g., `android.permission`) in their custom permission declarations. Since permission names and groups are currently the only information the system has regarding the intention and source of the permission, this treatment is at the very least hazardous. In Cusper, we address this naturally as we add prefixes to permission names and *never allow a custom permission to use a name prefix reserved for system permissions*. Since we decouple the two types, we can now identify the type and origin of permissions, and readily enforce this rule. To maintain backward compatibility and ensure that the custom and system permission names are distinct, we also ignore system permission names for custom permissions (as the original system currently does).

Implementation. To thwart custom permission spoofing attacks of any sort (including our *Confused Deputy* attack), apart from distinguishing between custom and system permissions, we further need a way to track the origin of custom permissions and uniquely identify them in the system. Towards this end, we implement a naming convention for custom permissions in Cusper. Our implementation consists primarily of a permission name translation operation to prefix the permission names with their source id to ensure uniqueness in the system. This translation happens during installation and update for the names of the declared custom permissions and requested install time permissions, and at runtime for dangerous permissions and the permissions used to protect components (guards).

At the time of installation, we allow the system to parse declared custom permission names from an untrusted app's manifest; however, we translate their names to be prefixed with the hash of their app's signature before the actual permission is created in the system. In the case an app is signed with multiple keys, we sort the hashes of the keys and concatenate them. Note that one could attempt to perform the translation in place. For example, it could perform the translation while parsing a permission name from the manifest. However, at that point, the app's certificates are not yet collected. Doing so would incur non-negligible overhead since it involves a number of file opening and reading operations (`PackageParser.collectCertificates()`). Instead, we keep the parsed data unaltered until after the certificate collection normally happens. Then, we scan the package's meta-data to perform the necessary translations. Our approach resulted in great performance savings which keep Cusper's performance comparable to the original system (see Section VII).

Similarly, we first proceed to translate the names of the requested permissions during installation or update. This is done to correctly grant install time permissions (i.e., normal and signature). Note that a requested permission might not necessarily exist in the system at this time and therefore the permission name translation cannot happen. For example, an app that declares the permission might be installed at a later point in time. Since the declared permission will be translated, it will essentially be treated as a different permission than the one requested, violating application developers' expectations. This is not a problem with install time permissions: the permission correctly will not be granted as its definition does not exist on the system at the time of installation, which is on a par with the behavior of the original Android OS. In the case of dangerous permissions which are granted by the user at runtime, we need to dynamically check for existing declared permissions. Therefore, we perform a requested permission translation at runtime. In particular, when a dangerous permission is to be displayed to the user, we perform a scan on all declared permissions to find a custom permission with the same suffix as the requested permission. In our implementation, we do not allow declaration of custom permissions with the same name which ensures that the scan will result in only one possible permission. This is also the current design of Android which does not allow two apps to declare the same permission. Note, however, that since we prefix custom permissions, one could extend our system to allow multiple apps to use the same custom permission names. In case of an app requesting that permission, we could readily resolve the conflict if one of the declarers has the same signature. If all declarer apps come from different developers, a mechanism similar to Intent filters could be utilized to allow the user to select the appropriate declarer app.

It is worth noting that one could alternatively create a separate hash map for custom permissions (e.g., key-value pairs of (suffix, prefix)) to avoid the linear scan for suffix lookup. However, this hash map would need to be kept consistent with the original hash map for all declared permissions in the system (e.g., tracking addition/removal of permissions), which is hard to achieve since there are multiple places throughout the Android source code where this in-memory data structure is updated or sometimes even constructed from scratch from files in persistent storage. Hence, for the sake of consistency and not breaking utility, we prefer the linear scan method and do not change the structure of the in-memory data types for permissions. As we will show in our evaluation in section VII, this method does not result in any significant overhead.

Finally, as for permissions that are used to protect app components (guards), their name translation takes place at runtime during enforcement since a guard might not necessarily exist in the system at the time of installation.

VI. Android Permissions Alloy Model

As a part of the software development process, to verify that a piece of software meets the requirements, it is common practice in industry to rely *only* on software testing and not provide formal proofs of program correctness for the underlying model as formal verification is highly time consuming, difficult and expensive. However, we argue that fundamental components like a permission system are naturally worth more effort as any failure in such components can make way for critical security vulnerabilities or even render the security of

the whole system ineffective. Additionally, numerous security bug reports on similar issues present further proof that the current testing methodologies for Android permissions are not completely effective and a better way of proving program correctness is necessary. Hence, in this section, we focus on providing a formal model of Android (runtime) permissions and a formal proof for the correctness of our design for Cusper.

Formal verification allows us to systematically reason about our design of Cusper by covering many cases that would otherwise be difficult to investigate with static analysis or testing. This is not to say software testing is unnecessary when a formal correctness proof is provided. In fact, we still need software testing to verify that our implementation conforms to our proposed model (which is formally verified to be correct). On the other hand, "formal verification reduces the problem of confidence in program correctness to the problem of confidence in specification correctness" [16]. In other words, verification is performed not on the actual implementation but on a representation that is as close to the original implementation as possible. This is because it is challenging to perform formal verification at a scale required by source code, especially at the huge scale of the Android source code. Progress in the area does exist towards this for other programming languages [27], but such approaches are typically employed at the time of development, where the developer is required to annotate the code. This would be infeasible in our case where a large portion of the Android source code is already written. Additionally, correctness is proved *only* with respect to a set of fundamental properties that were defined based on the specification. There is *no* guarantee the system will behave correctly under any condition that was not a part of the defined properties or in case of redesigns of the system that might invalidate the model assumptions. Hence, the state of the art formal verification is not a silver bullet but still a best effort technique for proving correctness.

To analyze the security of Android permissions, previous work proposed formal models that correspond to the older Android versions which supported only install-time permissions [20], [21], [29]. Unfortunately, no such model exists for Android's currently-adopted runtime permissions. Hence, we build the first formal model of the Android runtime permissions and use it to verify the correctness of Cusper. This allows us to investigate Cusper under many cases such as all possible installation orders and app declarations. Note that having such a formal model has other benefits; for example, security researchers can use it to verify other properties of their interest on the runtime permission model. We based our model on the Alloy implementation of [20] as Alloy is a high-level specification language that is easy to interpret. However, we spend a significant amount of effort to extend this model to conform to the official specification for the new runtime permissions [4]. We analyze the security of the model through an automated analysis and show that when it is augmented with the design of Cusper, the fundamental security properties that were originally violated are satisfied. Our main contributions to the existing formal analysis on Android permissions can be summarized as follows:

- We updated the definitions of permission-related data abstractions in the model to comply with the new definitions of the runtime permission model.
- We significantly updated the permission granting scheme to comply with the complex granting scheme of the runtime model specification (e.g., permissions can be granted as either install or runtime).
- We implemented permission groups and permission granting on a group basis for dangerous permissions according to the runtime permission model.
- We enabled apps to dynamically change their manifest declarations and introduced an app update mechanism (apps could not be updated in the previous model).
- We identified and fixed the bugs in the existing model (e.g., missing signature checks for permissions).
- We demonstrated the existence of the aforementioned custom permission vulnerabilities in the model.
- We implemented our defense, Cusper, in the model to thwart these vulnerabilities and showed that Cusper satisfies the fundamental security properties.

We only model the parts of Android that concern permissions (e.g., permission-related data abstractions and operations) as it would be infeasible to model all of Android due to its large scope and complexity. Additionally, due to space limitations, we will be only be presenting the parts of our model that are key to understanding the general operation of the model or that significantly differ from the previous model. As for the actual Alloy implementation, we will present only a small part of it in this section. but the full implementation can be found in [17]. Our model can be dissected into three main parts: 1) abstractions related to permissions, device architecture, and applications on Android, 2) system operations that concern permissions, and 3) fundamental security properties to verify the correct behavior of the model.

A. Abstractions

In this section, we present the abstractions in our model that correspond to the representations of permissions, applications and devices on Android.

Permissions. Our `Permission` abstraction is on a par with what we described in Section II: each permission is associated with a name, a source package to indicate the defining package name, a protection level, and at most one permission group. Listing 2 presents the Alloy implementation for permissions, protection levels and permission group abstractions.

Applications and Components. Each `Application` on Android has a unique package name, a signature used by the developer to sign the app, and a target SDK level. Additionally, each app can comprise of several components, defined by the `set Component`, where a component can be one of the four Android components. Each component can be protected with a permission that we call `guard`. Furthermore, an application itself can have a `guard` to protect *all* of its components. Component guard takes precedence over the application guard in case they both exist. Each application can define a set of custom permissions and request a set of permissions.

In order to keep track of the permissions that are granted to apps, each app is associated with a `permissionsState` field that consists of a set of `PermissionData` objects which carry system flags and state information (e.g., whether a permission is granted as runtime or install time) regarding

each permission granted to the app at any time. This concept of "stateful" permissions is one of the major representation differences between the runtime and the install time models.

In order to implement an app update mechanism, we need to allow apps to dynamically change the declarations in their manifest file. To achieve this, we associate the fields that require to be mutable with an object from the totally-ordered set of `Time` in order to allow pairing of the fields with different values at different time steps. Obviously, package name and the signature should be immutable since these are the unique identifiers for apps and developers. The ability to dynamically change declarations is another important feature we introduce in our model, as this gives us the ability to update Android apps that are already installed on a given device. Listing 3 demonstrates the application-related abstractions in Alloy.

Device. Each `Device` comes with a set of built-in system permissions and a set of custom permissions defined by third-party apps. We also include a platform signature in our device representation to correctly perform signature checks when granting signature permissions defined by the system. Listing 4 illustrates the device abstraction in Alloy.

B. System Behavior

In this section, we describe the main system operations (i.e., Alloy predicates) that deal with Android permissions. By carefully investigating the Android source code, we have observed that most of these critical operations have either undergone a significant amount of change or been recently introduced with the runtime permission model. Specifically, apart from the significant change in abstractions, main operations such as install, uninstall and update now all require a scan over all the other existing applications to properly adjust their permissions whenever there is a change in the set of permissions (e.g., removal of a permission). Additionally, Android's permission granting scheme changed drastically with the introduction of runtime permissions. We aim to reflect all of these changes in our formal model. It is important to note that the order of statements in the presented predicates do not affect their correct operation since Alloy is a declarative (rather than imperative) language.

Grant Permissions. In contrast with the install permission model where permissions can be granted only at installation, in the runtime permission model, depending on the protection level and the app's target SDK level, permissions can be granted as either install or runtime permissions. Permissions with protection level normal and signature are always granted as install permissions, whereas for dangerous permissions, the behavior changes based on the target SDK level of the app being installed.

Table III shows the cases that can happen for when granting permissions. Each case will add/remove "stateful" permission objects for this app. As explained in IV, we observed implementation flaws in this part of the Android source code which make the aforementioned attacks possible and we mirrored the same erroneous behavior in our Alloy predicate for granting permissions (`grantPermissions`). For example, when denying "dangling" permissions to apps, we skip to revoke runtime permissions and only revoke install permissions as it is currently implemented in Android. Also, when a custom permission is updated from normal to danger-

Listing 2: Permissions and permission groups in the model

```
sig Permission {
   name: PermName,
   protectionLevel: ProtectionLevel,
   sourcePackage : PackageName,
   permGroup: lone PermGroupName// if perm belongs to
        group
}

abstract sig ProtectionLevel {}
one sig Normal, Dangerous, Signature extends
     ProtectionLevel {}

sig PermissionGroup {
   name: PermGroupName,
   perms: Permission -> Time // set of changing perms
}
```

Listing 3: Applications and components in the model

```
sig Application {
   packageName : PackageName,
   signature : AppSignature,
   declaredPerms: Permission -> Time, // custom
        permissions
   usesPerms: PermName -> Time, // requested permissions
   guard : lone PermName, // protects all components
   components: set Component,
   targetSDK: Int -> Time,
   // carries info regarding granted perms
   permissionsState: PermissionData -> Time
}
abstract sig Component { // def. shortened for brevity
   app: Application,
   guard: lone PermName, // protects only this component
}
sig PermissionData {
   perm: Permission,
   flags: Flags,
   isRuntime: Bool // runtime or install permission
}
```

Listing 4: An Android Device in the model

```
one sig Device {
   apps: Application -> Time,
   builtinPerms: set Permission, // system permissions
   customPerms: Permission -> Time, // custom permissions
   platformPackageName: one PackageName,
   platformSignature: AppSignature,
   builtinPermGroups: set PermissionGroup // system
        groups
}
```

TABLE III: Cases of grantPermissions. (* Precondition: Permission should exist as an install permission. † Deny if no other case matches.)

Grant Permission Cases	Protection Level	SDK Level	Grant As
Grant install	Normal	Any	Install
Grant install	Signature	Any	Install
Grant legacy install	Dangerous	<23	Install
Grant upgrade*	Dangerous	>=23	Runtime
Grant runtime	Dangerous	>=23	Runtime
Deny †	-	-	-

Listing 5: Granting permissions in the Alloy model

```
pred grantPermissions[app: Application, t, t': Time] {
   all pname: app.usesPerms.t' |
   pname in (Device.builtinPerms +
        Device.customPerms.t').name ⟹
   (let p = findPermissionByName[pname, t'] {
      p.protectionLevel = Normal // Case GRANT_INSTALL
          (normal)
         ⟹ grantInstallCase[p, app, t, t']
      else // Case GRANT_INSTALL (signature)
         p.protectionLevel = Signature and
         (verifySignatureForCustomPermission[p, app, t']
              or
         verifySignatureForBuiltinPermission[p, app])
            ⟹ grantInstallCase[p, app, t, t']
      //...other cases (grant runtime etc.)
      else // Case GRANT_DENY (deny permission)
         no pd: PermissionData | pd.perm.name = pname
              and pd in app.permissionsState.t'
   })
   else //permission doesnt exist, evoke (wrongfully
        only install perms)
      let pd = getPermissionData[pname, app, t]{
         hasPermissionData[pname, app, t]
         and pd.isRuntime = True
         and pd.perm.protectionLevel = Dangerous
            ⟹ pd in app.permissionsState.t'
      else
         no pd: PermissionData | pd.perm.name = pname
              and pd in app.permissionsState.t'
      }
   // make sure app cannot be granted unrequested
        permission
   no pd: app.permissionsState.t' |
      pd.perm.name not in app.usesPerms.t'
}
```

ous protection level, we treat this as an app SDK update—just as Android mistakenly does—and automatically grant the dangerous permission without user's consent. Note that our final formal model corrects these and other problematic issues according to Cusper's design. Listing 5 illustrates this operation; an example to how an individual case are handled can be found in Listing 10 in Appendix.

Installation. As a precondition to the `install` operation, the app being installed should not exist on the device and the list of apps after the operation is completed should consist strictly of all the apps before installation augmented by the new app. As a result of installation, custom permissions of this app will be added to the device. Just as it is currently handled in Android, our Alloy predicate for installation does not allow an app to declare a custom permission which has the same name as an existing permission on the device. Custom permissions that are declared to be part of system permission groups also get added to the respective groups. Finally, the permissions requested by the app will be granted to the app without affecting the permissions granted to other apps, that is "stateful" permission objects should not change.

Uninstallation. The `uninstall` operation removes an existing app and its custom permissions from the device and it readjusts the permissions granted to other apps in case there is a change in the set of custom permissions. In order to achieve this, `grantPermissions` is executed for all the apps on the device to reassign permissions and make sure that apps

Listing 6: Uninstall operation in the Alloy model

```
pred uninstall[t, t': Time, app: Application] {
   app in Device.apps.t // precondition
   // remove app from list
   Device.apps.t' = Device.apps.t - app
    // remove custom perms defined by app
   Device.customPerms.t' = Device.customPerms.t -
        app.declaredPerms.t
   all a : Application - app | grantPermissions[a, t, t']
   // remove permissions from permission groups
   all pg: Device.builtinPermGroups, p: Permission |
      p in pg.perms.t and p not in app.declaredPerms.t'
         ⟹ p in pg.perms.t' else p not in pg.perms.t'
}
```

Listing 7: Update operation in the Alloy model

```
pred update[t, t': Time, app: Application] {
   app in Device.apps.t // precondition
   Device.apps.t' = Device.apps.t
   // 1. Fix custom permissions on the device
   Device.customPerms.t' =
      Device.customPerms.t - app.declaredPerms.t +
           app.declaredPerms.t'
   // 2. Update all other apps if a perm is removed
   anyPermissionRemoved[t, t', app] ⟹
        updatePermissions[Application - app, t, t']
   else
      all a : Application - app | a.permissionsState.t'
           = a.permissionsState.t
   // 3. Regrant permissions for the current app
   grantPermissions[app, t, t']
   // 4. Adjust permission groups
   adjustPermissionGroups[app, t, t']
}
pred updatePermissions[apps: set Application, t,t':
     Time] {
   all app: apps | grantPermissions[app, t, t']
}
```

will be revoked the custom permissions of the removed app. This is a new behavior introduced by Google as a response to the previous bug reports regarding the issues with custom permissions. Listing 6 demonstrates uninstallation in Alloy.

Update. We introduce the update operation in our model since this operation is necessary to demonstrate the *Custom Permission Upgrade* vulnerability. Similar to `uninstall`, if a custom permission defined by this app is being removed from the manifest file with the update, we invoke `grantPermissions` for all other apps on the device in order to revoke this permission. Additionally, `grantPermissions` is executed for the app being updated to readjust its granted permissions, regardless of any change on the set of permissions. Permission groups are also readjusted such that the permissions removed from the app are also removed from their respective permission groups and the newly-added permissions are added to their respective groups. Listing 7 demonstrates the update operation in Alloy.

C. Correctness of Cusper

In order to verify the correctness of a proposed model, one needs to first compile a set of fundamental properties that need to be satisfied by the model under all conditions. Then, we need Alloy assertions, which are sanity checks to verify that this model behaves as expected with respect to these properties.

All security properties that had to be satisfied by the original Android permission model should also be satisfied by Cusper. Here, we focus on the properties that were violated by the original model. Our observation is that the new classes of vulnerabilities we discussed in Section IV are made possible because of the violation of two fundamental security properties that should always hold on the Android runtime permission model: 1) *dangerous runtime permissions should never be granted without user interaction*, 2) *there should never be an unauthorized application component access*. The first property means that a dangerous permission should only be granted with the user's approval for when the app's target API level is equal to or more than 23. The second one suggests that an app cannot access another app's components if it does not have the right permission for it. For example, if an app component is being protected by a signature custom permission, only the applications that possess the same signature as this app should be able to access the component.

In order to verify our observation, we built Alloy assertions of fundamental security properties and showed that the original model indeed does not satisfy the two aforementioned properties as the Alloy analyzer is able to produce counterexamples for both assertions indicating the violation of these properties. These correspond to the attack instances we have previously described. However, when the permission model is augmented to describe Cusper, we show that all the security properties are always satisfied, formally verifying the correctness of our design.

VII. System Evaluation

In the previous section, we verify the correctness of the formal model of Cusper which provides confidence regarding our design decisions. Next, we want to generate evidence regarding the practicality of Cusper's respective system implementation. Toward this end, we empirically evaluate our implementation of Cusper on Android, with respect to (a) its ability to thwart the specific attacks we presented and (b) its performance overhead incurred in the affected Android operations.

Effectiveness. To evaluate the effectiveness of Cusper, we carried out the two attacks we mentioned in Section IV on Cusper-augmented Android and showed that both attacks fail.

First, we attempted the *Custom Permission Upgrade* attack on Cusper-augmented Android and verified that the attack could no longer succeed. The user is correctly being consulted to grant the pemission by the system once a permission declaration changes from a normal protection level to dangerous. Moreover, we verified that a third-party app can neither assign a custom permission in a system permission group, nor declare a custom permission group using the system permission group naming convention. At the same time, normal operations of benign third-party and system apps are preserved.

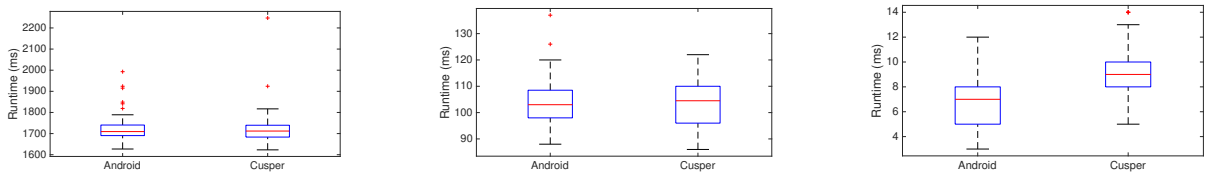
With respect to the *Confused Deputy* attack, using the the apps mentioned in Section III (i.e., Skype, CareZone) as well as other real-world apps, we verified that the attack can no longer succeed while again utility is preserved with Cusper. We further tested that permission revocation happens correctly when the declarer app is uninstalled. We also verified that declared custom permissions are prefixed by a hash of the app developer's signature, and the same happens for the custom permissions used to protect app components. Finally, we tested that granting normal and signature permissions at installation time, granting dangerous permissions at runtime, and using the permissions to access protected app or system components, happen correctly; hence, we do not break any utility.

Efficiency. In evaluating the performance of our system, we focused on the operations affected by our modifications. These include the app install operation, the app uninstall operation, runtime (dangerous) permission granting, and permission enforcement. We did not include our evaluation for the app update operation as its performance is similar to that of app install. We use a Nexus 5 phone running Android 6.0 (android-6.0.1_r77) for all our experiments. According to a previous study, Android users have on average 95 apps [6] installed on their devices. In addition, according to our prevalence study in Section III, apps create one custom permission on average. In order to evaluate Cusper under realistic conditions, we mimic this average case in our experiments and make sure the device contains 100 custom permissions along with all of the system permissions.

In our *app install* and *app uninstall* experiments, we used the *Android Debug Bridge* (adb) to install and uninstall an app of size 1.2 MB 100 times. The app declares a custom permission, with `protection-level` dangerous, uses the permission, and declares a service which is protected by that permission. We instrumented the `installPackageAsUser()` method in the `PackageManagerService` class to get the start time of app installation. We got the end time at the point before the system broadcasts the `ACTION_PACKAGE_ADDED` intent indicating the completion of the package installation. For app uninstallation, we instrumented the methods `deletePackage()` and `deletePackageX()` to get the start time and end time respectively. Figure 1a and Figure 1b illustrate our results.

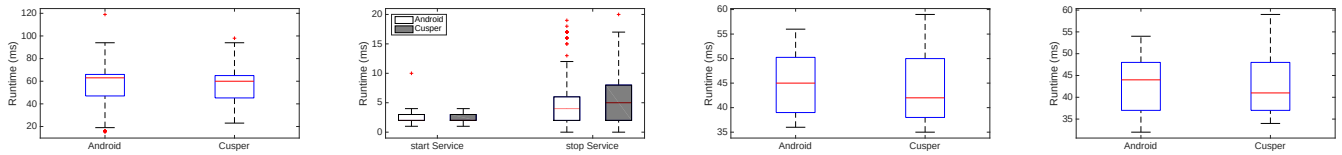
We compared our system with the unmodified Android version (*Android*). During installation, our system performs checks during parsing, performs the permission translation, and handles the permission revocation. While parsing, it checks and stores whether a permission definition is for a custom permission and it enforces the permission group checks. Then, it parses the in-memory meta-data of an app to perform a custom permission translation. Nonetheless, as shown in our evaluation, the performance overheads are indeed negligible: there is no statistically significant deviation between Cusper and the original version.

In addition, we evaluated the operation of granting a dangerous permission at runtime. We used an app which requests a custom permission previously defined in the system. Note that this is a process which involves user interaction: the system pops up a dialog box asking the user to grant or deny the permission request. We automated this process and ran this experiment 100 times. However, to avoid the unpredictable temporal variable of user interaction, we do not count the time between the display of the dialog box and the time the dialog box is removed. Our evaluation instrumentation is deployed in the `GrantPermissionsActivity` class. Figure 1c summarizes our results. Evidently, Cusper does not



(a) App Install (b) App Uninstall (c) Grant Dangerous Permission

Fig. 1: Performance evaluation of Cusper for installation, uninstallation and runtime (dangerous) permission granting.



(a) Activity (b) Service (c) Broadcast Receiver (d) Content Provider

Fig. 2: Performance evaluation of Cusper for component access.

incur any distinctive overhead.

Finally, we evaluated the performance of permission enforcement for custom permissions. For this case, we show performance results for accessing permission-protected app components of all kinds (i.e., activity, service, broadcast receiver, and content provider) in Figure 2. As can be seen, Cusper indeed incurs negligible overhead for all types of component invocation operations that require permission checks.

In summary, our modifications to the Android system are shown to have no perceivable performance overhead while they greatly strengthen the security of the Android OS.

VIII. Related Work

Previous work investigated Android Permissions and IPC security from many different perspectives.

IPC security on Android. Previous work has shown ways of exploiting IPC on Android to acquire unauthorized access to resources. In [26], the authors discuss the permission redelegation problem where an unprivileged app can access system resources through a privileged app via IPC. Additionally, [23] shows ways of exploiting the Intent mechanism to send or receive Intents in an unauthorized manner and get access to other app's private resources.

Analysis of Android Permissions. Wei et al studied the evolution of permissions across Android versions and showed that the set of permissions on Android tends to grow with every release [31]. Stowaway tool aims to detect if apps follow the least privilege for permission requests [25]. Additionally, [20] presents a formal analysis of Android permissions for older Android versions (<6.0) in Alloy; whereas [21], [29] introduce similar models in Coq.

Android Runtime Permissions. One of the early works on runtime permissions shows the necessity of having revocable, ask-on-first-use type permissions on Android, supported by user studies [25]. [33] provides an initial analysis on the runtime permission model and identifies several problems in this model that might open up ways for exploits. In [24], the authors analyze the undesirable side effects of switching to runtime permissions and introduce a tool called RevDroid that aims to identify these problems in apps. DP-transform provides a tool which helps developers adapt to the runtime model by automatically introducing the permission requests required by the model into the application code [22].

Android Custom Permissions. Although previous work has studied Android permissions, there is little work done specifically regarding Android custom permissions.The blog post in [9] discusses how the "first one wins" approach for custom permission definitions can create problems. Shin et al presents a viable attack on custom permissions by exploiting the naming convention problem of custom permissions [30], to which Google responded with bug fixes. In [28], the authors discuss how permissions can stay dormant on the Android platform, later to be revived by the installation of a permission definer app, and demonstrate attacks on custom permissions via the exploitation of this undesirable property.

IX. Conclusion

In this work, we investigate the Android runtime permission model and identify design flaws in custom permissions that can open up ways for adversaries to escalate their privileges to obtain unauthorized access to app components and platform resources. In order to systematically fix these flaws, we propose a defense mechanism, Cusper, that provides separation of trust between system and custom permissions and introduces an internal naming convention for custom permissions to effectively track their origins. To show the correctness of our approach, we first construct a formal model of the Android runtime permission model using Alloy specification

Listing 8: New permission representation according to Cusper

```
sig Permission {
   name: PermName,
   protectionLevel: ProtectionLevel,
   sourcePackage : PackageName,
   isCustomPermission: Bool, // new field  for Cusper
   permGroup: lone PermGroupName,
   sourceId: AppSignature // new field for Cusper
}
```

language and formally prove the existence of the vulnerabilities in this model. Then, we leverage this formal model to show that Cusper satisfies the fundamental security properties that were previously violated due to the custom permission vulnerabilities. Our evaluation of Cusper on Android shows that Cusper effectively fixes the existing vulnerabilities while inducing minimal overhead.

Acknowledgments. This work was supported in part by NSF CNS grants 15-13939, and 13-30491. The views expressed are those of the authors only.

APPENDIX

A. Implementation of Cusper in Alloy

In our formal model, we update the representation of permissions in order to reflect our design decisions. First, we add a boolean field to our `Permission` Alloy signature to indicate whether a permission is custom or system. Then, we also add a source id field, which will be used during permission enforcement to uniquely identify permissions. Here, we use app signature and not the hash of it for our formal model for simplicity. Updated permission abstraction can be seen in Listing 8.

For app and component guards, Cusper performs name translation at runtime during enforcement. Listing 9 represents the component invocation operation and the predicate in line 8 illustrates how permission enforcement is done according to Cusper. Whenever a component is being invoked, we retrieve the name of corresponding app or component permission, perform a lookup operation to find the corresponding Cusper name (see line 11). In addition, we update how we perform enforcement, so that when the system checks whether a calling app has the permission required to invoke a component, it will use both the permission name and the source id (see line 14).

Furthermore, we utilize the boolean custom permission indicator field to correctly set the `FLAG_PERMISSION_REVOKE_ON_UPGRADE` flag as shown in line 25 of Listing 10, which illustrates how `grantInstall` case is handled.

B. Other discovered attacks

In addition to the attacks we discussed in Section IV, we also discovered another attack on Android custom permissions that utilizes the lack of naming conventions for permissions to launch attacks on benign apps [14].

As described in Section II, custom permissions can also be created dynamically via the Android APIs. In this attack, the adversary spoofs the dynamic custom permissions of the victim. This attack is currently reproducible only on older versions

Listing 9: Component invocation with Cusper

```
pred invoke[t, t' : Time, caller, callee: Component]{
   caller.app + callee.app in Device.apps.t
   canCall[caller, callee, t]
   noChanges[t, t']
}

// Permission enforcement in the model according to
     Cusper
pred canCall[caller, callee: Component, t : Time] {
   let pname = guardedBy[callee],
      // name translation during enforcement for
           components
      source = getSourceId[callee, t],
      pd = getPermissionData[pname, caller.app, t] {
         pname in pd.perm.name
         source in pd.perm.sourceId
      }
}

// Return name of the permission protecting component
fun guardedBy : Component -> PermName {
   {c: Component, p: Name |
      // component-specific permission takes priority
           over the app-wide permission
      (p = c.guard.name) or (no c.guard and p =
           c.app.guard.name)
   }
}

// Guard names are translated during enforcement
fun getSourceId[c: Component, t: Time] : one
     AppSignature {
   {p: AppSignature |
      (p = findPermissionByName[c.guard.name,
           t].sourceId) or
      (no c.guard and p =
           findPermissionByName[c.app.guard.name,
           t].sourceId)
   }
}

// No changes in device and granted permissions
pred noChanges[t, t': Time] {
   Device.customPerms.t' = Device.customPerms.t
   Device.apps.t' = Device.apps.t
   all a : Application | a.permissionsState.t' =
        a.permissionsState.t
}
```

of (<6) due to some other issues in the new versions. Since we focused on modeling the new versions of Android and did not find strong evidence for the use of dynamic permissions by third-party developers as of now, we did not address this attack in our work. However, we believe it is worth presenting here as it demonstrates the extent of custom permission vulnerabilities and provides further proof that Android custom permissions are problematic at their current stage.

• *Steps to Produce the Attack.* In order to carry out the dynamic custom permission attack, the adversary builds an app that statically declares a custom permission that the victim app is planning to dynamically create via the `addPermission()` API method, which requires the static declaration of a permission tree with a specific domain name by the victim. The attack can work only if the installation of the attack app is performed before the victim app has an opportunity to dynamically create the permission of interest. After this, the attack app can gain unfettered access to signature protected components of the

Listing 10: Grant install case for grantPermissions predicate

```
pred grantInstallCase[p: Permission, app : Application,
     t, t' : Time] {
   hasRuntimePermission[p, app, t]
      ⟹ revokeRuntimePermission[p, app, t, t']
         grantInstallPermission[p, app, t, t']
   // no runtime permission should exist for this
        permission
   no pd: app.permissionsState.t' |
      pd.perm = p and pd.isRuntime = True
}

pred hasRuntimePermission[p: Permission, app :
     Application, t :  Time] {
   one pd: app.permissionsState.t |
      pd.perm.name = p.name and pd.isRuntime = True
}

pred revokeRuntimePermission[p: Permission, app :
     Application, t, t' : Time] {
   no pd: app.permissionsState.t' |
      pd.perm.name = p.name and pd.isRuntime = True
}

pred grantInstallPermission[p: Permission, app :
     Application, t, t' : Time] {
   one pd: app.permissionsState.t' |
      pd.perm = p and pd.isRuntime = False
      // Cusper fix: clear the flag to disallow
           automatic upgrade to runtime for custom
           permissions
      p.isCustomPermission = False ⟹
           clearPermFlags[pd, t']
      else setPermFlags[pd, t']
}

pred clearPermFlags[pd: PermissionData, t: Time] {
   pd.flags.FLAG_PERMISSION_REVOKE_ON_UPGRADE = False
   // clear other flags..
}

pred setPermFlags[pd: PermissionData, t: Time] {
   pd.flags.FLAG_PERMISSION_REVOKE_ON_UPGRADE = True
   // set other flags..
}
```

victim app, while the victim will not be able to dynamically create its own custom permission anymore since it is already defined in the system.

• *Internals of the Attack.* Android does not seem to perform any checks on the availability of the permission names for statically defined custom permissions against the permission tree names on the device. In other words, an app can still statically declare a custom permission with the domain name of a permission tree declared by another app, even though the operation would fail if the app tried to declare this permission dynamically (i.e., throws `SecurityException` stating the tree belongs to another app). Same kind of name translation approach we presented in Cusper can be used for the names of permission trees to resolve this problem.

As we mentioned, this attack works only on older Android versions (<6.0) since the new versions require `MANAGE_USERS` or `CREATE_USERS` permissions for the `addPermission()` API to properly work even though this behavior is not documented in the Android developer guides. We believe this itself might be an undesired behavior that was introduced by the system developers while implementing possibly the multi-user framework in Android; hence, if this implementation is changed, the dynamic custom permission spoofing vulnerability should emerge on Android 6.0 and onward.

REFERENCES

[1] "Alloy: A language and tool for relational models." http://alloy.mit.edu.

[2] "Android : Requesting permissions," https://tinyurl.com/y8gp4dn6.

[3] "Android dashboard," https://tinyurl.com/qfquw3s.

[4] "Android permissions." https://tinyurl.com/y863owbb.

[5] "Android plugin application," https://tinyurl.com/ycfd9pot.

[6] "Android users have an average of 95 apps installed on their phones, according to yahoo aviate data," https://tinyurl.com/ybc7dqbn.

[7] "Apktool decompiler," http://ibotpeaches.github.io/Apktool/.

[8] "Creating apps with plugin architecture," https://tinyurl.com/ydfdk9z7.

[9] "Custom permission vulnerabilities." https://tinyurl.com/y7yoae52.

[10] "Dex2jar." https://github.com/pxb1988/dex2jar.

[11] "Google cloud messaging," https://tinyurl.com/ybocrrqw.

[12] "Jd-gui." http://jd.benow.ca/.

[13] "Privilege escalation by exploiting fcfs property of custom permissions," https://issuetracker.google.com/issues/37131935.

[14] "Privilege escalation by exploiting permission trees and dynamic custom permissions," https://issuetracker.google.com/issues/37324008.

[15] "Privilege escalation through custom permission update," https://issuetracker.google.com/issues/37130844.

[16] "Program correctness, the specification," https://tinyurl.com/y8r8cze8.

[17] "Resolving the predicament of android custom permissions," https://sites.google.com/view/cusper-custom-permissions/home.

[18] "Upload applications to appaloosa," https://tinyurl.com/y94pb3cv.

[19] "Yoga guru," https://tinyurl.com/yb3dqopp.

[20] H. Bagheri, E. Kang, S. Malek, and D. Jackson, "Detection of design flaws in the android permission protocol through bounded verification," in *International Symposium on Formal Methods*, 2015.

[21] G. Betarte, J. Campo, C. Luna, and A. Romano, "Verifying android's permission model," in *Theoretical Aspects of Computing*, 2015.

[22] D. Bogdanas, N. Nelson, and D. Dig, "Analysis and transformations in support of android privacy," Tech. Rep., 2016.

[23] E. Chin, A. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *MobiSys*, 2011.

[24] Z. Fang, W. Han, D. Li, Z. Guo, D. Guo, X. Wang, Z. Qian, and H. Chen, "revdroid: code analysis of the side effects after dynamic permission revocation of android apps," in *Asia CCS*, 2016.

[25] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *CCS*, 2011.

[26] A. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: and defenses." in *USENIX Security*, 2011.

[27] K. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming AI and Reasoning*, 2010.

[28] J. Sellwood and J. Crampton, "Sleeping android: The danger of dormant permissions," in *SPSM*, 2013.

[29] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "A formal model to analyze the permission authorization and enforcement in the android framework," in *SocialCom*, 2010.

[30] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka, "A small but non-negligible flaw in the android permission scheme," in *POLICY*, 2010.

[31] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *ACSAC*, 2012.

[32] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading your android, elevating my malware: Privilege escalation through mobile os updating," in *IEEE Security and Privacy*, 2014.

[33] Y. Zhauniarovich and O. Gadyatskaya, "Small changes, big changes: an updated view on the android permission system," in *RAID*, 2016.