Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References

Yousra Aafer[*,1], Nan Zhang[*,2], Zhongwen Zhang[3], Xiao Zhang[1], Kai Chen[2,3]
XiaoFeng Wang[2], Xiaoyong Zhou[4], Wenliang Du[1], Michael Grace[4]
[1]Syracuse University
[2]Indiana University, Bloomington
[3]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences
[4]Samsung Research America
{yaafer, xzhang35, wedu}@syr.edu, {nz3, xw7}@indiana.edu, {zhangzhongwen, chenkai}@iie.ac.cn, {x.zhou01, m1.grace}@samsung.com

ABSTRACT

Android is characterized by the complicated relations among its components and apps, through which one party interacts with the other (e.g., starting its activity) by referring to its attributes like package, activity, service, action names, authorities and permissions. Such relations can be easily compromised during a customization: e.g., when an app is removed to fit an Android version to a new device model, while references to the app remain inside that OS. This conflict between the decentralized, unregulated Android customization process and the interdependency among different Android components and apps leads to the pervasiveness of *hanging attribute references* (Hares), a type of vulnerabilities never investigated before. In our research, we show that popular Android devices are riddled with such flaws, which often have serious security implications: when an attribute (e.g., a package/authority/action name) is used on a device but the party defining it has been removed, a malicious app can fill the gap to acquire critical system capabilities, by simply disguising as the owner of the attribute.

More specifically, we discovered in our research that on various Android devices, the malware can exploit their Hares to steal the user's voice notes, control the screen unlock process, replace Google Email's account settings activity and collect or even modify the user's contact without proper permissions. We further designed and implemented *Harehunter*, a new tool for automatic detection of Hares by comparing attributes defined with those used, and analyzing the references to undefined attributes to determine whether they have been protected (e.g., by signature checking). On the factory images for 97 most popular Android devices, Harehunter discovered 21557 likely Hare flaws, demonstrating the significant impacts of the problem. To mitigate the hazards, we further developed an app for detecting the attempts to exploit Hares on different devices and provide the guidance for avoiding this pitfall when building future systems.

[*]The two lead authors are ordered alphabetically.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813648.

1. INTRODUCTION

Never before has any operating system (OS) been so popular and diverse as Android. So far, over one billion mobile devices are running the OS, whose official versions (Android Open Source Project or AOSP) have been aggressively *customized* into thousands of system images by almost everyone in the production chain, hardware manufacturers (e.g., Qualcomm), device manufacturers (e.g., Samsung, LG, HTC), carriers (e.g., Verizon, AT&T, etc.) and others, for the purposes of tailoring the OS to different hardware platforms, countries/regions and other needs. This practice has led to a highly *fragmented* ecosystem, a wild wild west when it comes to the supports for Android applications (*app* for short) to operate across different devices (e.g., phones from different manufacturers). To put this unregulated ecosystem under control, Google has launched the Android Compatibility Program [1] to guide the customization process. This effort, however, fails to address a serious concern that comes with the aggressive customization: security hazards could arise when proper precautions have not been taken in changing the OS and apps to fit different devices.

Hares in the Wild Wild Android. For example, the manufacturer may customize a smartphone OS for a tablet without 3G capability by removing some components, including the messaging and telephony provider apps; however, in the presence of the apps capable of receiving SMS/MMS messages, malware on the tablet could impersonate the missing telephony providers (using its SMS/MMS authorities) to communicate with those apps and their users (e.g., cheating them into believing that their friends are sending them messages from the VoIP channel). Fundamentally, what causes the problem here is the intrinsic interdependent relations between different Android components (apps and framework services), which connect one party to another through references to the latter's attributes such as package, activities, services names, authorities of content providers and permissions: e.g., `startActivity` called by one app to invoke another's activity (whose name is specified through `setClassName`). Customizations made to those components, if not well thought-out, could easily break some of such relations, resulting in the references to nonexisting attributes (e.g., the authorities of the SMS/MMS providers not on the tablet). We call them *hanging attribute references*, or simply *Hare*.

As a side effect of the Android fragmentation, Hares could also be brought in by the third-party developer who designs her app to run on various Android versions, with or without certain service components it utilizes. For example, a reference to the nonexisting messaging content provider could also be embedded in a third-party app meant to work on both the smartphone and the tablet.

Compared with the customization flaws discovered in the prior research, which are about misconfigurations of Linux-layer device drivers [26], the hanging reference is a framework-layer issue and potentially more pervasive, given the fact that system apps on that layer have always been the focus of a customization [24]. However, such a problem has *never* been studied, whose security implications, scope and magnitude, therefore, are not clear at all.

Our findings. This new type of vulnerabilities have first been discovered in our research, which shows that they are indeed both security-critical and extensive. More specifically, we found that a Hare on Note 8.0 can be exploited to steal the user's voice note and another flaw on Tab S 8.4 allows a malicious app to impersonate the Facelock guard to gain control on the user's login authentication. The popular Tango app contains an unprotected reference to the missing `sms`, which can be leveraged to steal the user's messages. Also through hijacking various packages, activities or missing content providers, the adversary is able to replace Google Email's internal account settings interface, inject activities into LG FileManager and LG CloudHub to steal the user's password, and trick S-Voice into launching a malicious program whenever the user needs to use the pre-installed voice recorder. Moreover, on Note 3 (phone) and Note 8.0 (tablet), a Hare related to an absent permission can be exploited to steal all the contact information (e.g., email, phone number, etc.) of the device user and even tamper with its content (e.g., changing a friend's phone number, email and URL to those under the adversary's control), when the malicious app does not have the privilege to do so.

To understand the scope and magnitude of the security hazards introduced by Hares, we ran a new tool to automatically evaluate over 97 OS images for Google, Samsung, LG, HTC and Motorola devices. This measurement study shows that unprotected Hares exist on every single device we tested and are completely open to exploits. Also interestingly, we found that though such flaws can be caused by carriers and other parties, apparently they have been primarily introduced by the manufacturers when customizing the same OS to different device models. Further, the problems are still pervasive even on the latest OS versions and phone models, across different manufacturers, indicating that this security risk has yet come to their attentions. These findings point to the gravity of such security hazards and the urgent need to develop effective solutions to address them. We reported the high-profile Hares discovered in our research to Google, Samsung and other related organizations, who all acknowledged the importance of our findings. Video demos of some attacks are posted on a private website [6].

Detection and protection. Our measurement study was made possible by *Harehunter*, a new tool for automatic detection of the Hare vulnerabilities within system apps. For this purpose, Harehunter first performs a differential analysis, comparing all the attributes defined by the system apps on an Android image with those referred to by them. Any discrepancy between the definitions and the references reveals a Hare risk. This instance is further evaluated through automatic program analysis to find out whether it is actually protected: e.g., whether a package's signature has been verified before its activity is invoked. If not, then the problem is reported as a likely Hare (*LHare*) case. Running Harehunter on 97 popular device images, we discovered 21557 likely Hares within 3450 vulnerable system apps, which have been documented in a database. This database is utilized by a protecting app we developed, called *HareGuard*, to inspect every newly installed app on these devices, identifying the suspicious ones that attempt to exploit the Hares there, thereby securing the device even before its manufacturer can fix the problems. Our study further evaluated the efficacy and performance of Harehunter and HareGuard, which were both shown to be highly effective. We further discussed the lessons learnt from our study and the effort that needs to be made to avoid similar problems in the development of future systems.

Contributions. The scientific contributions of the paper are outlined below:

- *New findings*. We discovered *Hare*, a new category of Android vulnerabilities never known before. The problems are not isolated, random bugs and actually caused by the fundamental conflict between the under-regulated Android customization process and the complicated interdependencies among the apps and components on different Android systems. Our research reveals the significant impacts of the flaws, which could lead to privilege escalation or information leaks on almost every popular Android device we inspected. Given the serious consequences of the flaws once they are exploited, our research highlights the importance of incorporating proper security checks into the ongoing effort on regulating the highly fragmented Android ecosystem.

- *New techniques*. We developed a set of new techniques for automatically detecting Hares within different Android versions and protecting them against exploits. These tools can be utilized by the device manufacturers and other parties to improve the security quality of their custom OSes. Also the Android user can get immediate protection for her device by simply installing our user-land app, before the manufacturers are able to fix their problems. More importantly, the design of our flaw detection mechanism could inspire the follow-up research on automatic recovery of the interdependent relations on different OS versions to support a securer customization process.

- *Implementation and evaluation*. We implemented Harehunter and HareGuard, and evaluated them on a large number of customized Android versions.

Roadmap. The rest of the paper is organized as follow: Section 2 introduces the background of our study; Section 3 elaborates different Hare flaws and their security implications through a few high-profile examples; Section 4 describes our detection and protection techniques, and our large-scale measurement study using the tool; Section 5 discusses the lessons learnt from our study; Section 6 compares our work with related prior research and Section 7 concludes the whole paper.

2. BACKGROUND

Android Fragmentation. As mentioned earlier, the AOSP baselines have been intensively customized by different parties. For each new version released by Google, hardware manufacturers such as Qualcomm first change the OS to support their products, and then the device manufacturers like Samsung, LG and HTC modify the version to enrich its functionalities and tailor it to different devices (phones, tablets for different languages, countries/regions, etc.). This customization process continues when the devices reach carriers, such as Verizon and AT&T, which revise services or add in new apps to distinguish their phones or tablets from those of others. Further complicating the situation is Android upgrades: since September 2008, 21 official versions (from 1.0 to 5.1) have been released; such rapid updates outpace the distribution of the hardware platforms capable of supporting the new systems. As a result, a large number of custom Android systems have been built (over 10,000 for Samsung alone), and many of them, at various version levels, co-exist in today's market.

Prior research shows that the most heavily-customized components are actually a device's pre-installed apps. As an example, a study found that across the smartphones produced by major device manufacturers (Samsung, HTC, LG, Sony), only 18% of the pre-installed apps were from their corresponding AOSP baselines, 65% from the manufacturers and 17% added there by other parties such as carriers [24]. Although it has been reported that some of these apps contain known vulnerabilities, such as re-delegation [13], content leaks [15] and permission overprivilege [12], little has been done to understand whether new, customization-specific security flaws have also been introduced, which our study aimed at.

Attribute reference and Android security model. Different Android components (apps or their internal activities, services, content providers, receivers, etc.) are connected together by Inter-Component Communication (ICC), such as *Intent messaging*. An Intent is a message that describes the operations to be performed by the recipient: for example, `startActivity` that triggers an activity (a set of user-interface related operations) associated with an app. The app's package name and activity name can be specified through the Intent, using the method `setPackage`, `setClassName`, `setComponent`, etc. Here the reference from one component to another happens through the latter's attributes, i.e., the package name and activity name. When these attributes have not been set for the communication, the Intent is *implicit* and needs to be resolved by the OS to locate the recipient capable of handling it. In this case, the sender needs to provide an *action* (e.g., `android.intent.action.Edit` through `setAction`) and other parameters (such as *data*), and the recipient is supposed to declare an *Intent filter* for its component (activity, service, receiver) that matches these parameters in order to get the Intent. Another important Android component is *content provider*, which manages access to an app's databases (structured datasets). To operate on another app's content provider, one must get an URI "`content://authorityname/path`", through which the database table corresponding to the `path` can be read (`query`) and written (e.g., `insert`), under the consent of its owner. In all such ICC communication, once the target of a reference (e.g., package name, activity name, action name and authority name) is not present on the same system, the reference becomes hanging, which can have serious security implications (Section 3).

Android protects its information assets through an application-sandbox and permission model, in which every app runs within its own compartment (enforced through the Linux user protection) and can only access sensitive global resources and other app's components (content provider, service, activity, broadcast receiver) with proper permissions. More specifically, the app can specify for each of its components a permission and only process the message or service request from the parties with the permission. For example, a content provider can be guarded with a `readPermission` and a `writePermission`; a broadcast receiver can be configured to get the message only from those with a specific permission. Such permission protection is mostly set statically within an app's manifest file, but it can also be specified programmatically, using the APIs like `checkPermission`. An app that wants to obtain such a permission needs to ask for the user's consent. However, when the party that defines such a permission does not exist on a custom version, the permission protection becomes hanging: anyone that defines the permission can silently gain the privilege to access protected app components.

Adversary Model. We consider a scenario where a malicious app has been installed on the target device. However, the app does not need to have any suspicious permissions. Actually, in the case of hanging permission protection, it can define the missing permission by its own to launch all kinds of attacks. To deliver the information stolen from the device, the app needs the communication capability. This can be done explicitly by asking for the network permission, which has been requested by almost all apps. Alternatively, the malicious app can utilize other channels, such as browser, to send the data out, as demonstrated in the prior work [8].

3. EXPLOITING HARES

As mentioned earlier, a hanging attribute reference could be an ICC call to a nonexisting package, activity, service (which could be implicitly specified by the action or data filters) or authority of a content provider, or the use of a missing permission to protect an app component (service, activity, broadcast receiver and content provider). In the presence of such a reference, a malicious app that claims its target attribute could gain access to the information assets exposed by the ICC or guarded by the permission. More specifically, when the reference is not *guarded* along the execution path involving the Hare, that is, no validation of the existence and legitimacy of the attribute before using it, the malware that acquires the attribute (e.g. package/authority/permission name) automatically obtains the privilege associated with the attribute and becomes entitled to get sensitive messages from the sender, utilize its component, etc. Examples of the attacks are presented in the rest of the section.

It is important to note that not every hanging reference is exploitable. It can be protected by verifying the existence of the package that supposes to define it and then verifying its signature (extracted through `getPackageInfo` with flag `GET_SIGNATURES`), or its application info `FLAG_SYSTEM`, or by checking the current device's model, country code or other properties (e.g. `getProperty`). The presence of such protection was identified in our study through automatic code analysis (Section 4.1). On the other hand, if the security check is not in place, a Hare becomes vulnerable to exploits, even though it could still be nontrivial to find the conditions for triggering the code.

In our research, we systematically analyzed 97 Android factory images from major device manufacturers (Google, Samsung, LG, HTC, Motorola), and found 21557 hanging attribute references that are likely to be vulnerable (Section 4.2). To understand the security risks they may pose, we built end-to-end attacks on a few Hare instances. Except a small set of them that were discovered manually, which motivated the whole research, most of the Hares, particularly those within pre-installed apps, were detected automatically using Harehunter described in Section 4.1. We reported all these security-critical flaws to the manufacturers, including Samsung, LG, Google and HTC. Some of them have already been fixed. Following we elaborate what we learnt about such vulnerabilities and the consequences once exploited. Also, some of the attack apps we built passed the security check of Google Play, while the rest were accepted by other leading app markets like Amazon Appstore and even Samsung's own app store, which demonstrates that the security risks posed by these vulnerabilities are realistic[1].

3.1 Package, Action and Activity Hijacking

Among all the Hares discovered in our research, the hanging references often point to package names and actions. These attributes play an important role in Hare exploits, even when the main targets are other attributes. This is because a missing package can be

[1]To avoid causing any damage to those inadvertently downloading our apps, we either removed them as soon as they were approved by the app markets or make sure that they do not send out sensitive user data or perform other actions that could harm the user.

the owner of absent activities, and actions often need to be specified for receiving the Intent caused by vulnerable references. Moreover, references to nonexisting activities were also found to be pervasive. By exploiting these vulnerabilities, the malware can let a trusted source (a system service or app) invoke a malicious activity, making it look pretty trustworthy to the user. This enables a variety of highly realistic phishing attacks that can lead to disclosure of sensitive data, such as passwords. Following we elaborate a few examples for such Hare flaws and our end-to-end attacks.

A limitation of the exploits on package names is that once the owners of the targeted names are already on Google Play, our attack apps can no longer be uploaded there, as the Play Store does not allow two apps to have the same package names. This restriction, however, is not applied to other attributes. So those not relying on package names can still get into the Store. Also, third-party app stores like Amazon and Samsung typically do not have the target apps of our attacks and therefore the code for hijacking their package names can often be accepted there. Interestingly, we even managed to publish some of the attack apps on Samsung App Store, even though they performed days of manual analysis on our submission.

Stealing voice note. S-Voice is a personal assistant and knowledge navigator service app pre-installed on certain devices (e.g. Note 8.0). One of its features is voice memo: the user can simply say "take memo" or "take note" to activate the functionality and follow the instruction ("please say your note") to record her note. After the note is taken, the app first checks whether another system app `com.vendor.android.app.memo` (`memo` for short) exists, and if so, connects itself to the latter's service by calling `bindService` using an action name specified by its Intent filter. This hands over the note to the `memo` app. In the case that the app is not there, S-Voice looks for another system service to handle the voice note.

We found that S-Voice fails to verify the signature of `memo` when referring to it. As a result, on the device where the app is missing, the references to both its package name and action (through `bindService`) become hanging. A malicious app can then impersonate `memo` using its package/action names to steal the user's voice note. In our research, we built an attack app with the package name of `memo` that defines a service with the action Intent filter `com.vendor.android.intent.action.MEMO_SERVICE`. The app also includes an interface for receiving service requests and data from S-Voice. We ran it on top of Note 8.0, a device that does not have the `memo` app, and successfully stole the voice note recorded from the user. Our attack app was successfully uploaded to Amazon Appstore. A video demo is posted on our private website [6].

Cheating AOSP keyguard. Prior to 5.0 (only around 10% of the market share [5]), all AOSP versions after 2.3 support face-based screen unlock, which is done through a system app called *Facelock* (`com.android.facelock`). Once this biometric authentication option is chosen by the user, the Android Keyguard service will bind itself to a Facelock service, enabling the user to use her face and the front camera to unlock her device. More specifically, whenever the security settings fragment within the Settings app is created, Settings app will invoke `isBiometricWeakInstalled` in `LockPatternUtils` framework class to check if the Facelock app is installed. If so, it will add Facelock as an available screen lock option. Later when the user clicks on the option, Settings sends an Intent to Facelock for configuration. After this step is done (which also includes configuring a back-up PIN or Pattern), FaceUnlock is set as the lock screen option. Under the option, whenever the user clicks on a locked phone, Keyguard will bind itself to the face-unlocking service by sending an Intent specifying the action `com.android.internal.policy.IfaceLockInterface` to the Facelock app. The screen is unlocked once Facelock informs Keyguard that the user is authenticated.

A problem here is that on all the AOSP versions prior to 5.0 supporting the FaceUnlock option, the Android framework class `LockPatternUtils` fails to verify the signature of the Facelock app. As a result, on the device model where the app is not present, the reference to its package name becomes hanging and can be exploited by a malicious app. In our research, we installed on Tab S 8.4 an attack app that impersonated `com.android.facelock` along with the required setup activities and unlocking service, and successfully activated the FaceUnlock option. When the option was selected, the attacker app was invoked and consequently set as a phone lock. When the user wished to unlock the screen, the attacker app utilized the action `com.android.internal.policy.IfaceLockInterface` to cheat Keyguard into binding to its service. As a result, the malware gained full control of the screen unlock process and was able to expose the device to whoever it wanted. This attack poses a particularly serious threat to the multiuser framework provided by Google from Android 4.2, where an attacker purposely installs the malicious Facelock app as a backdoor to other user's accounts. In fact, once installed in the malicious user's account, the app will be immediately enabled on other users accounts as discussed in the prior research [22]. Note that though Lollipop and the later versions no longer offer FaceUnlock, and instead push the support for the functionality to device manufacturers, this security flaw still has a significant impact, given the fact that around 90% of the devices in the market are running the versions below 5.0. The attack app was uploaded to Amazon Appstore and its demo is on the website [6].

Faking Dropbox on LG. LG FileManager is a system app on LG devices that helps the user manage her file system. It also supports the use of Dropbox, which can be opened by clicking on a button with the Dropbox icon. Interestingly, on LG G3 factory image, our analyzer (Section 4.1) found that the button actually first tries to launch an activity within `com.vcast.manager`, a Verizon cloud app, and only goes to the Dropbox's web login page once the attempt fails. This program logic could be designed for the devices distributed by Verizon but leave the reference to the service hanging on those with other carriers and development phones.

In our research, we built an attack app to impersonate `com.vcast.manager` and hijacked the activity pointed to by the hanging reference. Since LG FileManager does not check the target app's signature before starting its activity, it blindly invoked our app whenever the user clicked on the "Dropbox" button. This gives the app an opportunity to show up a fake Dropbox login activity to steal the user's credentials.

Replacing official recorder. S-Voice performs voice recording using a default recorder. There are two such recorders, `com.sec.android.app.voicerecorder` and `com.sec.android.app.voicenote`. What happens is that S-Voice first attempts to use the activity of `voicerecorder` and only when this fails (the app does not exist), it switches to `voicenote`. Again, such a two-choose-one process does not involve proper authentication of the target. This allowed us to construct an attack app impersonating `voicerecorder` app with the activity `VoiceRecorderMainActivity` to control the target of the reference. On Note 8.0, our experiment shows that the attacker's activity was always invoked, even in the presence of `voicenote`, which enabled it to record sensitive user conversation or perform a phishing attack.

Hulu on watch. WatchON is a popular app that allows its user to view the TV programs in their TV or select movies from the Video-on-Demand service that integrates Hulu, Vudu, popcornflix, etc. Once the user clicks on a Hulu movie, WatchON sends an implicit Intent to launch Hulu's activity. For some movies requiring a HuluPlus account, the user will be redirected to an upgrade activity where she can pay to be upgraded to the HuluPlus status.

The problem here is that the references to the Hulu' activities were found to be hanging in our research: even though WatchON indeed checks whether Hulu exists before sending the implicit Intent, it fails to verify the app's signature. Therefore, we were able to build a malicious app that masqueraded as Hulu and set an Intent filter with action `hulu.intent.action.LAUNCH_VIDEO_ID` to get the upgrade Intent. Through launching a malicious activity, we could cheat the user into entering her login credentials for Hulu. More seriously, when she actually clicked on a paid movie, the malware displayed an upgrade activity, asking for her credit-card information. Since all these activities were triggered by WatchON, the malware is very likely to get what it wants. We successfully uploaded this attack app to Samsung App Store, which analyzed our code both statically and dynamically for days.

3.2 Content-Provider Capture

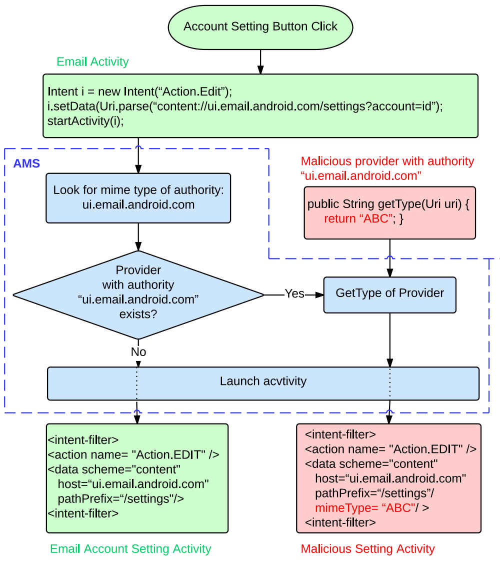
Just like actions and activities, content providers are also extensively used for inter-app and app-framework interactions. Specifically, an app may query another app's content provider by directly referring to its *authority*, one or more URIs formatted in a Java-style naming convention: e.g., `com.example.provider.imageprovider`. However, just like what happens to other attributes, such a reference (to the authority) can also become hanging, when the related provider is in absence on a device. This opens another avenue for the Hare exploit, when a malicious app strategically defines a content provider to misinform the querier.

Note that unlike package name, duplicated authority names are not forbidden on the Play Store. As a result, all our attack apps were successfully uploaded to Google Play. Following we describe a few attacks on the Hares of this type.

Hijacking Intent invocations. A surprising finding of our research is that a subtle content-provider Hare within Google Email (version 6.3-1218562) allows a malicious app to completely replace its internal account settings with a malicious activity. Specifically, Google Email, the standard email application on every Google phone, lets the user configure different email accounts (Gmail, exchange, etc.) through a Settings interface. To invoke this activity, the app sends an implicit Intent with action `android.intent.action.EDIT` and data `content://ui.email.android.com/settings?account=x`, where x is the email account ID used to inform the account settings activity which email's setting to edit. These two parameters are specified within the account settings activity's Intent filter, as illustrated in the following code snippet:

```
<!-- Account Settings Intent Filters-->
<activity
  android:name=".activity.setup.AccountSettings"
       android:exported="true">
  <intent-filter>
      <action android:name="android.intent.action.EDIT"/>
      <category android:name=
           "android.intent.category.DEFAULT"/>
      <data android:scheme="content"
        android:host="ui.email.android.com"
        android:pathPrefix="/settings"/>
  </intent-filter>
```

This implicit Intent can be received by any app that specifies the above Intent filter for its activity. However, when this happens,


Account Setting Button Click
Email Activity
AMS
Look for mime type of authority: ui.email.android.com
Provider with authority "ui.email.android.com" exists?
Yes
No
GetType of Provider
Launch acvtivity
Malicious provider with authority "ui.email.android.com"
Email Account Setting Activity
Malicious Setting Activity

```
Intent i = new Intent("Action.Edit");
i.setData(Uri.parse("content://ui.email.android.com/settings?account=id");
startActivity(i);
```

```
public String getType(Uri uri) {
    return "ABC"; }
```

```
<intent-filter>
<action name= "Action.EDIT" />
<data scheme="content"
  host="ui.email.android.com"
  pathPrefix="/settings"/>
<intent-filter>
```

```
<intent-filter>
<action name= "Action.EDIT" />
<data scheme="content"
  host="ui.email.android.com"
  pathPrefix="/settings"/
  mimeType= "ABC"/ >
<intent-filter>
```

Figure 1: Exploiting a Hare Authority to Hijack Email Account Settings Activity

Android pops up a window that lists all eligible receivers to let the user select. What we want to do here is to circumvent this protection, making a malicious app the only qualified recipient.

To this end, we analyzed the `data` part of the Intent filter in the code snippet above and checked how the `ActivityManagerService` (`AMS` for short) resolves the Intent sent to this Intent filter. Figure 1 depicts the Intent resolution steps in this scenario. If the data's scheme is `content`, `AMS` will try to infer the MIME (Multi-Purpose Internet Mail Extension) of the attached data to identify the recipient that can handle this type: the data type here is supposed to be given by the content provider `ui.email.android.com`. However, this provider does not exist and as a result, the type is typically ignored and the Intent is sent to whoever define the `action.EDIT` and data filter (with `scheme="content"`) without a specified MIME type (as `No` branch in Figure 1).

The security risk here is that the reference to the content provider is hanging and can be exploited by a malicious app defining that provider. What the malware can do is to name the provider's authority `ui.email.android.com` to receive the query from the `AMS` (the `Yes` branch in Figure 1), return a MIME type of its own choice to misinform it, and in the meantime specify this type within its own activity Intent filter, making itself the only eligible app to get the Intent (for invoking the account settings activity). In our research, our attack app took over the content provider and responded to the query from `AMS` with a MIME type `vnd.android.cursor.dir/vnd.example.ABC`. Also, the attacker defines an Intent filter as illustrated in the next code snippet, by claiming a `mineType` with the type it told the `AMS`.

```
<!-- Malicious Setting Activity Intent Filters-->
<activity android:name=".MaliciousSetting">
  <intent-filter>
      <action android:name=
           "android.intent.action.EDIT"/>
      <category android:name=
           "android.intent.category.DEFAULT"/>
      <data android:scheme="content"
        android:host="ui.email.android.com"
        android:pathPrefix="/settings"
        android:mimeType=
            "vnd.android.cursor.dir/vnd.example.ABC"/>
  </intent-filter>
```

In this way, the Intent from the app went only to the malware, leading the user to a malicious activity that lets her enter her password. A demo of the attack is posted on the website [6]. We also successfully submitted the app to Google Play, before notifying Google of this security-critical flaw.

Tango in the dark. Tango is a popular cross-platform messaging app, offering audio, video calls over 3G, 4G and Wi-Fi networks. The app has been installed over 100 million times from Google Play. To display SMS messages received, it sets up an Intent filter with the action `android.provider.Telephony.SMS_RECEIVED` to get the Intent that carries the message from the Telephony Manager. When the user sends a message through Tango, the app saves it to `sms`, telephony's content provider.

On a device without Telephony, Tango's reference to its content provider becomes hanging. A malicious app, therefore, can define a content provider using the authority `sms` to get the SMS message the user sends. This can happen when the malware first sends a message, causing the inadvertent user to reply. What can be leveraged here is another vulnerability in Tango: the app does not protect its SMS receiver with the system permission `android.permission.broadcast_sms`, as it is supposed to do. This allows any party broadcasts to the action `SMS_RECEIVED` to inject a fake short message into the app. In our research, we implemented the attack on Tab S 8.4, sending a fake message to Tango and receiving the user's response using the malicious content provider. The demo of the attack is online [6].

LG CloudHub scam. LG CloudHub is a system app that allows managing cloud accounts, uploading data to clouds and accessing it from different devices. By default, the app supports Dropbox and Box, and on various devices can also connect the user to other services, including LG cloud provider. The information about these additional services is kept in a content provider `com.lge.lgaccount.provider`, which LG CloudHub looks up each time when it is invoked.

Interestingly, on some phones, this provider does not exist. A prominent example is LG G3. When this happens, LG CloudHub just displays the default services, Dropbox and Box. However, this makes the reference to the content provider a Hare case and exposes it to the manipulation of a malicious app. Specifically, we implemented an attack app that defined `com.lge.lgaccount.provider` and placed in the content provider an entry for LG Cloud account. This account was then displayed on the LG CloudHub available accounts list. Once it was clicked by the user, the app sent an implicit Intent with action `com.lge.lgaccount.action.ADD_ACCOUNT`. On the device (G3), no pre-installed apps define the action, which enabled the malware to define the action, claiming that it could handle the Intent. The consequence is that the user's click on the system app (LG CloudHub) triggered a malicious activity that masqueraded as the login page for LG Cloud account, which was used to cheat the user into exposing her password and other credentials. Here is the demo for the attack [6].

3.3 Permission Seizure

The Hare flaws can also be introduced by permissions, which are defined by system apps and utilized to control the access to various system (e.g., GPS, audio, etc.) or app-defined resources (e.g., content providers, broadcast receivers, etc.). During the OS customization process, the apps that specify the permissions (their original "owners") could be removed. In the meantime, if the resources guarded by these permissions are still there, the uses of the permissions (for protection) become hanging references. To exploit such flaws, the adversary can simply define those missing yet still being utilized permissions to gain access to the resources they protect. This problem was also found to be extensive in our research, present on all 97 factory images we scanned. Making this threat particularly perilous is the fact that Google Play does not check duplicate permissions: all our attack apps were successfully uploaded there. Here we describe two examples.

Getting contacts from S-Voice. The system app S-Voice includes a content provider (`com.vlingo.midas.contacts.contentprovider`) that maintains the information about the user's contacts, including names, email addresses, telephone number, home addresses, etc. Access to the provider is guarded by a pair of permissions `com.vlingo.midas.contacts.permission.READ` (`READ` for short) and `com.vlingo.midas.contacts.permission.WRITE` (`WRITE`). However, we found that they are not on defined on Galaxy Note 3 (phone) and Note 8.0 (tablet), which opens the door for the exploit.

Specifically, we built an attack app for both devices, which defined the `READ` and `WRITE` permissions. The app was found to be able to successfully read all the contact data from S-Voice and also update its data managed by the content provider at will, e.g., changing the email address, URLs and phone number of a contact, which could lead to information leaks and other consequences (e.g., causing the user to visit the adversary's URL placed in her friend's contact). We post a demo on our website [6].

Cracking Link. Link is a system app that allows its user to synchronize her data (files, images, audio, video, etc.) across different devices (phone, tablet, laptop, etc.). For this purpose, on a mobile device (phone or tablet), the app uses a content provider `com.mfluent.asp.datamodel.ASPMediaStoreProvider` to maintain the information about such data, together with the geolocations of the user. This provider is protected by `com.mfluent.asp.permission.DB_READ _WRITE` (`DB_READ_WRITE` for short). However, on many factory images, we did not find that the permission has been defined. As a result, the protection here becomes hanging.

We built an attack app in our research that defined the `DB_READ _WRITE` permission. On Galaxy Note 3 and Note 8.0, this app successfully acquired sensitive information from the content provider, including the user's geolocations, all the meta-data of documents, audio and video files (names, directory path, artist, genre, etc.). Also, the malware was able to change the meta-data.

4. DETECTION AND MEASUREMENT

To better understand Hares and mitigate the security risks they pose, we built a suite of tools in our research, including *Harehunter*, an automatic analyzer that detects Hare flaws from pre-installed apps on factory images, and *HareGuard*, an app that catches the attempts to exploit known hares on a device. Using Harehunter, we also performed a measurement study that inspected 97 factory OS images for popular devices like Galaxy S5, S6, Note 3, 4, 8.0, LG G3, Nexus 7, Moto X, etc. Our study brought to light 21557 likely Hares across these devices, which demonstrates the pervasiveness of such security-critical vulnerabilities. In the rest of the section, we elaborate the design and implementation of these new techniques and our findings.

4.1 Harehunter

As mentioned earlier, Harehunter is designed to identify hanging references within system apps and can achieve a high accuracy. We focus on these apps because prior research shows that pre-installed apps are the most intensively customized components across different Android devices [24], and therefore the most likely sources of Hare vulnerabilities. Our manual analysis further indicates that the major portion of Hares indeed come from system apps. On the other hand, framework services may also include hanging references, so do third-party apps (e.g., Tango). Harehunter can be directly applied to find the problems in the third-party apps and extended (by tweaking the pre-processing step) to work on Android services. Following we describe the idea, design of Harehunter and its implementation.

Design. The idea behind our design is simple. For each factory image, we first run a *differential analysis*: extracting all the attributes (package names, actions, activities, services, content providers and permissions) its pre-installed apps define and all the references to the attributes within their code and manifests, and then comparing the references with the definitions. Any discrepancy between these two ends indicates the possible presence of Hares. For example, if a package name is used to start an activity (`startActivity`) or bind a service (`bindService`) but it is not owned by any pre-installed apps on a device, the reference to it is likely to be hanging. On the other hand, such a reference could turn out to be well guarded: for example, before referring to the package, a system app may first check its existence, collect its signature information (e.g., `getPackageInfo` with `GET_SIGNATURE` flag) and verify it against the signature of the authentic app. To detect a truly vulnerable Hare, we have to analyze the code between a potential guard (e.g., functions for signature checking) and a possible hanging reference (e.g., `startActivity`) to find out whether they are indeed related. Only an unprotected reference will be reported as a Hare.

To implement this idea, we designed a system with three key components, *Pre-processor*, *Differ* and *Guard Catcher*, as illustrated in Figure 2: Pre-processor extracts app packages from an OS image and converts them into the forms that can be analyzed by follow-up steps; Differ performs the differential analysis and reports possible hanging references; Catcher inspects the APK involving such references to determine whether they have been guarded. In the rest of the section, we describe how these components were built in our research.

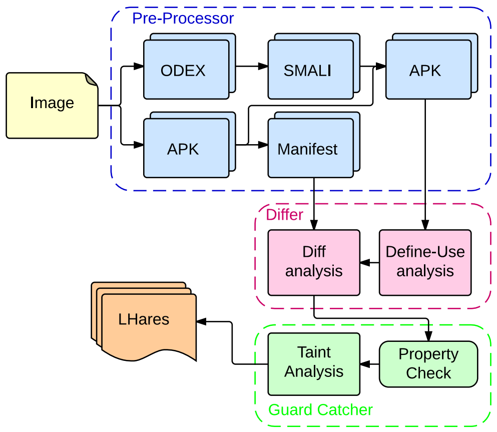


Figure 2: Design of Harehunter.

Pre-processing. From each factory image, Harehunter first collects all its pre-installed apps, in the forms of APK and ODEX files, and runs `Apktool` to extract each app's manifest file and `Baksmali` to decompile the app into Smali code. For some devices, particularly those with Samsung, a system app's ODEX file is often separated from its APK file, for the purpose of improving its loading time, while *Flowdroid*, the static analyzer we built our system upon, only works on APKs. To address this issue, our pre-processor was implemented to automatically unzip an ODEX file, decompile it and then recompile and compress it, together with its resource files, into a new APK file. Further complicating this process is that for Android 5.0 Lollipop, ODEX files are replaced with OAT files, which include native code. For the app in such a form, Harehunter first unzips its OAT files and then runs `oat2dex` to convert it to the ODEX formate, enabling the above process to move forward.

Differential analysis. To perform a differential analysis, Differ first searches all extracted, decompiled code and manifest files for the definitions of the targeted attributes. Running an XML parser, our approach can easily collect defined package, actions as well as content providers authorities and permissions from individual apps' manifest file. Note that all these attributes, except the action for receiving broadcast messages, can only be defined within the manifest. Although the action used in an Intent filter for a broadcast receiver can be specified programmatically, it only serves to get a message, not invoke a service or activity, and therefore its absence will not cause a Hare hazard.

Most references to these attributes are within the code, in the forms of various API calls. Specifically, package names and actions are utilized through `startActivity`, `startActivityForResult`, `startService`, etc. The authority name of a content provider appears in various operations on the provider, such as `update`, `query`, `delete` and others. Permissions are claimed in manifests or verified through `checkPermission` and other APIs. To identify these references, Differ first locates the call sites for all related functions from an app's Jimple code (an intermediate representation output by Soot [4]), and then performs a define-use analysis from each call site to recover the targeted attribute names, using the control-flow graph (CFG) constructed by Flowdroid. An issue here is that Flowdroid cannot create a complete CFG, missing quite a few program entry points like `onHandleIntent`. In our implementation, we added back as many entries as we could find, but were still left with some target function calls whose related CFGs could not be built by Flowdroid. For these calls, our current prototype can only deal with the situation where the attribute names are hardcoded within the related functions.

Guard detection. As mentioned earlier, references to missing attributes are often protected. There are two basic ways for such protection, signature guard or feature guard. Figures 3 and 4 present the examples for both cases. Signature guard tries to obtain the signature of the package to be invoked, and compare it with what is expected. In the example (Figure 3), this check is done through extracting the signature of `"com.facebook.katana"` through `getPackageInfo` with `GET_SIGNATURES` as a flag and then invoking `compareSignature` to compare it with that of the legitimate Facebook app, before binding to the target app's service (`bindService`). The presence of the authentic package can also ensure the correctness of action and activity names. The other way to protect these attributes is to check the build model of the current device, since only some of them come with certain features (in terms of packages, content providers and others): e.g., input methods, email apps can all be different from builds to builds; SMS/MMS providers may not even exist on a tablet. As an example, Figure 4 shows that an app first runs `hasSystemFeature` to check whether the current device supports Google TV (`com.google.android.tv`): if so, it invokes the app `youtube.googletv`, and otherwise, just YouTube.

To detect such protection, Guard Catcher conducts a *taint analysis* through both an app's data flows and control flows, using the functionalities provided by Flowdroid. Specifically, our approach first identifies a set of guard functions like `hasSystemFeature` and `getPackageInfo` with `GET_SIGNATURES` parameter and

```
public boolean extendAccessToken(Context context,
      ServiceListener servicelistener){
   Intent intent = new Intent();
   try{
        PackageInfo pi =
              context.getPackageManager().getPackageInfo
              ("com.facebook.katana",
              PackageManager.GET_SIGNATURES);
        // Compare signature to the legitimate Facebook
        // app Signature
        if (!compareSignatures
              (pi.signatures[0].toByteArray())){
           return false;
        } else{
           intent.setClassName("com.facebook.katana",
                 "com.facebook.katana.platform.
                 TokenRefreshService");
           return context.bindService(intent, new
                 TokenRefreshServiceConnection(context,
                 servicelistener), 1);}
   }catch(PackageManager.NameNotFoundException e){
        return false;
   }
}
```

Figure 3: Signature Based Guard Example

```
private void ViewVideo(Uri uri){
   Intent intent = new
         Intent("android.intent.action.VIEW", uri);
   if (getPackageManager().hasSystemFeature
         ("com.google.android.tv")){
      intent.setPackage("com.google.android.youtube.googletv");
   } else{
      intent.setPackage("com.google.android.youtube");
   } startActivity(intent);
}
```

Figure 4: Feature Based Guard Example

then attempts to establish relations between them and the hanging references discovered by the differential analysis, a necessary condition for these references to be protected. For this purpose, the outputs of these guards are set as taint sources and the references (e.g., `startActivity`, `bindService`) are labeled as taint sinks. Flowdroid is run to determine whether the taint can be propagated from the former to the latter. For the sinks that cannot be tainted, they are reported as likely Hares.

Running a full taint analysis (through both explicit and implicit information flows) for every guard and reference pair can be very slow. To make the guard detection more scalable, Catcher takes a multi-step hybrid strategy, combining quick property checks with the taint analysis. Specifically, it first inspects whether a source and its corresponding sink are within the same method. When this happens, in the vast majority of cases, they are related and therefore the reference is considered to be protected. Otherwise, our approach further compares the package name involved in a signature check with that used for a reference. A match found between the pair almost always indicates a protection relation. An example is `com.facebook.katana` within the code snippet in Figure 3 that shows up both within `getPackageInfo` and `setClassName`. Only when both checks fail, will the heavyweight taint analysis be used. In our large-scale analysis of factory images (Section 4.2), we found that most of the time, the guard for a reference can be discovered in the first two steps.

Evaluation. We evaluated the effectiveness of our implementation in a measurement study, which involves the OS images for 97 popular devices, all together over 24000 system apps. Harehunter reported 21557 likely Hares. From all these Hares, we randomly sampled 250 and manually analyzed their code. Only 37, i.e., 14%, were found to be false detection: that is, falsely treating a guarded reference as a Hare. We further measured the false negative rate of the Guard Catcher by randomly checking likely hanging references reported by Differ and comparing the findings with what was detected by Catcher. In all 250 samples, 46 (19%) were missed by our implementation: i.e., true Hares falsely considered to be guarded. Looking into those false positives and negatives, we found that they were all caused by the incomplete call graphs output by FlowDroid. Flowdroid is known to have trouble in dealing with ICC [18] and other issues like missing entry points and incomplete call graphs. When this happens, a taint analysis cannot go through.

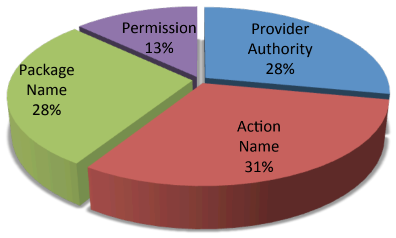


Figure 5: Distribution of Hares across Different Hare Categories

4.2 A Large-scale Measurement Study

To understand the scope and magnitude of the security hazards caused by Hares, we performed a large-scale measurement study on 97 factory images. The study shows that Hares are indeed pervasive, with a significant impact on the Android ecosystem: over 21557 LHares were discovered and many of them could lead to the consequences such as activity hijacking, data leakage and pollution. Following we report our findings.

OS Image collection. In our research, we collected 97 factory images from Samsung Update [3], Android Revolution [2] and physical devices, which include around 183 apps per image and 24185 all together apps. These images are customized for 49 different phone or tablet models, 36 countries and 23 different carriers. They operate Android versions from 4.0.3 to 5.0.2. The detailed information is presented in Table 1. Please note that we are anonymizing vendors upon their request.

Landscape. When analyzing those factory images, we found that about 13% of their pre-installed apps could not be decompiled by `Apktool` or analyzed by `Flowdroid`. Among those that could be analyzed, Harehunter discovered all together 21557 flaws (unguarded hanging references) within 3450 vulnerable apps. Note that some of these flaws might occur more than once within the same app, and some of the vulnerable apps show up on multiple devices. Our research reveals that every single image contains a large number of Hare flaws, ranging from 8 to 598. On average, 14.3% of pre-installed apps on 4.X and 11.7% on 5.X were found to be vulnerable. Table 2 shows the details.

Also as we can see from the table, the problems are also pervasive across different device manufacturers: both Vendor A and C have a significant portion of their system apps involving hanging references. By comparison, Vendor D has the smallest number of flaws (29) and the lowest ratios (8%) of faulty apps. A possible reason is that the OS images its devices run are the least customized ones, which minimizes the chance for introducing Hares.

Figure 5 illustrates the distribution of the flaws across different Hare categories. Most problems come from undefined action

Table 1: Android Images Collected

Vendor	# of Images	# of System Apps	Avg # of System Apps per Image	# of Countries	# of Carriers	# of OS Versions
Vendor A	83	21733	261	36	23	10
Vendor B	7	1561	223	1	1	4
Vendor C	1	174	174	1	1	1
Vendor D	4	398	99	1	1	3
Vendor E	2	319	159	2	1	2
Total	97	24185	183	36	23	10

Table 2: Hares Prevalence in System Apps per Vendor

Vendor	Hares in Android 4.X		Hares in Android 5.X		Avg Hares per Device	Min Hares per Device	Max Hares per Device
	# of Hares	# of vulnerable apps	# of Hares	# of vulnerable apps			
Vendor A	19279	3045 (18%)	608	99 (6%)	239	23	598
Vendor B	679	121 (13.3%)	425	85 (15.5%)	157	100	224
Vendor C	N/A	N/A	248	33 (21.5%)	241	248	248
Vendor D	107	31 (12.4%)	8	5 (5%)	29	8	45
Vendor E	187	23 (15.6%)	16	8 (12.1%)	101	16	187
Total	20252	3220 (14.3%)	1305	230 (11.7%)	153	8	598

names. By comparison, a relatively low percentage of permissions were found to be involved in hanging references.

Impacts. The impacts of Hares are significant. In addition to the end-to-end attacks we built (Section 3), we also randomly sampled 33 flaws and manually analyzed what could happen once they were exploited. Note that due to the lack of a large number of physical devices, all we could do is just static analysis to infer possible consequences once an exploit succeeds. Such an analysis may not be accurate, but it is still important for understanding the impacts of this type of security flaws that have never been noticed before. The outcomes of our analysis are shown in Table 3.

As we can see here, 5 instances of the randomly picked Hares might be exploited to launch similar Phishing attacks as discussed in Section 3, due to undefined package and activity names and/or action names for activity Intent filters. One Hare found in the HTC Task App allows redirecting an Intent through exploiting a non-defined content provider used for Intent resolution, just like the GoogleEmail attack. 4 Hares (on the devices such as Note 8.0 and S5) might cause content leakage (notes and browser bookmarks) once malware impersonates undefined content providers, which the victim apps insert data into. 4 instances might expose user's private information when hanging package names are hijacked. Particularly, we found that on Note 8.0, a hanging reference involves an explicit Intent delivered to a nonexisting package. The Intent includes a content URI pointing to private data (e.g., photos) and also a permission `FLAG_GRANT_URI_PERMIS SION` that enables the recipient to read the data without requesting a permission. As a result, an unauthorized app using the target's package name could gain access to the data.

Also, on LG G3, a hanging reference to a nonexisting content provider might open the door for the adversary to define those providers to contaminate the data synchronized to the user's other devices. Further, our analysis reveals 3 instances that might cause denial-of-service attacks when the adversary creates undefined content providers that victim apps use, and sets their exported flag to false. From the app code, this attack could cause a security exception when the victim app attempts to read or write to these providers. A prominent example is Amazon MP3 app (pre-installed on specific HTC models such as One M8). Once launched, it checks an undefined provider. If a malicious app declares this provider and sets its exported flag to false, Amazon MP3 will never be able to run until the malicious app is uninstalled. Some other Hares may lead to unexpected situations: e.g., an app with a certain package name will not show up in system Task Managers and other apps on LG G3 could not be forced to stop from the LG Settings app.

We also found that Hares in 3 apps might only cause display of dialogs or notifications. Also, there are 6 hares related to missing services whose functionalities we could not figure out. Finally, we did not find any entry points for 4 Hares, which could be dead code.

Table 3: Possible Impact of 33 Randomly Picked Hares

Impact	Hare Category	# of Hares
Activity Hijacking	Package and Activity Name	3
Activity Hijacking	Action Name	2
Activity Hijacking	Provider Authority	1
Data Leakage	Provider Authority	4
Data Leakage	Package and Activity Name	1
Data Pollution	Provider Authority	1
D.O.S.	Provider Authority	3
Dialog Popup	Action Name of Activities	3
Others	Package Name	5
Impact Not Clear	Action Name of Services	6
Maybe Dead Code	All Categories	4

Responsible parties. We further looked into which parties introduce such flaws and when this happens. For this purpose, we inspected 6 images from Vendor A all running 4.4.2, as described in Table 4. The percentage of Hare flaws that are uniquely introduced by these models ranges from 9% to 29%. We further grouped the images into subgroups (e.g., phone, tablet) and checked which ones exhibit the highest percentage of common Hare cases. Tablet models have the highest percentage of common Hares 63%, while phone models have the second highest common Hares 56%. The common Hare cases between a tablet and phone device model is at most 38%. So customizing the OS to tablet models or to phone models introduces a lot of Hares. In the meantime, we also compared the flaws found on the same model (Phone 3 running Android 4.4.2)customized for different carriers. The results are in Table 5.

Table 4: Hare Flaws in Different Vendor A Models Running Android 4.4.2

Model	# of New Hares Introduced by Model
Tablet 1	106 (27%)
Phone 2	35 (21%)
Phone 3	75 (29%)
Tablet 4	57 (22%)
Tablet 5	22 (9%)
Tablet 6	72 (20%)

As we can see from Table 5 given a Phone 3 image, its customizations across 6 carriers bring in about 3% to 20% of flaws. Clearly, both manufacturers and carriers cause Hare flaws. However, the former apparently needs to take more responsibility than the latter. Also, most Hares are likely to be introduced during the OS customizations for different device models (phone or tablet).

Trend. Figure 6 further compares the ratios of vulnerable apps over different OS versions across multiple manufacturers. For Vendor A

Table 5: Hares in Phone 3 Running Android 4.4.2 For Different Countries and Carriers

Country	Carrier	# of Hares Introduced by Carrier
China	China Unicom	51 (20%)
U.S.	AT&T	22 (13%)
Chile	Entel pcs	4 (3%)
Argentina	Movistar	5 (3%)
Brazil	Vivo	5 (3%)
S. Korea	SK Telecom	44 (18%)

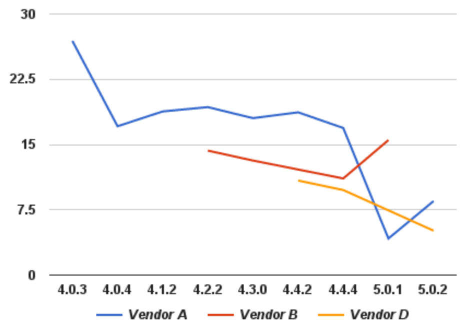


Figure 6: Ratios of Vulnerable Apps Across Different OS Versions and Manufacturers

devices, there is an observable trend that the higher versions (5.0.1 and 5.0.2) contain fewer Hares than the lower ones: the faulty ratio comes from 26% on 4.0.3 down to about 8.2% on 5.0.2. On the other hand, for Vendor B phones, the trend is almost constant: the ratio is 14.3% on 4.2.2 and 15.1% on 5.0.1 . Also, on all these devices, the Hare risks remain significant, which indicates that manufacturers have not yet realized the gravity of this type of vulnerabilities.

4.3 App-level Protection

Motivation and idea. Fundamentally, the Hare flaws can only be fixed by device manufacturers and app developers, who are supposed to either remove the hanging references in their code or put proper security checks in place. However, given the pervasiveness of the problem and its root cause, i.e., the under-regulated Android ecosystem, we believe that they cannot be completely eliminated within a short period of time. Before their complete solution can be implemented (Section 5), it is important to help individual Android users protect their systems, in the presence of these flaws. Compared with a frame-work layer protection, which can only be deployed by manufacturers and carriers, the most practical solution is app-level defense, as all the users need to do is just to install a protecting app from Google Play to get immediate protection against the threats to the vulnerabilities on her system. We found that this can actually be easily done.

In our research, we developed such simple protection, using an app, called *HareGuard*, to scan other third-party apps whenever they are installed to ensure that they are not taking advantage of any known Hare vulnerabilities on a specific device model. HareGuard collects a device's model information and queries a server-side database to acquire all the Hares within the model (which are detected off-line, for example, through Harehunter). Whenever an app is installed, HareGuard immediately checks its manifest file for the package name, activity, action, authority name and permissions it defines, making sure that the app does not intend to hijack any missing attributes. This scanner app is invoked through `startForeground`, running with a notification posted on the Notification Center.

Implementation. Specifically, as soon as HareGuard is installed, it calls `Build` class to collect the device information, including `Build.MANUFACTURER` and `Build.MODEL`, and queries our database for all the Hare flaws on the device. The scanner also utilizes an Intent receiver with actions `android.intent.action.PACKAGE_ADDED` to monitor new app installed and `android.intent.action.PACKAGE_CHANGED` to detect whether an app is updated. For each new or recently updated app, it uses the API `openXmlResourceParser` to open its manifest file and identify all the attributes it defines. These attributes are then compared with a set of hanging references retrieved from our Hare database to detect Hare risks: i.e., defining an attribute associated with a hanging reference. Once a risk is found, HareGuard alarms the user, explaining potential security hazards to her and urging her to make sure that the app indeed comes from a reliable source or simply remove it. To assist the user in this process, the scanner can compare the signature of the app with the one belonging to the authorized party, whenever it exists in the database.

We implemented HareGuard in our research, using a database that documents the findings made by Harehunter when scanning the factory images for popular mobile devices.

Evaluation. Our implementation of HareGuard was found to be effective at detecting all the attack apps we built. We further evaluated its performance, whose impacts on its host system are minimum: the scanner was found to utilize only 4.29 MB memory and consume 0.29% CPU when scanning an app's manifest. To protect the Android users from the serious security risks brought in by Hares, we plan to release the app in the near future, as soon as the code has been thoroughly evaluated.

5. DISCUSSION

Hares are not just a few isolated, random bugs introduced by implementation lapses. The presence of such flaws implies the weaknesses in Android's design philosophy and its ecosystem. Fundamentally, Android is a complex system, whose components and apps are meant to work together, which leads to highly complicated *interdependent* relations among them. In the meantime, the Android ecosystem is known to be highly diverse and de-centralized: each OS version is customized and re-customized by various parties almost independently and utilized by anyone who can build an app for the version; so far little guidance has been provided to help regulate the customizations and app development, making sure that they respect the existing complicated relations among system components and apps introduced by themselves and other parties (AOSP, manufacturers, carriers, app developers, etc).

In the absence of such guidance and a proper enforcement mechanism, hanging references become inevitable. As evidenced by our research (*the first one on this new category of problems*), indeed Hares are pervasive, existing on every single device we inspected, and also indeed they are security-critical, endangering sensitive user data (e.g., voice memo) and even the proper execution of system apps (e.g., activity injection in Google Email). Even though not every problem reported by Harehunter is exploitable, which depends on the conditions for running vulnerable code, the pervasiveness of such unprotected code is alarming: without deep inspection of individual cases, no one knows whether they can be exploited under certain conditions, leading to unexpected consequences.

Moving forward, we believe that systematic effort needs to be made to eliminate these flaws, and also lessons need to be learnt to avoid the similar pitfall when building other open computing systems. Following are a few thoughts.

Elimination of Hares. To completely eliminate the Hare risks, it is important to have such interdependent relations well documented and make them open to the parties involved in OS customizations and app development. Also, there should be a policy in place that requires that anyone who modifies the OS or builds an app should not create a hanging relation such as referring to a nonexisting attribute, and a mechanism for the policy compliance check. The policy enforcement here can leverage the existing Android compatibility program, which currently still cannot do security check. The challenging part is the collection of the interdependent relations for all known Android versions. Such information is not there yet. Actually, our study shows that the manufacturer seems unaware of the relations on its own device, often breaking them and causing Hares when customizing an Android version to different models. A systematic tool, like Harehunter, is needed to identify such information.

In the meantime, effort should be made to secure each attribute reference. Most importantly here is explicit authentication before a reference. All too often we have seen that references are only protected *implicitly*: e.g., the reference to a system app is secured by the presence of the app on a device, which excludes any other app using the same package name. Such protection is fragile, completely falling apart once the app is removed when the OS is customized for a new device model.

On the other hand, a security check can be more complicated than it appears to be. More specifically, even though references to package names can be directly guarded with a signature check. Other attributes like content providers, actions, etc. can be directly used and their presence on a specific device is often verified by checking the current device model and other features. The correctness of such a check, again, hinges on the knowledge about the components/apps relations across different versions, models, etc., which need to be recovered by Harehunter and other similar tools.

Protection of legacy systems. Before we can even think about how to eliminate Hares in developing future systems and apps, an issue we first need to address is how to secure existing devices, which, as shown in our research, are riddled with different kinds of Hare flaws. The techniques we developed, Harehunter and HareGuard, made a first step toward identification and protection of these vulnerabilities. Particularly, as mentioned earlier, Harehunter can also play a critical role in gathering the interdependent relations to help secure new systems and apps. With its great potentials, our current implementation is still preliminary: it introduced about 14% of false positives and missed 19% of truly vulnerable cases in our study (Section 4.1). Most problems are caused by Flowdroid, the static analysis tool that supports our system. It is conceivable that Harehunter will become more effective once a more capable analyzer is used. Also, for device manufacturers who have the source code for all the services and system apps, a tool similar to Harehunter, but working on source code, could be more accurate in detecting the Hare flaws. We expect that these directions will be explored by both the academia and the industry in the near future.

6. RELATED WORK

In this section, we review related prior research and compare our work with those studies.

Security risks in Android customizations. The security risks introduced by the fragmented Android ecosystem has been studied before. Prior research analyzes the pre-installed apps on 10 factory images and reports the presence of a few known problems such as over-privilege, permission re-delegation, etc [13] and [12]. Unlike this prior study, here we focus on a type of vulnerabilities never reported before, hanging attribute references, which is also specific to the customization process. Our research demonstrates the serious consequences of the new flaws and identifies their fundamental causes. This has never been done before.

Another related study is the security configurations of Android's Linux device drivers [26]. The research finds that many of these devices have not been properly protected, causing their exposures to the parties that should not access them: e.g., an app can directly command an exposed camera driver to take picture, even when it does not have the camera permission. This research also involves a measurement study that reports the pervasiveness of the problem across over 2,000 factory images. By comparison, our study on Hares happens on pre-installed apps, which requires more complicated code analysis than a simple check of Linux drivers' security configurations, as does the prior work [26]. Also importantly, pre-installed apps are known to be the main target of a customization [24] and their customization-specific flaws have never been investigated before, up to our knowledge.

Activity and Service Hijacking on Android. Earlier research work [10] has studied unauthorized intent receipt where an attacker can hijack activities and services in case of implicit intents. The work does not directly touch the Hare flaws as it does not require the absence of the legitimate activity/service being referred. Rather, it discusses the cases where multiple recipients are present on the device. In our work, we discuss that even an explicit intent can be hijacked when the legitimate recipient is not in place. We further evaluate the security consequences of hijacking other components such as content providers and permissions.

Vulnerability detection on Android. Security vulnerabilities on Android have been extensively studied. Prominent examples include the re-delegation problem [13], content provider leaks [15], security issues in push-cloud messaging [19] and others [16]. However, up to our knowledge, never before has anyone investigated the security risks of hanging references: i.e., the parties to be invoked do not exist and can therefore be impersonated by a malicious app.

Most related to our work is the recent study on security risks in Android upgrades [25]. What has been found is that a malicious app installed on a lower version Android can claim the capabilities (e.g., permissions, shared UID, etc.) only show up on a higher version, and then automatically acquire such capabilities during an OS upgrade. This problem (called Pileup [25]) is caused by the logic flaws within the Android upgrade mechanism, which tends to avoid substituting a new app for the existing one with the same attributes such as package names. In a Pileup exploit, new attributes not in use are preempted by a malicious app before an upgrade, while in a Hare attack, the adversary takes advantage of the attributes that do not exist but are still used on a device.

Static analysis on Android. The Hare flaws can be detected by statically analyzing Android apps. Techniques serving this purpose has been extensively studied [11, 14, 27, 23]. Particularly, FlowDroid [7] has been widely used for taint analysis on Android apps. With its popularity, the tool suffers from a few limitations. For example, it does not handle ICC (inter-component communication) well, which later has been improved by systems such as Epicc [21], Didfail [17] and IccTA [18]. This problem could make guard detection more difficult: e.g., when a reference to a package has been separated by an ICC call from the program location where its signature is verified. In practice, however, this is found to rarely happen: a vast majority of security checks occur right before the reference. Therefore, this weakness has never become a big issue in our study. What does cause a problem is Flowdroid's limited capability to identify all the entry points within an app, missing a lot of callback functions such as `onHandleIntent`. As a result, we may not be able to precisely analyze some references that can only be found through such missing entries. Again, the problem can be addressed by a more capable static analysis tool, which will further enhance the accuracy of our approach.

Dangling pointer protection. Remotely related to our work is the prior research on dangling pointers, a memory vulnerability in which a pointer in a program does not point to a valid object [9]. The problem has been studied for decades and can be detected by various tools such as Valgrind [20]. Given the conceptual similarity between this old problem and Hare, the new security risk actually comes from the interconnections among different apps and system components, whose detection and mitigation need to be done across the whole operating system. This poses a new challenge to the system security research.

7. CONCLUSION

In this paper, we report our research on a serious Android security flaws that have never been studied before. The problem, called Hare, has been caused by the conflict in the decentralized, unregulated Android customization process and the complicated interdependencies among different Android apps and components. Once an Android version is modified without fully considering such dependency relations, references to some attributes can easily become hanging, which can be taken advantage by the adversary to steal sensitive user information or compromise the integrity of her data.

Our research brings to light the significance of this security risks, revealing the damages that can be done on various Android devices, such as stealing voice memos, controlling the screen unlock process, replacing Google's Email account settings activity, etc. We further built the first tool for automatically detecting Hares and utilized it to analyze the factory images for 97 popular Android devices. This study shows that almost every single device contains a large number of Hare flaws and manufacturers have yet realized the significance of this security hazard. To provide Android users immediate protection, we developed an app for automatically identifying the attempts to exploit the vulnerabilities on various devices. Also our study highlights the importance for regulating the fragmented Android ecosystem, and offering guidance and enforcement mechanisms to improve the security assurance for the system customization and app development.

8. ACKNOWLEDGEMENTS

We thank anonymous reviewers for their comments. This project was supported in part by the NSF grants CNS-1117106, 1223477, 1223495, 1318814 and 1527141. Kai Chen was supported in part by NSFC 61100226. Yousra Aafer thanks Samsung Research America for supporting this project during her internship at Samsung.

9. REFERENCES

[1] Android compatibility. `http://source.android.com/compatibility/`.

[2] Android revolution mobile device technologies. `http://android-revolution-hd.blogspot.com/p/android-revolution-hd-mirror-site-var.html`. Last Accessed: May 13, 2015.

[3] Samsung updates: Latest news and firmware for your samsung devices! `http://samsung-updates.com/`. Accessed: 05/02/2013.

[4] Soot: A framework for analyzing and transforming java and android applications. `http://sable.github.io/soot/`. Last Accessed: May 13, 2015.

[5] Dashboards. `https://developer.android.com/about/dashboards/index.html`, 2015. Accessed May 13, 2015.

[6] Hare hunting. `https://sites.google.com/site/androidharehunting/`, May 2015.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, New York, NY, USA, 2014.

[8] P. Brodley and leviathan Security Group. Zero Permission Android Applications. `https://www.leviathansecurity.com/blog/zero-permission-android-applications/`. Accessed: 10/02/2013.

[9] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012. ACM, 2012.

[10] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys '11, New York, NY, USA, 2011. ACM.

[11] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013.

[12] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In Proceedings of the 18th ACM conference on Computer and communications security, CCS '11, New York, NY, USA, 2011. ACM.

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

[14] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. Springer, 2012.

[15] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In Proceedings of the 19th Network and Distributed System Security Symposium (NDSS), Feb. 2012.

[16] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, New York, NY, USA. ACM.

[17] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis. ACM, 2014.

[18] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. arXiv preprint arXiv:1404.7431, 2014.

[19] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, New York, NY, USA, 2014. ACM.

[20] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In PLDI, 2007.

[21] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In Proceedings of the 22Nd USENIX Conference on Security, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[22] P. Ratazzi, Y. Aafer, A. Ahlawat, H. Hao, Y. Wang, and W. Du. A systematic security evaluation of Android's multi-user framework. In Mobile Security Technologies (MoST) 2014, MoST'14, San Jose, CA, USA, May 17 2014.

[23] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, New York, NY, USA, 2014. ACM.

[24] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In Proceedings of the 2013 ACM SIGSAC conference on Computer communications security, CCS '13, pages 623–634, New York, NY, USA, 2013. ACM.

[25] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP '14, pages 393–408, Washington, DC, USA, 2014. IEEE Computer Society.

[26] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA.

[27] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In Proceedings of the 19th Annual Network & Distributed System Security Symposium, Feb. 2012.