

# Your phone is leaking data!

## Evaluating Android content provider permissions

Prajit Kumar Das  
University of Maryland,  
Baltimore County  
prajit1@umbc.edu

Anupam Joshi  
University of Maryland,  
Baltimore County  
joshi@umbc.edu

Sandeep Narayanan  
University of Maryland,  
Baltimore County  
sand7@umbc.edu

Nilanjan Banerjee  
University of Maryland,  
Baltimore County  
nilanb@umbc.edu

Stanislav Bobovych  
University of Maryland,  
Baltimore County  
sb9@umbc.edu

Ryan Robucci  
University of Maryland,  
Baltimore County  
robucci@umbc.edu

### ABSTRACT

The number of mobile devices in the world surpassed the number of personal computers in 2010. Mobile devices now carry sensitive personal data, captured through sensors on the phone, as well as confidential corporate data through work emails and apps. As a result, they have become lucrative targets for attackers and the privacy and security of these devices have become a vital issue. Existing access control mechanisms on these devices, which mostly rely on a one-time permission grant, are too restrictive and inadequate. Such mechanisms are incapable of controlling contextual or custom app-data flows. In this paper we focus on this scenario and show how data leakages may occur due to developer inadequacy and a lack of proper checks for such leakages. We describe a potential loophole in the Android permission verification mechanism and a way to capture such a vulnerability on a user's mobile device. We also show a mechanism of injecting such a vulnerability into any app.

### Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection—Access Controls

### Keywords

Access Control, Android Content Providers, Permission Control

## 1. INTRODUCTION

Mobile devices have become ubiquitous due to their power, convenience and low cost. Android has become the biggest player in the market as Google boasts of more than a billion 30-day active user [14]. According to the International Data Corporation's Worldwide Quarterly Mobile Phone Tracker

report, Android has a 85% market share in the smartphone category. Apps from the Google Play Store and a variety of other outlets like Amazon App Store and Samsung Galaxy Apps provide a plethora of ways through which Android users can get their apps [2]. According to Statista [13], as of July 2015, there are more than 1.6 million Android apps in the Google Play Store.

The proliferation of smartphones has led to the popularity of the BYOD (Bring-Your-Own-Device) paradigm, whereby people use their personal devices in their workplaces to access business information and services. Naturally, this creates a greater need to ensure strong access control mechanisms for data on such devices. In certain domains the access control needs are critical. For example, for Medical and Health & Fitness apps, it is essential to maintain the highest level of security for patient data. Hospitals today use various hardware devices that are smart enough to communicate with smartphones and may even contain sensitive medical data. In addition, Android apps are capable of collecting huge amounts of data about smartphone users, often without their knowledge.

In this paper, we introduce Heimdall<sup>1</sup>, a heuristics based system that is currently in-the-works in our group. Heimdall is capable of detecting common vulnerabilities on an Android device that can cause leakage of app data. Heimdall has been created with a BYOD scenario in mind, where part of the system resides on the mobile device and part on the server. The server-side includes a dashboard that gets notifications of apps being installed on mobile devices used by employees of the organization. The system is then able to analyze and detect if the app is vulnerable with respect to a list of previously known heuristics. We are adding new heuristics as we discover and study them.

In the current paper, we have focused on a vulnerability in custom permissions created by app developers. The purposes of these permissions usually is to protect the data exposed through an app's content providers. It is advised by Google that, if an app developer creates a content provider for allowing access to their own data, they should also create a permission to control access to it. However, this requirement is not a stringent one and a developer might simply

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HotMobile'16 February 23-24, 2016, St. Augustine, Florida, USA  
Copyright 2016 ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

<sup>1</sup>Heimdall is the all-seeing and all-hearing guardian sentry of Asgard who stands on the rainbow bridge Bifröst to watch for any attacks on Asgard

ignore creating such a permission. We show in this paper, how such a vulnerability might lead to leakage of app data. We use two different mechanisms to demonstrate the issue. We show that it is possible to exploit this vulnerability using our own data access app and content provider app pair. We also show that it's possible to reverse engineer and repackage any standard app to create this vulnerability. We did observe that it is possible to handle this issue in code instead of leave the app vulnerable or delegating this issue to Android. However, through our evaluation we show that such a check might be beyond standard practices.

Previous work points out the extensive research that have gone into various mechanisms to study vulnerabilities in Android apps. The mechanisms have ranged from app meta-data analysis by Pandita et. al. [12], to detecting malware by studying their characteristics carried out by Zhou et.al. [15]. Such studies indicate a need for better mobile anti-malware solutions and access control mechanisms. We can understand from the extensive work done that there is significant knowledge about vulnerabilities on Android and ways to detect them. In this paper we present Heimdall, a system which can detect such vulnerabilities, and show an example of how one of these vulnerabilities can be detected using our system.

The rest of the paper is organized as follows. We describe our system overview in the section 2. That is followed by a description of the problem at hand in section 3. We present how a such a loophole can be introduced in any Android app in this section. We present a working prototype that is capable of detecting such a vulnerability in section 4. We conclude the paper with a discussion of related work in section 5 and future research directions that can lead to more vulnerability discoveries in section 6.

## 2. SYSTEM OVERVIEW

Heimdall has two components. The first is an app installed on a user's mobile device and the second is a web server that receives app install, uninstall and update notifications when these events occur on the device. Upon notification, the server processes all heuristics that apply to the app and generates a set of actions for a system administrator to look at. The list of recommended actions can then be sent over to the Heimdall Mobile App. The server also allows a system administrator to add more heuristics and add action notifications to be sent to a mobile device in a BYOD scenario. Heimdall server has two capabilities. It can generate reverse engineered apps that we can test on mobile devices. The reverse engineering process takes into account the heuristics that allow us to detect vulnerability in apps. It can then introduce these vulnerabilities into the repackaged apps. For example we discuss a vulnerability in the next section where content providers on Android could have a potential breach of data. We introduce this vulnerability into any provider associated with apps that we are reverse engineering. We then remove any associated permission and ensure that the "exported" tag for the provider is set to true. The primary task of Heimdall is to detect the vulnerabilities in the apps, like the one we just described.

For demonstrating these capabilities, we downloaded about 1500 apps from the Google Play Store and used a tool called apktool<sup>2</sup> to decompile the Android binary application pack-

<sup>2</sup> A tool for reverse engineering 3rd party, closed, binary An-

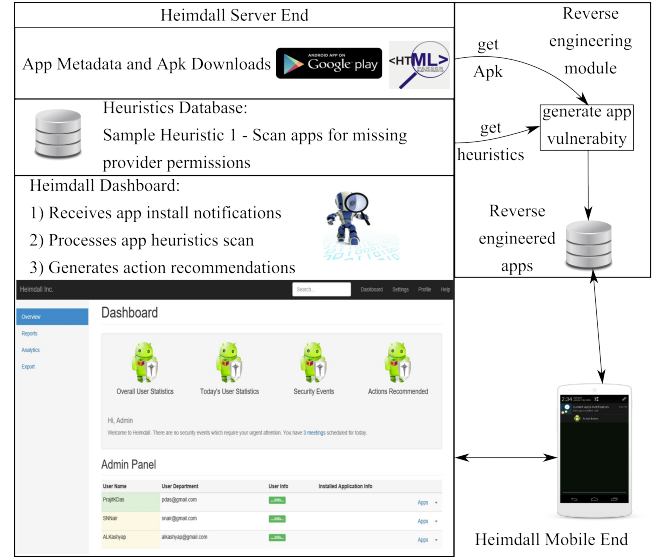


Figure 1: System Overview

ages (apks). We then parse the manifest files to find providers and thus determine whether the apps are vulnerable or not. At the same time if they are not already vulnerable we can introduce the vulnerability and repackage the app for testing purposes. We are working on including more heuristics into Heimdall to make it capable of detecting many more vulnerabilities.

## 3. VULNERABILITY DESCRIPTION

In Android, a content provider is a data repository for an application, which offers a consistent and standard interface for secure data access. It is mainly used for sharing data across applications and hence need to be protected with adequate permissions to prevent inadvertent data leakages. Unfortunately, Google's Android documentation has conflicting information in its documentation of access control mechanism for content providers. The first document states that data from a provider that specifies no permissions should not be accessible from other apps, while the second scenario states a conflicting view as quoted below.

- "[...] If a provider's application doesn't specify any permissions, then other applications have no access to the provider's data. However, components in the provider's application always have full read and write access, regardless of the specified permissions." <sup>3</sup>
- "[...] All applications can read from or write to your provider, even if the underlying data is private, because by default your provider does not have permissions set. To change this, set permissions for your provider in your manifest file, using attributes or child elements of the <provider> element. You can set permissions that apply to the entire provider, or to certain tables, or even to certain records, or all three." <sup>4</sup>

droid apps <https://ibotpeaches.github.io/Apktool/>

<sup>3</sup>Error in specification <https://developer.android.com/guide/topics/providers/content-provider-basics>.

Content Provider app	Content accessing app	Remark
No permission associated with provider	No permission used	<b>Potential data leakage</b>
Permission associated with provider	No permission used	Permission denied
Permission associated with provider	Permission used	Ideal scenario
No permission associated with provider	Permission used	<b>Potential data leakage</b>

**Table 1: Scenario when data leakage may happen**

We found the first document to be untrue. Table 3 lists the various scenarios and points out when a content provider app is not associated with a permission we may have data leakage. This happens because Android does not verify that each provider has an associated permission. There is one more condition required for this vulnerability to open up the provider to potential attacks and that is the exported setting to be set as true in the Manifest file for the provider app, as shown in code Listing 1. Possible solutions to mitigate this issue would require, either a change in how Android handles content provider access control or a change in the app developer’s code.

**Listing 1: Provider exported tag set as true**

```

...
<provider
    Android:name="contentProviderName"
    Android:authorities="authorityName"
    Android:exported="true">
...

```

Such a vulnerability can also be created deliberately. As shown in the work by Zhou et. al. [15] app repackaging is one of the most common techniques for Android malware creation, we show that it is possible through a simple change in code to introduce such a vulnerability in any app. There are some obvious ways to check for such manipulations and top developers in the Google Play Store usually do include such checks. However, these checks are not part of the Android framework or operating system and therefore a repackaged app can be used to fool users into installing a rogue application and allow their data to be stolen.

## 4. EVALUATION

We have discussed a potential loophole in Android’s custom provider data flow, in this paper. We are going to demonstrate four possible scenarios for this loophole through our experiments. In each case, the vulnerability either already exists in the app or it was introduced by us. In scenario 1 we have an app that has the vulnerability and does nothing

html#Permissions  
<sup>4</sup>Correct specification <http://developer.Android.com/guide/topics/providers/content-provider-creating>.  
html#Permissions

to protect itself and we know the exact URI call to access the content provider. Scenario 2 is where the app uses certificate key signatures to detect the reverse engineering and blocks any attempt to start the app itself. Scenario 3 is where the app does not crash at all and works like a normal app. However, when one tries to access a component of the app like a content provider, the app includes custom access control checks. Scenario 4 is still under investigation, but this is the case where an app’s URI string can be fully obtained by a combination of parsing the Manifest file and guess work. In order to demonstrate this problem we built a proof-of-concept(PoC). All our experiments were ran on a LG Nexus 5 device with Android Marshmallow 6.0 installed on it.

### 4.1 Scenario 1: Vulnerability with complete knowledge

In our PoC, we have an app(COMMAND) that has an exported content provider. We created another app(Parser) that is capable of accessing the content provider. We use two different application package sets and observe the results. The first set contains the COMMAND app without any permission specification and Parser app without any permission request. The second set contains the COMMAND app with permission specification and association with the content provider that was created. It also includes the Parser app with a request for the permission that was created by COMMAND.

#### 4.1.1 PoC case 1 for permission control

COMMAND has associated permission, Parser has requested said permission. We see in Figure 2 that, in this case there are no errors and we are able to make a sample query to the content provider.

#### 4.1.2 PoC case 2 for permission control

COMMAND has no associated permission, Parser has not requested said permission. We see in Figure 3 that, in this case there are no errors and we are able to make a sample query to the content provider. We propose that there should be a check in such a case to ensure that data access is to be allowed or denied. At present this does not happen and app developers resort to individual techniques to protect their data.

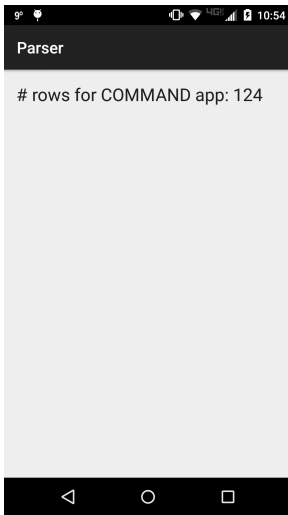
#### 4.1.3 PoC case 3 for permission control

COMMAND has associated permission, Parser has not requested said permission. We see in Figure 4 that, causes a permission denial error which is what we expected.

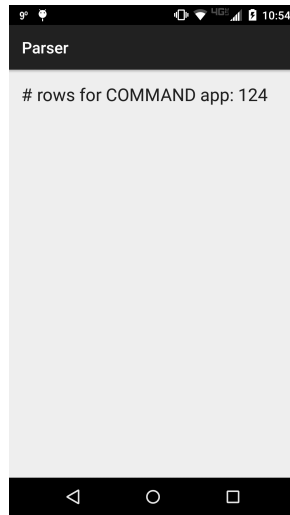
#### 4.1.4 PoC case 4 for permission control

COMMAND has no associated permission in the system, Parser has requested an unknown permission. In this case there is a potential of data leakage happening on the phone as the custom content provider is not protected.

The PoC proves that there is no difference from user perspective between an app which has a content provider with proper protection and an app which does not have such access control implemented. This is because in both cases the user does not see any error on the phone and safely assumes that their data is safe. This means user data can potentially leak without user’s knowledge. We ran our analysis on a set of 1500 randomly selected applications with a mix of popu-



**Figure 2: Android content provider accessed with permission**



**Figure 3: Android content provider accessed without permission**

lar applications like Facebook, GMail, Instagram as well as less popular and unknown apps like Expense Manager, Call App etc. Our system found 150 applications with content provider marked as exported="true" and no associated permission for the provider. Therefore about 10% of apps have this potential loophole but we wanted to find out if we could change an app to leak it's data.

This led to our second set of experiments trying to determine, if the apps had incorporated additional protection apart from the standard Android permission mechanisms. For these experiments we used the Facebook app and the Google Fit app. We removed all permissions associated with the providers on both the apps. We also set all the providers' exported setting to true.

## 4.2 Scenario 2: App checks for signatures

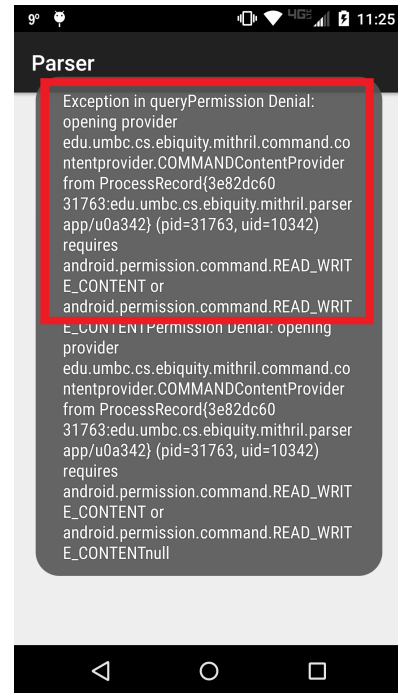
Upon installation the repackaged Google fit app immediately crashed and kept on crashing every time we tried to use it. Therefore, in order to figure out the issue we used logcat, the Android logging system that provides a mechanism for collecting and viewing system debug output. We discovered from logcat messages that the Google Fit app had included an additional check on the app key signature and it simply crashed because the signature is detected as unknown. You can see the error in Figure 5.

## 4.3 Scenario 3: App manages access control to it's components

For this case we used a repackaged Facebook app. We observed that the app never crashed and worked like a normal Facebook app. However, when we tried to access the app's content provider it blocked our attempts and you can see in Figure 6 that Facebook controls access to it's own component using a custom protection mechanism. Therefore, app developers are clearly detecting such issues on their apps but not always.

## 4.4 Scenario 4: Potentially vulnerable app

We found at least one app called Expense Management



**Figure 4: Android content provider permission denial**

```
Process: com.google.android.apps.fitness, PID: 5528
java.lang.SecurityException: Caller isn't signed with recognized ke
ys!
at android.os.Parcel.readException(Parcel.java:1599)
at android.database.DatabaseUtils.readExceptionFromParcel(Database
Utils.java:183)
at android.database.DatabaseUtils.readExceptionFromParcel(Database
Utils.java:135)
at android.content.ContentProviderProxy.call(ContentProviderNative
.java:646)
at android.content.ContentProviderClient.call(ContentProviderClie
```

**Figure 5: Google Fit app checks for certificate signatures**

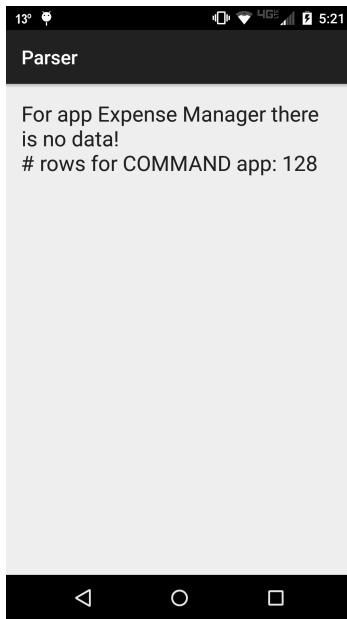
from our random sample set that allowed us access to it's content provider. However, we did not know the complete URI for the app's content provider. Therefore we had to make guesses. We repackaged this app too and following the same process as the previous apps, we observed that it did not implement any checkpoints. Also it doesn't cause any errors as we saw in the above-mentioned scenarios. You can see in Figure 7, that our query did not return any data but that was because the app wasn't writing it's data to it's database. We are still investigating other apps for a potential breach that could lead to a full fledged exploit. We are currently processing more apps to find out if they include such checks as encoded by popular apps from Google or Facebook. This processing takes a long time because we have to manually find the databases on the phone using a rooted phone and a SQLite explorer app. Thereafter, we have to make guesses for patterns that apps might have used in their content provider code. There are commonly used patterns like '#' that can be used as a part of the URI. Such a generated URI can then be used to call the content

```

java.lang.SecurityException: Component access not allowed.
at com.facebook.content.PermissionChecks.a(PermissionChecks.java:163)
at com.facebook.content.AbstractContentProvider.c(AbstractContentProvider.java:243)
at com.facebook.content.AbstractContentProvider.d(AbstractContentProvider.java:249)
at com.facebook.content.AbstractContentProvider.query(AbstractContentProvider.java:410)
at android.content.ContentProvider$Transport.query(ContentProvider.java:238)
at android.content.ContentProviderNative.onTransact(ContentProviderNative.java:112)
at android.os.Binder.execTransact(Binder.java:453)

```

**Figure 6: Facebook checks controls access to its component**



**Figure 7: No check points were found on a less popular app**

provider and obtain access to the data. We are trying to use such patterns to find out apps which have such a vulnerability.

## 5. RELATED WORK

A lot of work has been done in trying to detect malicious apps and malicious behaviors on mobile devices before. A survey by Enck [4] discusses most of the state-of-art in smartphone research, including efforts in designing new OS protection mechanisms, as well as performing security analysis of real apps. These techniques have included detecting re-delegation of permissions studied by Felt et.al. [6] in which an app with higher privileges performs tasks for another app with less privileges. A study by Zhou et.al. [15] have looked at ways for detecting malware by studying their characteristics like installation methods, activation mechanisms and malicious payload nature. Permission analysis research done by Barrera et al. [1] used Self Organizing Maps (SOM) to analyze permission usage. The WHYPER framework in-

troduced by Pandita et. al. [12], on the other hand tried to match purpose to permission. They used Natural Language Processing (NLP) techniques to identify the need for a permission in an app by reading the app's description. In another work done by Gorla et. al. [8] apps were clustered by descriptions and outliers were detected with respect to their API usage. They used the CHABADA prototype to carry out the behavior matching activity.

In our work we carry out app metadata analysis and detect apps with certain vulnerabilities as per our heuristics. Our work focuses on generating rules or recommendations that are based on heuristics. Our system is capable of incorporating the results of all these studies as heuristics to detect malicious behavior in addition to our own heuristics.

In a study carried out by Huckvale et. al. [9] it was found that despite the tests carried out by National Health Service (England) to ensure clinical data safety standards, apps had flouted privacy standards and sent data without encryption. This is the reason we are focusing on the loophole in content provider permission and the lack of any standard mechanism to verify whether a requester has access permissions or not. Expecting that the app developer would ensure security and leaving this loophole in Android exposes users' data to potential attacks.

At present our work focuses on apps from the official Google Play Store. We intend to include other app stores too in the future as done in the work done by Lindorfer et.al. [11] at the International Secure System Lab. The ADMIRE system created by them lets you analyze different Android marketplaces, searching potentially malicious applications. This project also provides a score to various app markets using the reaction time to deletion and other factors when a new malware app shows up on their store.

There have been multiple attempts at achieving the goal of properly managing access control on mobile (Android) devices. Efforts have been made by the open source community through the XPrivacy project (needs a rooted phone), the Privacy Guard project (available on Cyanogenmod, a custom Android ROM) and the PDroid application (needs a rooted device). Research project by Conti et. al. [3] (CRePe), Enck et al. [5] (TaintDroid) and Jagtap et al. [10] (Preserving Privacy in Context-Aware Systems) have made similar efforts. CRePe described a system where security policy enforcement was carried out based on context of the smart phone. TaintDroid was a research effort where the data flow on an Android device was studied to figure out when sensitive data left the system via an untrusted application. The work of Jagtap et al. [10] focused on constraining data flow in a context-aware system using a policy-based framework. A related work by Ghosh et al. [7] used a similar policy driven approach to constrain application permissions based on context. In our work we are generating recommendations by studying app metadata and apks. We allow system administrators to choose from these recommendations and possibly make them part of their corporate policy.

## 6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented Heimdall, a system that is capable of detecting security vulnerabilities on a mobile device. We use a heuristics based approach to detect these vulnerabilities. We also discussed a potential loophole in Android's custom content provider's security. This loophole could allow a malicious app to steal users' data from their



phones. We built a proof-of-concept for demonstrating this loophole and we also tested a random sample of 1500 apps to find out if this vulnerability exists in them. We reverse engineered two popular apps and a non-popular app, to see if there are additional checks present in the code, to handle access control to data. We presented our observations made in our evaluation process which led us to conclude that such checks are possible and are present in some apps but not all apps. In conclusion we can say that this is a potential loophole that could lead to user data leakage and thus have serious implications. Therefore, there needs to be some checking mechanism in form of an API to verify app signature keys or to verify component access control or maybe even strict permission association for custom providers.

In future, we hope to include more heuristics in Heimdall and capture more such vulnerabilities. As we have discussed in the related work section, a lot of research has been undertaken in this area and we hope to incorporate the ability to detect these vulnerabilities in Heimdall. Since the Heimdall project's primary goal is to be deployed in a BYOD scenario we are working on a mechanism to actually control the data flow on Android. This will allow us to study what data is being transferred to and from the phone as well as implement policies defined by the system administrator. Detecting discrepancy between app's expected behavior and actual behavior is also being studied by many researcher but we feel this problem still remains unsolved and we hope to make that into a feature of Heimdall.

## 7. ACKNOWLEDGMENTS

Support for this work was provided by NSF grants 0910838 and 1228198.

## 8. ADDITIONAL AUTHORS

Additional authors: Ting Zhu (University of Maryland, Baltimore County, email: [zt@umbc.edu](mailto:zt@umbc.edu)) and Tim Finin (University of Maryland, Baltimore County, email: [finin@umbc.edu](mailto:finin@umbc.edu)).

## 9. REFERENCES

- [1] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10*, pages 73–84, New York, NY, USA, 2010. ACM.
- [2] R. Chang. 10 alternative android app stores, 2014.
- [3] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In M. Burmester, G. Tsudik, S. Magliveras, and I. Ilic, editors, *Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 331–345. Springer Berlin Heidelberg, 2011.
- [4] W. Enck. Defending users against smartphone apps: Techniques and future directions. In *Proceedings of the 7th International Conference on Information Systems Security, ICISS'11*, pages 49–70, Berlin, Heidelberg, 2011. Springer-Verlag.
- [5] W. Enck, P. Gilbert, B.-G. Chun, L. P., Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6, 2010.
- [6] A. P. Felt, S. Hanna, E. Chin, H. J. Wang, and E. Moshchuk. Permission re-delegation: Attacks and defenses. In *In 20th Usenix Security Symposium*, 2011.
- [7] D. Ghosh, A. Joshi, T. Finin, and P. Jagtap. Privacy control in smart phones using semantically rich reasoning and context modeling. In *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*, pages 82–85, 2012.
- [8] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 1025–1035, New York, NY, USA, 2014. ACM.
- [9] K. Huckvale, M. Car, C. Morrison, and J. Car. Apps for asthma self-management: a systematic assessment of content and tools. *BMC Medicine*, 10(1), 2012.
- [10] P. Jagtap, A. Joshi, T. Finin, and L. Zavala. Preserving privacy in context-aware systems. In *Fifth IEEE Int. Conf. on Semantic Computing (ICSC)*, 2011.
- [11] M. Lindorfer, S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis. Andradar: Fast discovery of android applications in alternative markets. In S. Dietrich, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 8550 of *Lecture Notes in Computer Science*, pages 51–71. Springer International Publishing, 2014.
- [12] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *Proceedings of the 22Nd USENIX Conference on Security, SEC'13*, pages 527–542, Berkeley, CA, USA, 2013. USENIX Association.
- [13] Statista. Number of available applications in the google play store from december 2009 to july 2015, 2015.
- [14] C. Trout. Android still the dominant mobile os with 1 billion active users, 2014.
- [15] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, May 2012.