Short Paper: A Look at SmartPhone Permission Models

Kathy Au, Billy Zhou, Zhen Huang, Phillipa Gill and David Lie
Dept. of Electrical and Computer Engineering
University of Toronto, Canada

ABSTRACT

Many smartphone operating systems implement strong sandboxing for 3rd party application software. As part of this sandboxing, they feature a permission system, which conveys to users what sensitive resources an application will access and allows users to grant or deny permission to access those resources. In this paper we survey the permission systems of several popular smartphone operating systems and taxonomize them by the amount of control they give users, the amount of information they convey to users and the level of interactivity they require from users. We discuss the problem of permission overdeclaration and devise a set of goals that security researchers should aim for, as well as propose directions through which we hope the research community can attain those goals.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls, Information flow controls

General Terms

Design, Human Factors, Security

Keywords

Smartphone, Permissions

1. INTRODUCTION

Smartphones have grown in popularity in recent years. According to an informal study, the number of smartphones increased 74.4% worldwide, with a total of 302.6 million units shipped in 2010[1]. A large part of the popularity can be attributed to the ability of smartphones to run 3rd party applications, which is a defining feature of smartphones. These applications extend the utility of smartphones making them essentially general computing devices compared to their simpler "feature" phone and "dumb" phone predecessors.

Having been developed in today's security-sensitive environment, the operating systems (OS) for smartphones incorporate stronger sandboxing for 3rd party applications than previous consumer oriented operating systems. These sandboxing systems isolate applications from each other and from the resources on the phone by default. To access a sensitive resource on the phone the sandbox system implements a permission system that requires users to grant permission, either explicitly or implicitly, to the application. Some examples of sensitive resources may include personal contacts, the user's location or Internet access. While sandboxing is a standard security mechanism to mitigate the threat of malicious software, permission systems are a recent phenomenon that is becoming more and more widely used. Not only is some form of permission system a feature of almost every smartphone OS, but it is also used to constrain 3rd party applications in the Google Chrome Browser and on Facebook's application platform.

We define a permission system as having the first and possibly the second of the following properties. First, a permission system enables the user to define a per-application policy that constrains what resources an application may access on their phone. Second, a permission system communicates information to the user about what resources an application accesses or might access in the future. The second property can be thought of as a communication channel for the application, and indirectly, the application developer, that conveys to the user how much the user is trusting the application when they use it.

There has already been much work on analyzing the use of permissions in the Android OS, which is currently the most popular smartphone OS [1, 2, 3, 4, 5, 8]. The existing literature points out various problems with the permission system, such as the difficulty of interpreting the meaning of the plethora of permissions requested, as well as the lack of a way to convey that certain combinations of permissions are far more dangerous than the individual permissions in isolation. Furthermore, there is a large amount of anecdotal evidence, as well as a more recent academic study [3], that indicate that permission systems suffer from the problem of *overdeclaration*, where developers request more permissions than what they need. There are two drawbacks to overdeclaring permissions. First, overdeclaring breaks the principle of least privilege. By granting more privileges to an application than it actually needs, users open themselves up to more severe consequences should the overprivileged application have an exploitable vulnerability. Second, an application developer who overdeclares permissions may lose potential users who balk at installing an application that is asking for too many permissions.

In this paper, we begin with a survey of permission systems

[1] `http://technolog.msnbc.msn.com/_news/2011/02/07/6005519-smart-phone-growth-explodes-dumb-phones-not-so-much`

SPSM'11, October 17, 2011, Chicago, Illinois, USA.

OS	Initial Release Date	# of permissions	Control	Information	Interactivity
Android	2008/09/23	75[a]	Medium	High	Low
Windows Phone 7	2010/10/11	15	Medium	Medium	Low
Apple iOS	2007/06/29	1	Low	Low	Low
Blackberry OS	2006 Q3[b]	24	High	High	High
Maemo	2005/11/–	0	None	None	None
WebOS		1	Low	Low	Low

Table 1: Summary of smartphone OS permission models.

[a]This only includes permissions available to 3rd party applications. There are 137 total permissions.
[b]This is release date of the Blackberry 8100, which is the first Blackberry phone capable of running 3rd party applications.

across several popular smartphone OSs in Section 2 and then taxonimize and compare the various permission systems. We then propose a set of research goals for solving the overdeclaration problem in Section 3 and conclude in Section 4.

2. SURVEY OF PERMISSION SYSTEMS

We examine the permission systems of several modern smartphone OSs. We focus on three properties of the systems:

- **Control:** This indicates how much control the permission system gives the user over applications. An example of control is whether permissions can be individually enabled and disabled.
- **Information:** We also categorize permission systems by how much information they convey to the user. Permission systems can convey two types of information – what resources (and thus permissions) the application developer believes their application will access (a priori) and what resources the application *actually* accesses at run time.
- **Interactivity:** Finally, we indicate how much of a burden the permission system is on the user by indicating how much interaction is required to use the system. Some permission systems require a lot of interaction because they prompt the user frequently, while others take measures to reduce the amount of interaction.

We summarize the results of our analysis in Table 1 and give details of our analysis below.

Android: The Android permission system consists of four types of permissions. Two of the permission types, *Signature* and *System*, are reserved for applications that have been signed with a key available to the firmware developer or come installed by default on the firmware. These permissions are not available to 3rd party applications. The other two types of permissions are *Normal* and *Dangerous*. Normal permissions are automatically granted to the application without user involvement and so the user does not have a chance to deny these permissions before installation (though she may always examine them and uninstall the application afterwards). Dangerous permissions are presented in a prompt at installation time of the application. If the user proceeds with the installation, then the dangerous permissions are permanently granted to the application. The developer must declare what permissions the application needs in a file that is later included with the application installation package. As of Android 2.3.3, there are currently 75 dangerous and normal permissions available to 3rd party developers, making Android the most complicated permission system. While Android has many individual permissions compared to other OSs, users may only grant all requested permissions or deny them all by not installing the application. As a result, we feel it only gives a medium level of control to users. Users are given information about what permissions the developer believes her application needs, but not about what permissions the application actually uses. However, because of the large number of permissions, we feel it still conveys a high amount of information to users. We note that it is debatable whether this form of additional information is actually useful or more confusing for the user. Finally, users only interact with the permission system when they install applications and even then, only need to select between proceeding with the installation or cancelling it, making the level of interaction for Android low.

Windows Phone 7: The permission system in Microsoft's Windows Phone 7 OS bears many similarities to that of Android. Permissions are called capabilities in Windows Phone 7. One difference is that instead of the 75 permissions available to 3rd party applications in Android, Windows Phone 7 only provides 15 capabilities for developers to choose from. In addition, the Windows Phone 7 development environment provides a tool, called the "Capability Detection Tool", which tries to automatically detect which capabilities an application needs through static analysis of the application code. Users are informed of the capabilities an application is requesting from the application page on the market, but are not prompted again at installation time, except for legal disclaimers. Finally, the user is informed at application run time the first time the application requests the user's location. As in Android, users may not provide a subset of requested capabilities, but may only grant them wholesale or deny them all by not installing the application. Because of the strong similarities, we rate Windows Phone 7 the same as Android in control and interactivity, but give it a medium level of information because of the coarser permission categories.

iOS: Apple's iOS, which runs on the iPhone and iPad, does not have as comprehensive a permission system as Android or Windows Phone 7. Much of the security against malicious applications relies on manual inspection that is done by the Apple App Store. The viability of this manual process has been cause for concern [7]. iOS has no explicit permission interface. The first time an application attempts to access the user's location, the user is presented with a prompt asking if they will allow it. After that, a user must actively navitage to a menu in the iPhone settings that displays which applications have accessed the location resource in the last 24 hours. The same screen also allows a user to revoke location permission from an application. As a result, we rate iOS as low in all categories.

Blackberry OS: Research in Motion's Blackberry OS (version 6) contains 24 permissions available to 3rd party applications. However, rather than requesting the permissions up front, users have a default set of permissions, which are automatically granted to each application. At any time, the user may revise these permissions to be more strict or more liberal. If an application tries to perform an action for which they do not have the required permission, the user is given a prompt. The user may permanently allow, permanently deny or grant the permission for one time and thus continue to receive prompts. However, we note that applications are generally not written to function with a partial set of permissions. As a result, in practice, the Blackberry OSs ability to partially grant permissions is effectively all or nothing in many cases, since denying even a single permission causes some applications to malfunction. The Blackberry OS does not provide a channel for developers to communicate their intended use of permissions up front, though developers may choose to include this information in their documentation or as part of the application startup. However, the interactive prompts give users a great deal of information about what permissions an application is actually using. Because of the fine-grain control and information given to users and the high level of interactivity caused by the prompts, we rate the Blackberry OS high in all categories, though we re-emphasize that it is likely that these advantages are not likely realizable in practice due to the way applications are written.

Maemo: Nokia's Maemo operating system is based on Linux and is the successor to the Symbian operating system. Maemo is essentially a phone-specific distribution of Linux and implements the same security model as Linux. All applications execute as the same Linux UID and thus there is no isolation between applications. Maemo does not implement a permission system – it neither provides the user the ability to constrain applications, nor does it provide a way for the user to know what operations an application is performing on their phone.

Discussion: The choices in permission system design illustrate the classic tension between providing fine-grain control and information to the user, and reducing the effort on the part of the user to maintain security. Over time, smartphone OS architects have refined their permission models to improve this trade-off. Older OSs, such as Maemo, iOS and Blackberry OS sit at opposite ands of this spectrum, either providing a lot of information and control at high cost in user effort, or not providing any at all. In contrast, the two most recent OSs, Android and Windows Phone 7 have nearly identical permission systems that give users some control and information, but still offer a low level of interactivity by moving the granting of permissions to install time. Unfortunately, moving the permission granting to install time means that users may be asked to grant permissions that the application never needs or uses, resulting in the problem of *permission overdeclaration*. Interestingly, there seems to be demand for even more control without increasing the level of interaction by some users. Custom modifications of the open source Android OS enable users to grant or deny individual permissions for an application [9,10]. It remains to be seen whether this capability will be useful in practice, since some Android applications will not function properly with only partial permissions (as demonstrated by AppFence [6]), and whether these enhancements will be adopted by the official Android OS.

3. RESEARCH AGENDA

3.1 Problem description

Permission systems such as Android and Windows Phone 7, where the developer must declare what permissions their application needs up front, suffer from the problem of permission overdeclaration. The main goal of the developer is to get their application working so it can be placed on the market as soon as possible, thus enabling users to download and start using it. On one hand, overdeclaring permissions instead of underdeclaring leads to a higher probability that their application will work and thus reduces the effort needed to develop an application. On the other hand, there are claims that overdeclaration causes some users to reject an application [2].

Android Version	Release Date	Permissions Changed
1.0	2008/09/23	-
1.1	2009/02/09	2
1.5	2009/04/30	5
1.6	2009/09/15	9
2.0	2009/10/26	4
2.0.1	2009/12/03	0
2.1	2010/01/12	1
2.2	2010/05/20	5
2.3	2010/12/06	6

Table 2: Android releases and permission churn. A Permission Change represents a permission that has been added, removed or deprecated.

Since there are good reasons for a developer to overdeclare and not to overdeclare, we may ask why are developers choosing to overdeclare? If we examine the rate of Android OS releases, tabulated on Table 2, we see that there has been an Android release very 3 months on average with 4 permissions changing on average (either added, removed or deprecated) with each release. Thus, one might conclude the tendency towards overdeclaration is because permissions are constantly changing underneath the developers. However, this is not the case. If we examine the permissions that are most overdeclared in the study by Felt et al. [3], we see that there is nearly no overlap between the overdeclared permissions and the removed or deprecated permissions. As a result, the high rate of churn does not lead directly to overdeclaration.

If we dig a bit deeper, a commonly cited problem with Android permissions is the poor accuracy of the documentation that maps the permissions required for different types of application actions. Given the rapid rate of Android releases and churn in the permission system, it is not a surprise that Android developers have neglected to produce complete and accurate documentation governing the permission system. On various Android development newsgroups, developers often express frustration at bugs whose underlying cause turn out to be inadequate permissions – the lack of documentation greatly increases the effort to determine what permissions their application needs. Unfortunately, there is very little data on how the number of requested permissions factors into a user's decision to install or not install an application. As a result, developers perceive little benefit from deriving the precise set of permissions required. Thus, the benefits of overdeclaring are clear to the application developer (their application will work), while the drawbacks are unclear.

3.2 Goals

The underlying cause of overdeclaration is due to the imbalance between the difficulty of correctly declaring permissions and the unclear benefits of doing so. While this imbalance currently favors overdeclaring for the developer, this is not a situation that benefits the end users. Overdeclaration gives applications unneeded privileges, which puts the user at greater risk should an application be compromised. As a result, stopping overdeclaration is a worthwhile research goal.

[2]Facebook explicitly warns developers to not overdeclare stating that applications with fewer permissions are installed by more users – see `http://developers.facebook.com/docs/guides/canvas/#auth`

To stop overdeclaration, the balance between the cost of correctly declaring permissions and the benefits of doing so must be reversed, so that developers are motivated to correctly declare permissions. Thus we define several research goals to solve the overdeclaration problem.

Lower the costs of determining the correct set of permissions. The effort to determine the correct set of permissions should be made as low as possible – ideally zero. This can be accomplished with a tool that automatically determines what permissions are needed by an application. Microsoft already includes a tool that tries to detect what permissions an application needs. In addition, Felt et al. [3] present a tool, called StowAway, that can extrapolate the needed permissions of an application from the application binary. However, both tools currently cannot extract required permissions accurately for reasons we discuss in Section 3.3.

Make the costs of overdeclaration explicit to developers. Even if the effort to correctly declare permissions is reduced, developers must still weigh the effort that must be invested versus other development activities, such as adding new features or fixing bugs that users are complaining about. The problem is that overdeclaration is not a "bug" that users generally complain about, so developers have little motivation to fix the problem. The role that permissions play in a user's decision to install an application can be studied and quantified so that developers can fully weigh the benefits of correctly declaring permissions against the costs of not doing so. For example, if developers were made aware that by removing a certain permission, they may increase their user base by some percentage, this may motivate them to see if that permission is actually required.

Make developers aware which components in their application are using which permissions. Once the above are achieved, then developers can start making informed decisions about whether a particular feature increases the number of users because of its desirability, or drives away users because of the permissions it requires. Not all permissions are equal and some permissions (such as Internet access or location) are likely to raise the ire of some users more than others. Being able to tie permission use to specific components and features would be a final goal for research on permissions in smartphone OSs.

3.3 Proposed Solution

In the previous section, we proposed the creation of a tool that can automatically determine the permissions an application needs. To reliably do this one must extract a mapping between the APIs an application may exercise and the permissions the application needs to use those APIs. In StowAway, the mapping between APIs and permissions is extracted using fuzz testing of the API interface (and made available on their web page). However, this suffers from difficulty in getting complete coverage, as well as difficulty in getting realistic arguments from the fuzz testing tool for the API calls. The Microsoft Capability Detection tool relies on a set of rules, likely derived from the smartphone OS developers themselves. While this is likely to be more reliable, it requires effort on the developers part to keep it up to date, and thus is just another instance of the problem of maintaining accurate documentation of an OS that is undergoing rapid development. Ideally, an automated method could be devised that could automatically extract the mapping by performing static analysis on the source code of the OS.

We are currently exploring the use of static analysis on the source code of the Android OS to extract the mapping between APIs and permissions. The goal is to detect all program flows from APIs to permission checks. While this has the potential to be complete as compared to dynamic methods such as fuzz testing, the main limitation with any static analysis is that it is difficult to make scalable. In the case of Android, this is a particularly challenging problem. The Android code base includes approximately **XXX** lines of source. Further more, the path between an API call and a permission check may traverse several processes, as well as different languages, making traditional program flow analyses that normally only handle one language and a single name space inadequate.

Kathy to fill in?

- While the permission checks are spread out, most are done in JAVA (X/X?)
- To make things scalable we do a flow-insensitive call-graph analsys to find paths between API calls and permission checks
- some functions have many branches to other subroutines, we identify these and do a flow-sensitive analysis only on these to avoid loss of precision.
- RPC's are another challenge because the control flow is not explicit in the source code, goes through binder. Also spans names spaces so variables may change name?
- RPCs are handled by parsing the AIDL interface description files that are used to compile RPC interfaces in Android.
- we plan to compare our results with the fuzz testing results in StowAway [3].

4. CONCLUSION

We conclude that the trade-off between the obvious benefits of overdeclaring to save time and the less apparent drawback of losing potential users, currently seems to favor overdeclaring for developers. On the other hand, for users, overdeclaring is harmful in that it runs counter to the security principle of least privilege. As a result, we feel that the best way to solve this problem is to produce tools that reduce the effort required of developers to detect what permissions are needed, and a secondary way is to feedback to developers which permissions are most worrisome to potential users. For completeness, such a tool would have to rely on static analysis, but static analysis of a large operating system that spans different components and languages remains challenging. However, despite being developed at different times and by different people, the permission systems of all smartphone OSs bear many similarities. Thus, we believe that a domain-specific approach may be key to solving the overdeclaration problem.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments. Kathy is supported by an OGS scholarship and Phillipa is supported by an NSERC scholarship. The work in this paper was also supported by the NSERC ISSNet Strategic Network, and an NSERC Engage Grant.

5. REFERENCES

[1] D. Barrera, H. Kayacik, P. 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 2010)*, Oct. 2010.

[2] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, Nov. 2009.

[3] 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 2011)*, Oct. 2011.

[4] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, June 2011.

[5] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, Aug. 2011.

[6] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. "These aren't the droids you're looking for": Retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, Oct. 2011.

[7] K. Noyes. Why Android app security is better than for the iPhone. PC World Magazine, 2011. `http://www.pcworld.com/businesscenter/article/202758/why_android_app_security_is_better_than_for_the_iphone.html`.

[8] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Proceedings of the 25nd Annual Computer Security Applications Conference (ACSAC)*, Dec. 2009.

[9] senk9. How to control Android app permissions (Root/CM7). `http://senk9.wordpress.com/2011/06/19/how-to-control-1111-app-permissions-rootcm7/`.

[10] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST 2011)*, June 2011.