Short: Security and System Architecture: Comparison of Android Customizations

Roberto Gallo
IC-UNICAMP, KRYPTUS
gallo@kryptus.com

Patricia Hongo
IC-UNICAMP
patricia.hongo@gmail.com

Ricardo Dahab
IC-UNICAMP
rdahab@ic.unicamp.br

Luiz C. Navarro
IC-UNICAMP
lcnavarro@globo.com

Henrique Kawakami
IC-UNICAMP, KRYPTUS
kawakami@kryptus.com

Kaio Galvão
IC-UNICAMP
kaio.karam@gmail.com

Glauber Junqueira
SIDI
glauco.b@samsung.com

Luander Ribeiro
SIDI
luander.r@samsung.com

ABSTRACT

Smartphone manufacturers frequently customize Android distributions so as to create competitive advantages by adding, removing and modifying packages and configurations. In this paper we show that such modifications have deep architectural implications for security. We analysed five different distributions: Google Nexus 4, Google Nexus 5, Sony Z1, Samsung Galaxy S4 and Samsung Galaxy S5, all running OS versions 4.4.X (except for Samsung S4 running version 4.3). Our conclusions indicate that serious security issues such as expanded attack surface and poorer permission control grow sharply with the level of customization.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Android customizations, security architecture, permissions

1. INTRODUCTION

Security issues on mobile devices have been reported on a growing rate: in 2014, two thousand new Android malware samples were discovered daily, a 100% growth over the previous year [19]. Research shows that implementation defects and poor architectures are root causes of roughly the same number of security issues on a typical system [12]. Nevertheless, architecture-centric security analysis is seldom performed on real-scale systems and a limited set of tools exist for that end. Our main contribution here is to show that Android OS customizations vary greatly in key aspects, when architectural security is concerned.

WiSec'15, Jun 22-26 2015, New York City, NY, USA

ACM 978-1-4503-3623-9/15/06...$15.00.
http://dx.doi.org/10.1145/2766498.2766519.

Mobile devices are now pervasive and their complexity is comparable to that of any other full-fledged computing device, such as servers and personal computers. However, increased system complexity means, in general, not only more functionality and larger source-code, but also more intricate system architectures. It is well established that security issues and system complexity are strongly correlated [15]. However the root causes of security issues within a complex system may have quite different natures.

This paper is organized as follows: In Section 2 the problem is further defined and related work is presented. In Section 3 we present our proposal and contributions. Section 4 concludes and presents future work.

2. ARCHITECTURAL SECURITY AND RELATED WORK

There are many definitions of system architecture, but most are equivalent or similar to the following: "Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment... [11]." An architectural approach to system design is fundamental for a number of reasons, such as complexity handling and system analysis. In fact, system design is known to be an NP-hard problem and, according to Chapman et al [5], "this is the primary reason why engineers do not try to produce optimal systems: they merely produce designs that are good enough." While a "good enough" architecture is usually sufficient in several domains, when security is concerned several complications may arise.

Differently from general purpose architectural representations, security-aware descriptions must be expressive of all relations affecting a given security objective. For instance, consider confidential content x controlled by an Android app, whose (presumably) sole default access is done by an "activity", and that denied accesses are logged using the system's service. If this service is not carefully implemented, it may be the case that meta-information about x is found in the app's log, thus potentially violating x's confidentiality.

There are a number of ways Android components can communicate, either via "content providers", "activities", "services", or the OS's native libraries. In order to discipline and protect access, Android uses its well-known permission-based scheme. This scheme, implemented in the OS middleware is not perfect, as the seminal work of Felt et al [7] shows: they found unprivileged components, missing permissions, exposed non-public permissions, etc. These results were also confirmed by Au et al [2], who showed how permissions can be leveraged for system attacks [18, 4, 14].

In spite of Android's middleware limitations, permissions are

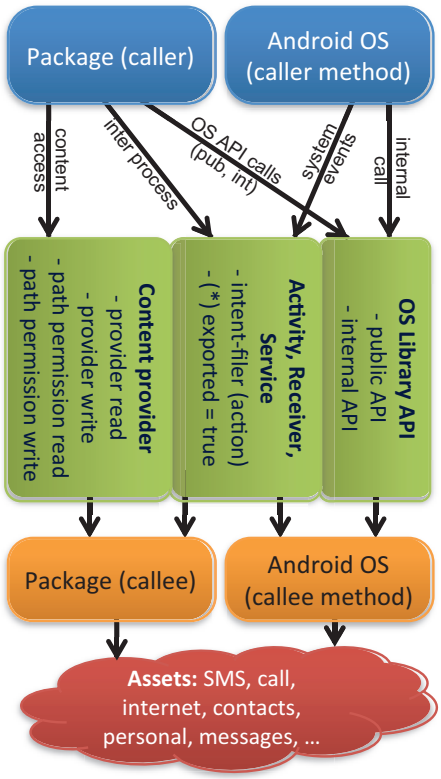

Figure 1: Permissions can be used both by software packages running on the middleware and Android OS components. Permissions protect different types of access, grouped in the green boxes. The access methods exposed by packages and components (in orange) are used to broker access to system and user assets (in red). Packages and components can use and consume multiple resources at the same time.

the cornerstone of Google's OS security, guarding the vast majority of the components' interaction channels, as shown in Figure 1. Since permissions are directly related to interactions between components, they are used, in Section3, as the basis of our security-aware architectural view of the different Android distributions, upon which we perform our security analysis.

Architectural security is clearly important but not as widespread in the software community when compared to hardware. UMLsec [13] presents a UML profile and a method to evaluate the security of general systems, but it lacks sufficient support for describing systems in fine detail. As for hardware architectural security evaluation, Potlapally [16] used the TLA+ language to build a high-level formal model of the hardware and map known vulnerabilities. Gallo et al [8] present a generic probabilistic security evaluation framework and tool. When considered Android customization comparison, Zhou et al [22] brings in-depth analysis of hundreds of customized/localized, public available Android images. Their analyses go from device drivers to source code to some permission interaction. Also Wu et al [21] compare 10 Android customizations, bringing metrics on permission usage, in the search for over-privileged packages, and vulnerability estimations. Our approach is different from both works on a number of ways, as we focus on the relations of all permissions at the same time, on a unique visualization format and we target specific devices with newer Android versions.

3. OUR CONTRIBUTIONS

3.1 Introduction and Methodology

Usually, system design has the definition of the architecture preceding implementation. However, unless automated code generation is used and tight project management is in place, there is a gap between the actual system architecture and the specified version, leading to *architectural erosion* [20]. Recovering from erosion is a hard and well know problem in Software Engineering. Unless systematic software patterns are used, automatic full recovery of general software architectures is computationally infeasible.

However, because we were interested in comparing real Android customized architectures with thousands of elements, we needed to be able to automatically reconstruct partial architectural descriptions of the target systems. For this end, we used a permission-based scheme which has both the security representativeness we need, and is common to most Android distributions.

Employing a custom-designed app, we gathered all permission-related data from five different Android devices in factory state, namely a Google Nexus 4, a Google Nexus 5, a Sony Z1, a Samsung S4 and a Samsung S5, all running Android version 4.4.X (except Samsung S4 which is running 4.3). Using suitable permissions, our app was able to examine every package installed in the target systems, thus gathering permission-related data.

Software packages may both **provide** and **use** permissions. When providing permissions, packages may assign a security level to a resource, these levels being `normal`, `dangerous` and `signature`, and `signatureOrSystem`, `signatureOrSystemOrDevelopment`, among other possible combinations. This assignment is done within the manifest file.

Of special interest to our analysis are the `dangerous` and `signature*` permissions, because of their impacts. Resources are (usually) marked as `dangerous` when they can lead directly to valuable assets or points where adversaries may act upon. Dangerous permissions examples include the self-explanatory `android.permission.INTERNET`, `android.permission.WRITE_EXTERNAL_STORAGE`, and `com.samsung.android.memo.READ`.

If granting `dangerous` permissions to apps is a security concern, the use of `signature*` can serve as a protection mechanism. When permissions are marked as `signature`, the Android permission manager restricts the permission usage so that only providers and user packages signed by the same vendor certificate can interact, and `signatureOrSystem` gives access also to packages installed in the system partition. In other words, this mechanism can be used to group packages so that a third party cannot ask for potentially sensitive permissions.

From the package and permission information gathered by our app, we reconstructed a directed graph representing all relations between software packages visible to the app, enabling us to analyse each target system in terms of its architecture.

3.1.1 A Simple Example of an Architectural Flaw

Here we illustrate an architectural flaw due to permission leakage [9]. Taking an architectural view of Android, where components are software packages, resources, or individual permissions, and relations represent granted or exposed permissions, then the graph representation of this abstraction is that of Figure 2.

In this graph, `CustomApp` has direct access to `CriticalResource`, (an asset) controlled (exposed) by Android OS by means of an explicit permission grant. This granting could be either performed by the user or statically embedded in the O.S. distribution, maybe due to vendor customization. When the user installs `DubiousApp`, permission is requested upon `CustomResource` (exposed by `CustomApp`, and for which `CriticalResource` is not directly associated.

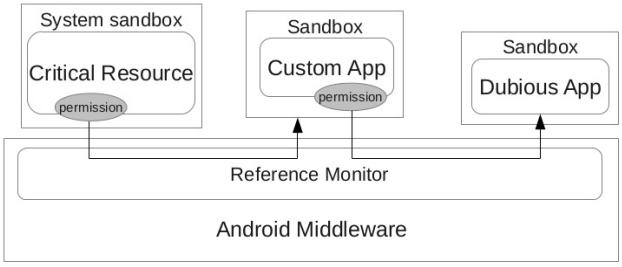

Figure 2: Permission escalation attack

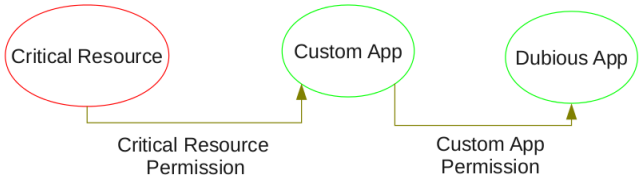

Figure 3: Permission escalation graph

It happens, however, that if `CustomApp` provides indirect access to `CriticalResource` by `CustomResource`, then `DubiosApp`, in fact, can deceive the user and compromise security. This exact scenario was used by Soundcomber, a sound Trojan with few permissions capable of stealing sensitive data, such as a user's credit card number, through a suite of techniques applied to information obtained from the audio sensor [17]. As Soundcomber does not request Internet access permission, transmission of data to a command-and-control server can be performed by making use of an application with network access. One way to do so is to request a web browser to access url `http://target?number=$<$creditcardnumber$>$` via malicious Intent.

3.1.2 *Classes of Architectural Flaws*

In the previous section, an architectural arrangement was the cause of a vulnerability. However, as `DubiousApp` got access to `CriticalResource` through a valid permission path, one could argue that it should not be classified as an architectural flaw but, rather, a misuse. We disagree; as we see it, if architecture allows for accidental misuse, then it is not sound. Nevertheless, we do recognize the fact that there are *classes* of security issues. As a result, we propose the following classification of architectural flaws:

- **Class I - Protection missing.** In this class, a critical parameter (user data, keys, etc.) is left unprotected to adversary direct or indirect exploitation, so that the adversary does not need to abuse individual components âĂŞ access is granted through regular interfaces. Systems without MAC (mandatory access control) usually fall into this category. Examples in older Android versions include permission to automatically start at boot and start on install [1];
- **Class II - Single point of failure.** Here, the failure or the exploitation of a single component (which may have subcomponents) suffices and allows the adversary to have wide access to critical parameters. Failures of sandboxes running with root privileges are typical examples;
- **Class III - Confusion.** In this class, security issues are related to architectures prone to misuse. Usually, misuse is related to the gap between the *expected* security policy and that implemented by the architecture. This gap, in turn, can be related either to complexity or to ill-defined protection mechanisms (see Section 2. Permission leakage on Android platform is an example.

3.2 Architectural Analysis and Discussion

The graph with the extracted package-permission information for each target system was loaded onto the Tulip framework [3]. Tulip is capable of not only producing visualizations for huge graphs but also enables the automation of graph manipulations and metrics calculations such as those in Figure 4 and Table 1.

In Figure 4, the complete permission graphs for the Google Nexus 5 (upper, smaller graph) and Samsung S5 smartphones are shown. In the graphs, green nodes represent software packages, while pink nodes represent assets (resources). A brown edge means that a given software package (green node) was given permission to *use* a given resource (pink node). The remaining edges represent permissions provided by a given package, with the following color convention: `normal`=blue, `dangerous`=orange, `signature`=gray, `signatureOrSystem`=purple, `signatureOrSystemOrDevelopment`=cyan. Table 1 shows comparative information from the five Android devices, including the two represented on Figure 4.

3.2.1 *Key Findings*

One important feature of Android permissions is that they allow for some decentralisation; applications can expose resources by themselves, so that there may be a permission mesh. Figure 4 clearly shows how complex the permission architecture for Android can be, especially on device ID 5 (Samsung S5). Because permission usage is widespread, information permeability is arguably high. Even for advanced Android users, it is practically impossible to determine whether a given permission set may lead to information leakage. As from Section 3.1.2, we identify a Class III (Confusion) architectural issue. In a scenario where applications can collude, preventing security issues is a very hard task.

Comparing different Android distributions also leads to interesting observations. For devices ID 1 and 2 in Table 1, more than 50% of all permissions are handled by the Google **Android** package. In these devices, because a single, coordinated developing vendor handles all permissions, it is expected that permission misuse will be lower on Google's vanilla devices when compared to "richer" permission usage devices like those from Samsung (IDs 4 and 5). On those devices, even with the similar Android version as the Nexus devices, an important growth on permission ecosystem exist: exposed and used resources are more than three times higher[1].

- Graph topologies are different: Samgung S5's is much more crowded and many clusters appear *addition* to those in Nexus 5 vanilla Android;
- there is a noticeable increase of `signature*` permissions, which can be used to insulate groups of applications: in the second graph, twenty-eight `signature*` permissions are created by a single package, namely `com.sec.enterprise`;
- there is no noticeable increase of `dangerous` resources but their usage is much larger.

In Figure 4, both diagrams (top-Samsung S5, bottom-Nexus 5) were drawn with the same scale and same "bubble-tree visualization algorithm", whose main characteristics include the ability to expose high degree nodes (such as the Android package) and to present strong stability in terms of graph changes [10] (similar graphs are drawn similarly). In those graphs, green circle nodes represent software packages, while square nodes represent assets (resources). A gray edge means that a given software package (green circle node) was given permission to *use* a given resource

[1]Of course, by installing applications, users can further crowd an already clumsy environment

Table 1: Comparative data for different android devices in factory state. VER, RSC, PKG, EXP, USE represent Android Versions, exposed resources (assets), software packages, exposing permissions edges, and permission usage edges, respectively. APR is the number of resources exposed by the Android package. PL is the average path length.

ID	Model	VER	RSC	PKG	EXP	USE	APR	PL
1	Google Nexus 4	4.4.4	388	110	381	1476	229	2.91
2	Google Nexus 5	4.4.2	392	104	373	1456	229	2.90
3	Sony Experia Z1	4.4.2	560	307	499	2864	222	3.04
4	Samsung Galaxy S4	4.3	879	379	767	4838	295	3.18
5	Samsung Galaxy S5	4.4.2	956	359	847	4871	339	3.27

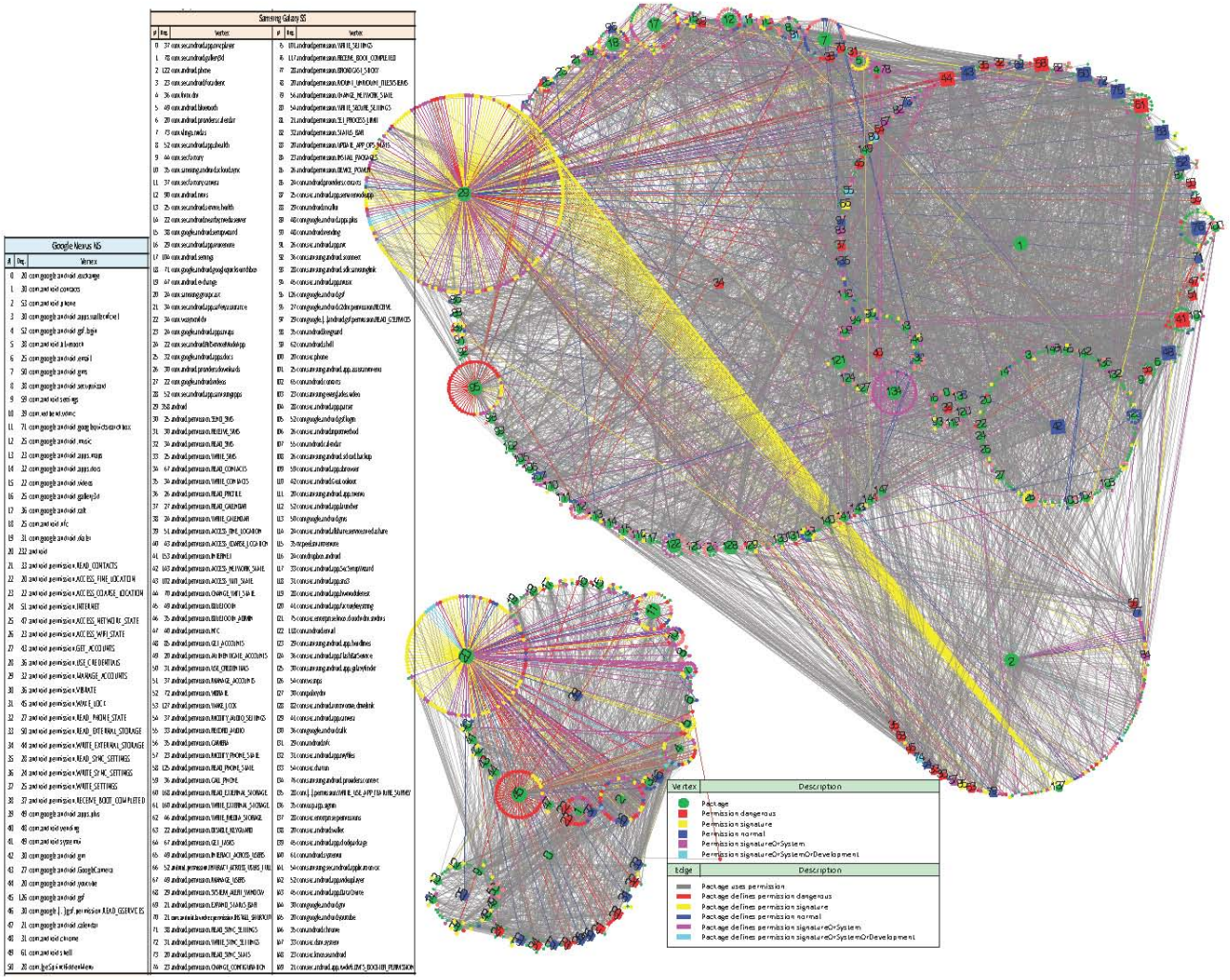

Figure 4: Package to Package communications allowed by permissions on a Samsung S5 (top) and Google Nexus 5 (bottom). Figures are zoomable and drawn at same scale. Vectorial figures and Tulip files: `http://goo.gl/JepwtC`.

(square node). The remaining edges represent permissions provided by a given package, according to the following color convention: `normal` = blue, `dangerous` = red, `signature` = yellow, `signatureOrSystem` = purple, `signatureOrSystemOrDevelopment` = cyan. Nodes are represented in three sizes according to their degree: small (up to 19), medium (from 20 to 69) and large (degree greater than 69).

It is easy to notice that the higher number of packages of device Samsung S5 (Table 1) translates into a much more populated graph. Although the increased number of vertices is not necessarily bad, Table 1 shows that there is no significant increase on the average path length, which in turn means that there are more parallel paths from (critical) assets to medium potentially controlled by adversaries, making Samsung S5 more prone to information leakage and collusion exploitation. In both graphs, the top right cluster (green dot) represents Android package and resources exposed by it (mainly System APIs). The closest cluster to the Android package is the Google play package, rich in "dangerous" permissions, making it a preferable target for attacks. For close-up details, the reader can zoom in on the electronic document.

So, by comparing the two graphs on Figure 4, the reader can observe some key features:

Further analysis of devices 4 and 5 showed that the additional resources and permissions are due mainly to additional packages and exposed resources, or, in other terms, there is more to protect and more being consumed. The addition of nodes and permissions could, in principle, improve security if properly implemented as gateways or proxies to resources, but average path lengths on Table 1 show only a 22% increase, even with the 200%+ rise in terms of resources. Also, the "proxy" hypothesis lacks support and requires further study, which is out of the scope of this work. It is noteworthy, however, that in terms of security, a proven (in practice and under the Fortuna model [8]) security best practice of "least privileges" is violated.

In order to better understand and compare the permissions *usage by type and by device*, we grouped permissions in different scenarios, as presented in Figure 5.

In the topmost subfigure, the number of permissions provided by different devices is classified with respect to the security level they present. Normal and dangerous permissions are roughly the same on all platforms, except on Samsung S4 and Samsung S5 (a 100% increase!), suggesting they both have more assets to protect due to the added software packages. When `signature*` permissions are considered, the growth is threefold. Although potentially positive, as a means of insulating different package groups, this increase is not accompanied by a reduction in normal or dangerous permissions, as one would expect, leading to the conclusion that there are indeed *more* paths to subvert the system.

In the mid-upper subfigure, the number of permissions requested for use by different devices is shown. In this figure, at least two aspects are noteworthy: (i) in Sony and especially in Samsung platforms, there is a very large incidence of permissions with no providers; although there is no obvious explanation for that phenomenon, we speculate that deprecation, provisioning for upgrades, software optionals or even customization capabilities are the main reasons, a hypothesis supported by others [7]; and (ii) the number of dangerous permissions being served at devices 4 and 5 are more than 3x larger than devices 1 and 2. With such a permeability, the odds of permissions leakage are much larger on Samsung devices.

On the bottom subfigures, permission usage is classified according to a matching criteria: in the middle-bottom subfigure, only permissions requested by packages by the same vendor are shown, while in the bottom subfigure the criteria is the complement. This latter subfigure is of greater interest as it shows a situation of (potential) conflict. There is a substantial number of signature permissions requested by packages with different vendor signatures which will be rejected by the Permissions Manager. This seems an odd situation, since a package will request a permission that it knows will be rejected. This needs investigation in a more detailed level: it could be related to permission misusage or packages with target SDK level prior to the introduction of the built-in permission on the SDK level of the running system ([6], p. 22). On the other hand, the number of permission requests of `signatureOrSystem` by packages signed by different certificate keys can be explained by their installation in the system partition as part of system image, otherwise they would be rejected as explained above for signature permissions. Also, an important number of `dangerous` permissions are requested by different vendors. This latter fact, however, can be explained by the large number of software packages added to the platform as a side-effect of OS customization.

Altogether, Table 1, graphs in Figure 4, and histograms in Figure 5 show the same, consistent scenario: mobile OS customizations deeply impact information permeability and the control capability over critical system and user assets. From a security perspective, Samsung S4 and Samsung S5 devices presented a considerably larger attack surface than the vanilla Android distribution of Nexus 4 and Nexus 5, prompting users to rethink their usage if security is a major concern and the added apps are not essential.

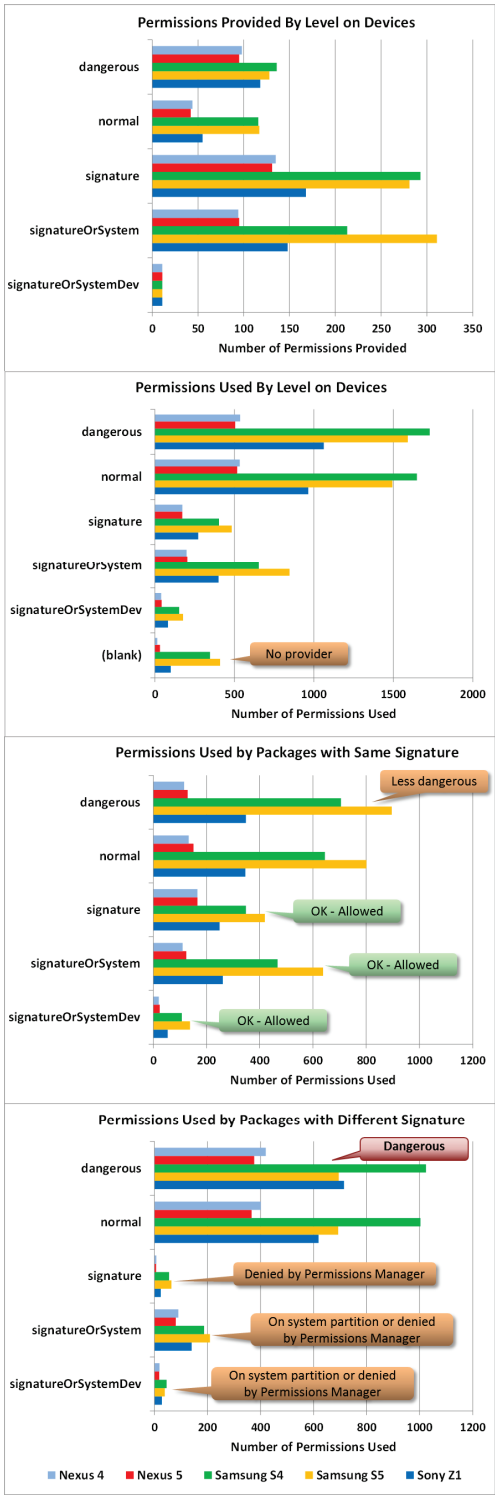

Figure 5: Permission types and usage scenarios. Nexus 4, Nexus 5, Sony Z1, Samsung S4, and Samsung S5 correspond to devices ID 1 to 5 from Table 1 respectively.

Nonetheless, even for the vanilla versions, the permission scheme implemented by the Android middleware is far from user-friendly and is prone to cause confusion. Also, it is important to note that if user's apps were to be considered, the complexity of the permission system would see further expansion and risks: while in the baseline we can assume that permission related problems are essentially accidental, in a deployed device, with potentially adversary applications, there is plenty and fertile ground for abuse. This may be one of the motivations for the permission domains being developed for the platform.

4. CONCLUSION AND FUTURE WORK

In this paper we showed how five different Android OS customizations can greatly vary in terms of security architecture due to modifications carried out by the manufacturers. By using different indicators and techniques, we showed that attack surface and information permeability are consistently worse on customized devices, as the addition of customization packages increases both the number of protected resources and the number of packages accessing those resources, violating both the "least privileges" and the "minimal trusted computing base" paradigms. At the very least it increases misuse and creates a more complex environment which can mask problems and is more difficult to analyse. Although the addition of new packages to OS customizations is not essentially bad, the need for compatibility with general apps prevents the removal of old, coarse-grained versions, when new permissions and APIs are added. Based on our findings, heavily customized Android distributions should be avoided by the security conscious user. Future work includes the investigation of how the security permission-related architecture is affected by typical user applications, and the differences for new Android versions (5.X) as these become available in more platforms. Also, a broader investigation on how graph visualization methods could help identify security issues is in our plans.

5. ADDITIONAL AUTHORS

6. REFERENCES

[1] D. L. R. A. Lineberry and T. Wyatt. These aren't the permissions you're looking for. BlackHat USA 2010. `http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf`, 2010.

[2] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[3] D. Auber, P. Mary, M. Mathiaut, J. Dubois, A. Lambert, D. Archambault, R. Bourqui, B. Pinaud, M. Delest, and G. MelanÃğon. Tulip : a scalable graph visualization framework. In S. B. Yahia and J.-M. Petit, editors, *EGC*, volume RNTI-E-19 of *Revue des Nouvelles Technologies de l'Information*, pages 623–624. CÃľpaduÃĺs-ÃL'ditions, 2010.

[4] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, Apr. 2011.

[5] W. L. Chapman, J. Rozenblit, and A. T. Bahill. System design is an NP-complete problem: Correspondence. *Syst. Eng.*, 4(3):222–229, Sept. 2001.

[6] N. Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, San Francisco, CA, USA, 1st edition, 2014.

[7] 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, pages 627–638, New York, NY, USA, 2011. ACM.

[8] R. Gallo, H. Kawakami, and R. Dahab. FORTUNA - A Probabilistic Framework for Early Design Stages of Hardware-Based Secure Systems. In *Proceedings of the 5th International Conference on Network and System Security (NSS 2011)*. IEEE, 2011.

[9] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*. The Internet Society, 2012.

[10] S. Grivet, D. Auber, J.-P. Domenger, and G. Melançon. Bubble Tree Drawing Algorithm. In *International Conference on Computer Vision and Graphics*, page to appear, Poland, 2004.

[11] I. A. W. Group. IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems. Technical report, IEEE, 2000.

[12] Ira Winkler. RSA Conference Keynote, 2013.

[13] J. Jürjens. UMLsec: Extending UML for secure systems development. In *UML 2002 - The Unified Modeling Language*, pages 412–425. Springer, 2002.

[14] C. Marforio, A. Francillon, and S. Čapkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical Report 724, ETH Zurich, April 2011.

[15] S. Moshtari, A. Sami, and M. Azimi. Using complexity metrics to improve software security. *Computer Fraud and Security*, 2013(5):8–17, 2013.

[16] N. Potlapally. Hardware security in practice: Challenges and opportunities. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 93–98. IEEE, 2011.

[17] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, 2011.

[18] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka. A small but non-negligible flaw in the android permission scheme. In *POLICY*, pages 107–110. IEEE Computer Society, 2010.

[19] V. Svajcer. Sophos Mobile Security Threat Report. `http://www.sophos.com/en-us/threat-center/mobile-security-threat-report.aspx`, February 2014.

[20] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha. Recommending refactorings to reverse software architecture erosion. In *16th European Conference on Software Maintenance and Reengineering (CSMR), Early Research Achievements Track*, pages 335–340, 2012.

[21] 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.

[22] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 409–423, Washington, DC, USA, 2014. IEEE Computer Society.