Exploring Reverse Engineering Symptoms in Android apps

Hugo Gonzalez
Faculty of CS
University of New Brunswick
Fredericton, NB, Canada
hugo.gonzalez@unb.ca

Andi A. Kadir
Faculty of CS
University of New Brunswick
Fredericton, NB, Canada
andi.fitriah@unb.ca

Natalia Stakhanova
Faculty of CS
University of New Brunswick
Fredericton, NB, Canada
natalia@unb.ca

Abdullah J. Alzahrani
Faculty of CS
University of New Brunswick
Fredericton, NB, Canada
a.alzahrani@unb.ca

Ali A. Ghorbani
Faculty of CS
University of New Brunswick
Fredericton, NB, Canada
ghorbani@unb.ca

ABSTRACT

The appearance of the Android platform and its popularity has resulted in a sharp rise in the number of reported vulnerabilities and consequently in the number of mobile threats. Leveraging openness of Android app markets and the lack of security testing, malware authors commonly plagiarize Android applications (e.g., through code reuse and repackaging) boosting the amount of malware on the markets and consequently the infection rate. In this study, we present AndroidSOO, a lightweight approach for the detection of repackaging symptoms on Android apps. In this work, we introduce and explore novel and easily extractable attribute called String Offset Order. Extractable from string identifiers list in the .dex file, the method is able to pinpoint symptoms of reverse engineered Android apps without the need for complex further analysis. We performed extensive evaluation of String Order metric to assess its capabilities on datasets made available by three recent studies: Android Malware Genome Project, DroidAnalytics and Drebin. We also performed a large-scale study of over 5,000 Android applications extracted from Google Play market and over 80 000 samples from Virus Total service.

Categories and Subject Descriptors

Security and Privacy [**Software and application security**]: Software reverse engineering

General Terms

Security and Privacy

Keywords

Android, malware

EuroSec'15 April 21-24 2015, Bordeaux, France

http://dx.doi.org/10.1145/2751323.2751330 .

1. INTRODUCTION

An appearance of a new Android platform and its popularity has resulted in a sharp rise in the number of reported vulnerabilities and consequently in the number of threats. According to Kaspersky's estimation, the number of mobile malware targeting the Android platform tripled in 2013, reaching 98.1% of all mobile malware [7]. This unprecedented growth in mobile malware is mostly attributed to the openness of Android application (aka app) markets and lack of security protection that allow malware authors to rapidly produce plagiarized apps reusing legitimate code and repackaging it with added malicious functionality.

Mobile app repackaging is often seen as an indicator of app's maliciousness. As such general detection malware apps and specifically, mobile app repackaging have been extensively studied in the last several years. RiskRanker [11], DroidRanger [25], Drebin [3], DroidScope [18], ViewDroid [19] are among general detection methods that are able to pinpoint malicious behavior either through a dynamic analysis of app's run-time behavior or through its static analysis. The majority of these studies employ complex sets of features (most of which are based on .dex file content), and thus offer good accuracy. Unfortunately, the complexity of these features' automatic extraction and the following pairwise apps' comparison often incurs higher computational cost making these methods prohibitively expensive for large-scale detection.

In this paper, we take an alternative strategy and propose to leverage formal specifications of Dalvik Executable Format that dictates the layout of .dex file. Based on these specifications, we explore symptoms of reverse engineering in .dex file composition through a novel and easily extractable attribute - *String Offset Order*. We leverage this feature in AndroidSOO, a lightweight approach for efficient triaging of Android applications. As opposed to the existing studies, the proposed method does not depend on the training period, a presence of the original code or a database of already analyzed apps. It also does not rely on a pair-wise comparison of Android apps which makes it suitable for scalable and efficient triage to pre-filter suspicious apps and improve efficiency of the following analysis.

We validate our approach with a set of 334 Android applications created with commonly used obfuscation tools. We show that AndroidSOO is an effective approach able

to pinpoint repackaging symptoms with 98% accuracy. We further evaluate the approach with datasets made available by the recent studies: Android Malware Genome Project, Drebin, Droid Analytics. Furthermore, we perform a large-scale study of over 5,000 Android applications extracted from GooglePlay market and over 80,000 samples from Virus Total.

2. RELATED WORK

With the recent burst of research interest in the area of Android device security, there have been a number studies focusing on mobile malware detection. These studies include detection of privacy violations during apps' runtime [20], unsafe permission usage [5, 16] and security policy violations [13].

There were a number of general studies offering methods for malicious app detection. These methods can be broadly divided into those focused on the detection prior to app installation (e.g., market analysis) and those that monitor app behavior directly on a mobile device. Among the studies in the first group are RiskRanker [11], DroidRanger [25], DroidScope [18] that dynamically monitor mobile apps behavior. Since these techniques are computationally expensive for a resource-constraint environment of a mobile platform, they are mostly intended for an offline detection. Among studies that focus on malware detection on a mobile device directly is Drebin [3].

With a recent wave of repackaged applications, a number of studies looked at the problem of mobile apps similarity. The majority of the existing methods look at the content of .dex files for app comparison. Juxtapp [12] evaluates code similarity based k-grams opcodes extracted from selected packages of the disassembled .dex file. The generated k-grams are hashed before they undergo a clustering procedure that groups similar apps together. Similarly, DroidMOSS [23] evaluates app similarity based of fuzzy hashes constructed based on a series of opcodes. DroidKin [10] detects the similarity between apps based on analysis of relations between apps. ViewDroid [19] leverages user interface-based birthmark for detecting app repackaging on the Android platform. Several methods were developed to fingerprint mobile apps to facilitate the detection (AnDarwin [8], DNADroid [9], [14]). All the studies require a pair-wise apps' analysis for detection of app similarity/repackaging. The proposed approach, AndroidSOO is able to pinpoint code repackaging without the presence of other apps, training datasets or pair-wise comparison.

3. BACKGROUND

Commonly an Android app is written in Java language and compiled to .class files, it is then converted into a .dex file that can be ran by Dalvik virtual machine on an Android platform. The apps are packaged in an .apk file containing the bytecode .dex file; AndroidManifest.xml file that describes the content of the package including the permissions information; and other resource files.

The .dex file is a binary container for the bytecode and the associated data. It includes a header containing meta-data about the executable followed by identifier lists that contain references to strings, types, prototypes, fields, methods and classes included in the app. The data section in the .dex file contains the information refereed in the references lists, including the actual bytecode and the strings data. The structure of .dex file is showed in Figure 1.

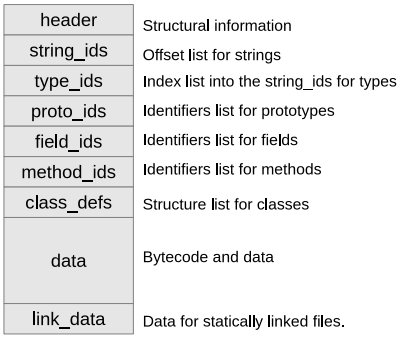

Figure 1: The layout of a .dex file

The development of an Android app requires a set of libraries and essential tools that are provided in Android Software Development Kit (SDK) environment. All these software packages are designed to support original app development. App repackaging however requires tools to reverse this process. One of the most widely used tools in this process is **apkTool**[1], a publicly available software for reverse engineering Android apk files. ApkTool became popular repackager widely used for both malicious and legitimate purposes. Studies such as ADAM [21] and PANDORA [15] have utilized the apkTool to disassemble the Dalvik bytecode to smali code to produce the repackaged versions of the apps. Several commercial tools based on apkTool also became available (e.g., APK Shell Decompiler [2], APKstudio[3]). Several alternative methods were explored in academia, such as dalvik-obfuscator tool and manual conversion [1].

Obfuscation and Optimization.

Obfuscation and optimization could be part of the process to produce apps. Some methods take an intermediate product as the .jar file to eliminate the code that are not in use, or minimize the redundancies. Other methods take an .apk file as an input, and produce a new one with the encrypted or obfuscated result.

Mobile malware obfuscation is gaining popularity in mobile devices. Potharaju et al. [14] have defined two obfuscation levels: basic Level-1 obfuscation that includes renaming and removal of unused identifiers (e.g., class, variable, methods), and the more advanced Level-2 obfuscation that includes insertion of junk code.

Until now though the most common obfuscation method applied in practice was Level-1 [14]. Obfuscation gained popularity partially due to the wide spread of repackaged applications, as it allows effective prevention of piggybacking malicious payload into original apps [24]. In this work, we experiment with ProGuard and DexGuard, tools that require source code, and APK Protect, HoseDex2Jar, Bangcle, and Dasho, tools that work directly with .apk file and produce a new obfuscated version of it.

ProGuard[4], the most well known of all the Android ob-

[1] `https://code.google.com/p/android-apktool/`
[2] `http://limelect.com/downloads/`
[3] `http://www.xda-developers.com/android/`
[4] `http://developer.android.com/tools/help/proguard.`

fuscators, is a free Java class file shrinker, optimizer, and obfuscator that detects and removes unused classes, fields, methods, and attributes and also optimizes bytecode and removes unused instructions. It is considered the most popular Level-1 obfuscator [14].

DexGuard[5] is the extended commercial version of ProGuard which focuses on code protection, with additional features like string and class encryption, obfuscation of the class and method names with non-ASCII characters. DexGuard-obfuscated samples were reported difficult to reverse engineer and can be considered as Level-2 obfuscation.

APK Protect is another advanced off-the-shelf obfuscation and protection method specific to Android executable files [4].

HoseDex2Jar[6] is a simple tool that prevents access to the source code of Android app using Dex2Jar tool.

Bangcle (in Chinese) or SecNeo (in English)[7] uses an online service to restructure the APK decrypting the app at runtime, obfuscating native libraries, and using libraries with stack protection enabled.

Dasho [8] provides protection for all Java and Android applications. Among obfuscation methods it employs are renaming of the methods' names, variables; encryption of strings in sensitive parts of the application; optimization with pruning which statically analyzes code to find the unused types, methods, and fields.

4. STRING OFFSET ORDER

One of the fundamental problems of detecting app' repackaging is designing features that are accurate and obfuscation-resilient in detection, while still scalable for automatic extraction and analysis.

While there are a number of features easily reachable from various components of .apk package, such as AndroidManifest.xml, a digital certificate authenticating an author, the resources and meta-data, none of them alone is sufficient for detecting repackaging. If employed these features are usually combined with various characteristics extracted through static and dynamic analysis of .dex file's content.

Since the majority of the previous studies acknowledged the importance of .dex file analysis in repackaging detection, we propose to focus on a .dex file's layout rather than its content. Dalvik Executable Format gives a formal specification of the layout of .dex executable file [2] guiding the development of Android apps. Since the development environments must follow the suggested guidelines, tracking discrepancies in implemented layouts can potentially pinpoint the use of optimization, obfuscation or tools employed in production of .apk file.

One example of such anomaly that was previously reported in [4] is the size of the header section. The formal specification explains that the header must have a '0x70' byte size. Different size however does not prevent Dalvik machine from correctly interpreting a file. As such HoseDex2Jar tool puts an encrypted version of the original code as part of the header leaving a distinct fingerprint on the file.

In this study we focus on the data section of the .dex file that contains all the strings, code, type, fields, classes specified in the reference lists. One component of .dex specification is string identifiers list that refers to all strings used, either as type descriptors or as objects referred to by source code. The specification states that this list must be free of duplicates and produced in alphabetical order based on the identifier. The example of such string identifier list is given in Figure 2. The sorted strings are grouped together in one part of data section. As a result the string offsets appear in ascending order, which can be seen in string identifiers list, i.e., a list of string offsets.

Since the requirement does not specify how this order can be enforced, the repackaging tools that offer easy access to the .smali code do not actually layout the strings (in the string portion of data section) in alphabetical order, but rather arrange their offsets so that the resulting list of strings appears to be in order. This is easily detectable as the corresponding offsets in string identifiers list do not follow any particular order and appear in disarray (see Figure 3). We leverage this observation for detecting signs of code repackaging and define the *String Offset Order* (SOO) as their indicator.

Table 1: Validation dataset

Origin	# of apps
Not-repackaged apps	
Original apps from individual sources	48
Obfuscated/optimized with	
Proguard	48
Bangcle	3
HosedDex2jar	3
DashO	3
ApkProtector	3
Apps enhanced with Mobile Ad library SDKs	5
Application generators	
PhoneGap	5
AdobeAir	5
Titanium	5
Bizness Apps	1
Andromo	1
App Inventor	2
iBuildApp	2
Como (Mobile by Conduit)	1
Dot42	14
DexGuard apps (GooglePlay)	5
DexGuard malware apps (VirusTotal)	2
Official apps from large open-source projects (optimized)	14
Repackaged apps	
akpTool	156
dalvik-obfuscator	5
manual repackaging	3

Table 2: Analysis results (validation dataset)

Apps	# of apps	Detected correctly	Missed
Apps without repackaging	170	165	5
Repackaged apps	164	161	3
Total	334	DR =98%, FPR = 2.9%	

5. VALIDATION

html

[5] http://www.saikoa.com/dexguard

[6] http://www.riis.com/hosedex2jar/

[7] http://www.secneo.com/

[8] https://www.preemptive.com/products/dasho

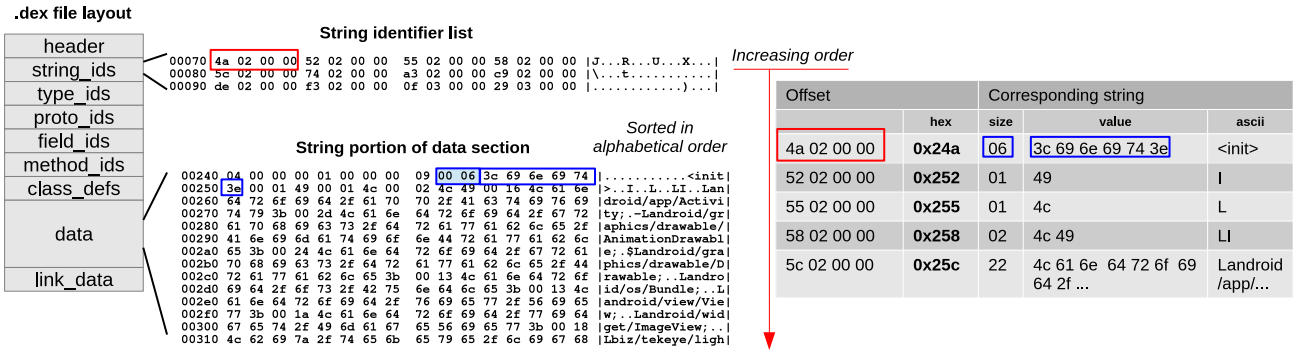

Figure 2: An example of string identifiers list with sorted string offsets.

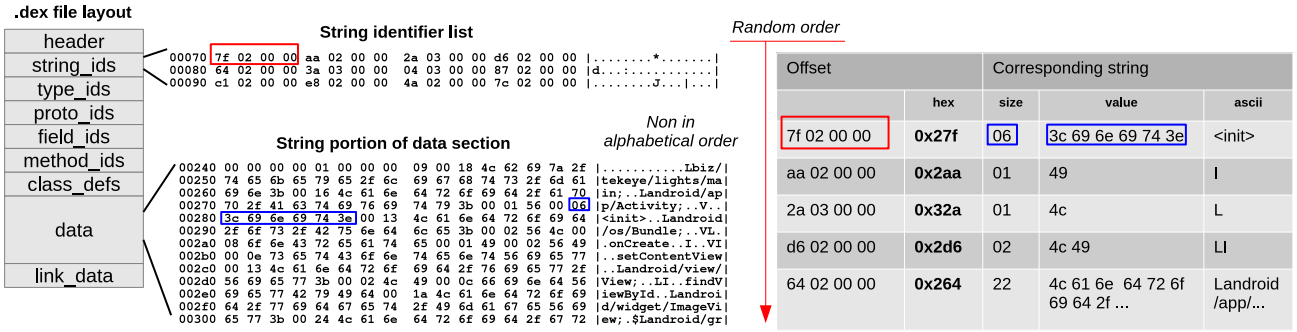

Figure 3: An example of string identifiers list implemented without sorting the string offsets.

To ensure the validity of the experiments and the effectiveness of the proposed approach, we conducted a validation study using a constructed dataset. For the validation dataset, source code for 48 unique Android apps was collected from public repositories, mostly associated with various personal open source projects. These original apps were employed to generate obfuscated and optimized versions using various tools. The listing of these tools and the number of the resulting apps are given in Table 1. Since the apps are commonly repackaged to include various ads, we compiled five original apps with various mobile ads SDKs. In addition to these apps, we selected 14 Android versions of widely used applications such as github, wordpress, and owncloud clients.

Although Android SDK (previously Eclipse ADT and ant, now Android Studio and grawdle) is the widely employed tool for apps generation, a study of GooglePlay market by Vuennot et al [17] revealed that over 15% of the available apps are built using various cross-platform frameworks and application generators. We employed these mostly commonly used application generators to produce 22 Android apps for the validation set. Finally, using the apktools framework we produced a repackaged version for each of these collected apps. One exception were 14 apps generated with the Dot42 framework[9], which uses C# source code to generate the apps, apktools could not succeed on repackaging these ones. We have also repackaged 5 apps using dalvik-obfuscator tool and 3 apps using manual process described [1].

The effectiveness of the string offset order as a feature for detecting symptoms of reverse engineering was tested on the final set of 334 apps. The details of these apps are given in Table 1.

Table 2 shows the results of analysis using AndroidSOO. As we expected the majority of apps (98%) produced directly from a source code retain the correct string offset order. Even obfuscated versions show compliance with specifications and produce both the string identifiers and the resulting offset list in order. The only exception to this are the manually repackaged apps. This not surprising though since after disassembling and modifying the code, the author still has to follow the traditional process creating .apk file with ADT tools which as our results show do retain the correct SOO.

Interestingly, the only generator that does not follow this guideline is Adobe Air that exhibits the same behaviour as apkTool reverse engineering tool. The use of Adobe Air for production of mobile app though can be easily confirmed through the employed libraries (specific to Adobe platform) or through an app name as apps produced by the Adobe Air SDK include some reference to Air.

All repackaged apps produced by apkTool and dalvik-obfuscator changed the original string identifiers list producing strings' offsets in a random order.

6. EXPERIMENTATION

To further evaluate the performance of the proposed approach we employed three datasets made available by the recent studies: Android Malware Genome Project[10] [24],

[9] `https://www.dot42.com/`

[10] We will refer to this dataset as Genome Project.

Table 3: Repackaging detection results

Dataset	Total	SOO random	SOO intact
Genome Project	1260	48.73%	51.27%
Debrin	5555	22.8%	76.72%
DroidAnalytics	2140	67.20%	32.80%
Googleplay	5,058	2.01%	97.99%
VirusTotal .dex	28,700	35.20%	64.80%
VirusTotal .apk	53,621	16.97%	83.03%
Total	96,334		

Table 4: Performance results

Dataset	Total (secs)	Unpack (secs)	Average total time per app (ms)
Genome Project	12.665	9.259	2
Debrin	62.530	49.644	2
DroidAnalytics	32.405	23.120	4
Googleplay	153.412	127.050	5
VirusTotal .dex	85.442	–	3
VirusTotal .apk	1994.977	1672.251	6

Drebin [3], DroidAnalytics [22], and we performed a large-scale study of top 5,058 Android applications retrieved from Google Play market and 53,621 .apk files and 28,700 .dex files retrieved from Virus Total repository. The total number of malware samples that have been scrutinized to evaluate the proposed approach is 96,334.

Repackaging detection results.

The results of analysis are given in Table 3. Among the analyzed sources, DroidAnalytics has the highest number of apps with SOO being random, i.e., likely repackaged apps (67%). This is followed by VirusTotal with 52% (2% created by Adobe Air) and Genome Project with almost 50% of samples. It should noted that these numbers do not imply that the remaining apps are legitimate, as malicious app may also be non-repackaged. In fact these numbers are similar to those reported by DroidKin [10] that found that 30% of malware in Genome Project have no similarity with other samples.

Table 5 shows the break down among top families of the suspicious apps found in Genome Project dataset. As expected, the families known for repackaging consistently show larger percentages of samples with random SOO. For example, in case of Geinimi family with 55 out of 61 samples (Genome Project dataset) were created from other apps.

Further analysis of data from Genome Project and its comparison to the results obtained in the original study reveal interesting disagreement. Several families that in the original study were not detected as repackaged (e.g., KMin, Plankton, YZHC). AndroidSOO though found them to contain random string offset order which indicate that these samples are most likely repackaged. The original study based repackaging detection on the difference in the payload between a sample on hand and the one retrieved from official GooglePlay market. Several studies since then indicated that insertion of malicious payload is just one of the reasons for repackaging [23]. This example emphasizes the benefits of viewing AndroidSOO as a complementary approach to analysis of suspicious apps.

The lowest number of apps with random SOO was found on set retrieved from Google Play, i.e., 2% (102 apps). The manual inspection revealed that 45 of them were created by Adobe Air platform and in fact are original apps, and the remaining 57 apps (1.12%) are repackaged apps. Among these 57, 30 apps were marked by VirusTotal as adware. This is not surprising as repackaging with the purpose of delivering adds is a common practice. The portion of repackaged apps found by AndroidSOO on the Google Play market (1.12%) is consistent with the findings reported earlier [6].

Performance evaluation.

Since the AndroidSOO does not require a pairwise comparison of apps, detection of repackaging can be performed for both stand alone apps and large-scale datasets. In both cases, the overhead introduced by the approach is critical in understanding its practicality. We perform a set of experiments to access AndroidSOO performance. In this experiments we employ an Intel Core i7 @3.24Ghz and 16 GB of RAM. The total processing time of an application consists of two parts: time necessary to extract .dex file from an .apk file and time to identify the correctness of SOO. As Table 4 shows on average identifying whether a given app was produced with apkTools only takes 2-6 ms.

Table 5: Genome Project Dataset (selected families)

Family	Samples	SOO Intact	SOO Random
Geinimi	61	6	55
GoldDream	47	13	34
Pjapps	45	1	44
DroidKungFu1	30	2	28
DroidKungFu2	30	13	17
AnserverBot	26	1	25
ADRD	19	10	9
YZHC	16	15	1
DroidDream	15	1	14
Plankton	11	2	9
KMin	9	0	9

7. DISCUSSION

Our evaluation results show that AndroidSOO can effectively pinpoint suspicious apps with symptoms of repackaging. AndroidSOO relies on a specific ordering property of the string constant addresses and effectively detects signs of apktools and dalvik-obfuscator. At this moment these are the most commonly used tools for repackaging apps. Modification of these tools to concur with specifications will potentially remove this property from newly repackaged apps. This however will not affect previously repackaged apps that will remain in the wild and thus be still visible to AndroidSOO. Similarly, development of new reverse engineering and repackaging software that would follow specifications would prevent AndroidSOO from detecting these new repackagers. As such AndroidSOO can be leveraged in combination with other features to improve the accuracy of existing detection approaches or given its scalable nature, as a triage filter to reduce the number of files that need to be analyzed further.

As our evaluation showed, AndroidSOO does not miss any apps repackaged with apktools and dalvik-obfuscator. However, it does not see the apps that underwent a manual repackaging process. At this point manual repackaging is a proof-of-concept process that requires significant amount of time and knowledge, thus we do not see it as a viable alternative to automatic repackaging in the near future.

Our experiments also indicate that AndroidSOO can ef-

fectively spot repackaging symptoms even in the presence of various obfuscation and optimization methods. While the majority of obfuscation techniques employed in our study can be described as Level-1 (simplistic) obfuscation, these are the most common methods employed in the app markets these days.

8. CONCLUSION

In this work we presented a lightweight approach for detecting symptoms of repackaging in Android apps using a novel feature - the String Offset Order. As opposed to the existing methods, SOO offers a robust and fast mechanism to differentiate between original and repackaged code even in the presence of obfuscation. Our experiments with commonly applied obfuscation techniques reveal the approach's resiliency and its ability to accurately pinpoint repackaging symptoms in apps generated with apkTool and dalvik-obfuscator. As opposed to the existing techniques AndroidSOO can incrementally process apps without training period, predefined detection patterns or existence of external repositories. Our results also demonstrate that the proposed approach is highly scalable as the analysis of an app on average takes only a few milliseconds. We see practical value of AndroidSOO as a complementary approach in more comprehensive analysis of malware.

9. REFERENCES

[1] Extending Existing Android Applications. Available: http://blog.inyourbits.com/2012/11/extending-existing-android-applications.html, 2012.

[2] Android. Dalvik Executable Format. http://source.android.com/devices/tech/dalvik/dex-format.html.

[3] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the 21th Annual NDSS*, 2014.

[4] R. N. Axelle Apvrille. Obfuscation in android malware and how to fight back. volume July 2014 of *Covering the global threat landscape*, pages 1–10. Virus Bulletin Ltd, 2014.

[5] 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 CCS*, pages 73–84, New York, NY, USA, 2010. ACM.

[6] L. Botezatu. 1.2 Percent of Google Play Store is Thief-Ware, Study Shows. http://goo.gl/Mx1X3j.

[7] V. Chebyshev and R. Unuchek. Mobile malware evolution: 2013, July 2014.

[8] J. Crussell, C. Gibler, and H. Chen. Scalable semantics-based detection of similar android applications. In *18th ESORICS*, Egham, U.K.

[9] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *17th ESORIC*, volume 7459, pages 37–54. Springer, 2012.

[10] H. Gonzalez, N. Stakhanova, and A. Ghorbani. Droidkin: Lightweight detection of android apps similarity. In *Proceedings of the 10th SECURECOMM*, 2014.

[11] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *The 10th MobiSys*, pages 281–294, 2012.

[12] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Proceedings of the 9th DIMVA'12*, pages 62–81, Berlin, Heidelberg, 2013. Springer-Verlag.

[13] M. Nauman, S. Khan, and X. Zhang. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ASIACCS '10*, pages 328–332, New York, NY, USA, 2010. ACM.

[14] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *Proceedings of the 4th ESSoS'12*, pages 106–120, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC*, pages 329–334. ACM, 2013.

[16] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: A perspective combining risks and benefits. In *Proceedings of ACM SACMAT*, pages 13–22, New York, NY, USA, 2012. ACM.

[17] N. Viennot, E. Garcia, and J. Nieh. A measurement study of Google Play. In *Proceedings of SIGMETRICS 2014*, page 13, 2014.

[18] L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[19] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *7th ACM Conference on WiSec 2014*, Oxford, United Kingdom, 2014.

[20] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC*, CCS '13, pages 611–622, New York, NY, USA, 2013. ACM.

[21] M. Zheng, P. P. Lee, and J. C. Lui. Adam: an automatic and extensible platform to stress test android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 82–101. Springer, 2013.

[22] M. Zheng, M. Sun, and J. Lui. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *2013 12th IEEE International Conference on TrustCom*, pages 163–171. IEEE, 2013.

[23] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the Second ACM CODASPY '12*, pages 317–326, New York, NY, USA, 2012. ACM.

[24] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (SP)*, pages 95–109. IEEE, 2012.

[25] 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 *19th Annual NDSS*, 2012.