On Malware Leveraging the Android Accessibility Framework

Joshua Kraunelis[1], Yinjie Chen[1] Zhen Ling[2], Xinwen Fu[1], and Wei Zhao[3]

[1] Computer Science Department, University of Massachusetts Lowell, {jkraunel,ychen1,xinwenfu}@cs.uml.edu,
[2] Southeast University, China, zhen_ling@seu.edu.cn,
[3] University of Macau, China, weizhao@umac.mo

Abstract. The number of Android malware has been increasing dramatically in recent years. Android malware can violate users' security, privacy and damage their economic situation. Study of new malware will allow us to better understand the threat and design effective anti-malware strategies. In this paper, we introduce a new type of malware exploiting Android's accessibility framework and describe a condition which allows malicious payloads to usurp control of the screen, steal user credentials and compromise user privacy and security. We implement a proof of concept malware to demonstrate such vulnerabilities and present experimental findings on the success rates of this attack. We show that 100% of application launches can be detected using this malware, and 100% of the time a malicious Activity can gain control of the screen. Our major contribution is two-fold. First, we are the first to discover the category of new Android malware manipulating Android's accessibility framework. Second, our study finds new types of attacks and complements the categorization of Android malware by Zhou and Jiang [22]. This prompts the community to re-think categorization of malware for categorizing existing attacks as well as predicting new attacks.

Key words: Android, Malware, Attack

1 Introduction

The number of mobile malware samples has increased enormously over the past two years while mobile devices have become a ubiquitous tool in our daily life. In March 2013, Juniper Networks [8] reported their Mobile Threat Center had discovered over 276 thousand malware samples, a 614 percent increase over 2012. With 92 percent of mobile malware being Android malware, analyzing and categorizing these malware are important steps toward predicting new attacks.

In this paper, we explore a security and privacy risk hiding within the Android accessibility framework. The Android accessibility framework is developed to assist physically impaired users. Android developers can utilize accessibility applications programming interface (API) methods to provide customized accessibility services in their own applications. However, the accessibility service has

access to critical sensitive information, including information about applications that are currently running and account information. Attackers could utilize such a vulnerability to conduct various types of attacks.

Our major contributions are summarized as follows:

- We are the first to identify malware leveraging Android's accessibility framework. The impact of this attack is severe, given that it can be used to activate various malicious payloads. For example, the payload can be various masquerade attacks, emulating Email, Facebook and other popular apps to steal user credentials and cause other damage. We test our proof-of-concept malware against multiple anti-malware applications and none are able to detect it.
- We identify a potential logic error in the way the Android user interface framework manages the order of application launches. When two applications are launched nearly simultaneously, both launch requests consider the first request to be processed as the next application to be launched, preventing the second launch from occurring. This can lead to a variety of attacks, such as denial of service and masquerade attacks, as we discuss in Section 3.3.
- Our study complements the categorization of Android malware by Zhou and Jiang [22], who present a systematic characterization of existing Android malware by their strategies of installation, activation, payload and permission use. We identify new events for activation and our attack shows that the payload can be other attacks, such as the masquerade attack demonstrated in this paper. This prompts the community to ask: are there better ways of classifying Android and other mobile malware? We hope this classification not only allows us a systematic and holistic study of existing malware, but also provides assistance to predict new attacks and design countermeasures.
- We also discuss possible countermeasures against the security risk from the Android accessibility framework.

The rest of the paper is organized as follows. Section 2 introduces Android accessibility service. Section 3 introduces our attack. We evaluate our proposed attack in Section 4, and discuss countermeasures in Section 5. Section 6 introduces other related work, and Section 7 concludes this paper.

2 Android Accessibility Service

An Android application must contain one or more of the following four components: Activity, Service, Broadcast Receiver, and Content Provider [5]. Activities represent tasks involving user interaction and can display drawable components, such as widgets, to the screen. The operating system ensures that only a single `Activity` for any application is displayed at once, i.e. only one application may be in the foreground at one time. Services are used for long running tasks that do not require a user interface. Unlike Activities, Services may run in the background and therefore multiple Services from different applications may run simultaneously. Broadcast Receivers receive messages, in the form of data constructs called Intents, from the Android system or user applications. An appli-

cation must register for those Intents which it is interested in receiving. The registration of certain Intents may require permission to be granted by the user at install time. Content Providers enable data sharing between applications. The exploit presented in this paper will focus on Activities and Services.

The Android operating system contains an accessibility framework [1] for enhancing the experience of users who have visual or other impairments. Typical accessibility enhancements include enlargement and text-to-speech conversion of on-screen elements, high contrast color schemes, and haptic feedback. Android provides a Java API to its accessibility framework so that developers can integrate accessibility functionality into their applications. All drawable elements derive from a common ancestor, the View class, which contains built-in calls to Accessibility API methods. Thus, most user interface widgets in the Android framework make their accessibility information available by default, though developers are encouraged to provide additional information in the `android:contentDescription` XML layout attribute. Accessibility information such as the UI event type, class name of the object in which the event occurred or originated, and string value representing some associated data can be populated into an `AccessibilityEvent` object and dispatched to enabled `AccessibilityService`s via the appropriate method call [9]. There are twenty two `AccessibilityEvent` types.

Additionally, as of Android 1.6, developers may create custom accessibility services by extending the `AccessibilityService` class [2]. Descendants of `AccessibilityService` must override the on`AccessibilityEvent` method, which gets called each time an Accessibility Event occurs and is passed the populated `AccessibilityEvent` object. A custom `AccessibilityService` may then use the information contained in the `AccessibilityEvent` or optionally query the window content for the contextual data needed to perform its function. Naturally, the receipt of an event and its associated data and the ability to query window content present a security risk. For example, every time a user inputs text into an `EditText` widget, a custom `AccessibilityService` will receive an `AccessibilityEvent` with type `TYPE_VIEW_TEXT_CHANGED` that contains the text that the user input. To mitigate this risk, the Android system requires that `AccessibilityService`s be enabled manually by the user and displays a dialog window alerting the user to the risk.

3 Malware Exploiting Accessibility Service

In this section, we first give an overview of the malware that exploits the Android accessibility framework. We then address major challenges for such malware to work, including detection of the launch of a victim app and race condition between the victim app and malware.

3.1 Overview

We now introduce the novel malware's installation, activation, malicious payloads and permission uses.

Installation: The new Android malware can provide regular accessibility service as it claims and conduct attacks silently. Therefore, impaired users and users who prefer big font text may be interested in such malware and install it onto their phones. This installation strategy is *installation.others.3rdgroup* in [22], referring to apps that intentionally include malicious functionality. For brevity, we denote installation.others.3rdgroup as *trojan*, "a program made to appear benign that serves some malicious purpose" according to the taxonomy in [16], although this definition of trojan may be controversial.

Permission Uses: During installation, the malware requests the `BIND_ACCESSIBILITY_SERVICE` permission. After installation, users must enable the `AccessibilityService` in Android's Accessibility Settings menu. Since a legitimate accessibility service also requests such permission and requires enabling, users may not suspect the motivation of our malware. Of course, other permissions are required if the malicious payload requires them.

Activation: After the installation, our malware can derive a list of all installed applications in that phone. This can be achieved via many sources, such as the Package Manager. Based on which applications are installed, our malicious application could download various payloads and use them to launch different attacks. Each time a user launches an application from the home screen or the application drawer, our malicious accessibility service is activated, and a malicious payload which targets that application is also activated.

Here, we make one complement to the events which could be used by Android malware, introduced in [22]. As we introduced in Section 2, the `AccessibilityEvent` is one critical event, which carries sufficient information. There are twenty-two types of `AccessibilityEvent`, and these types of events can trigger various types of malicious payloads.

Malicious Payload: Since we know what applications are installed, we can use different malicious payloads to launch different attacks. For example, the default Email application source code is freely available from the Android Open Source Project [4]. In this case, our malicious payload could masquerade as the Email application. Therefore, when a user launches the Email application, a fake login window will display and prompt the user to input account name and password. Such account information is then collected and sent over an encrypted channel to a remote server.

To implement the masquerade attack in this example, we extract the `AccountSetupBasics Activity` and its corresponding resources from the Email application, modify it, and package it into a fake application as a malicious payload. `AccountSetupBasics` is displayed when the Email application is launched and no previous email account has been setup. It was chosen because of its simple design, the popularity of the Email application, and having the fields required to demonstrate the attack. The `AccountSetupBasics` layout consists of two `EditText` widgets for username and password input, and two Buttons: one for activating

the Manual Setup feature and the other for navigating to the next step in the setup process. The counterfeit `AccountSetupBasics Activity` included in our `AccessibilityService` application duplicates all of the graphical user interface elements of the victim `Activity`, but the email account setup functionality of the victim `Activity` has been removed. Though we have chosen to imitate the Email application's `AccountSetupBasics Activity` for this experiment, any `Activity` may be used in the attack. In this example, our `AccessibilityService` provides no additional accessibility features, but attackers could easily provide such features to increase the guile of this attack.

Therefore, we make one complement to the categorization of malicious payload in [22]. Android malware can perform blended attacks. The payload could contain a vector of payloads that perform different functions. *The payload itself can be other malware or attacks.* In the example above, our malware, a trojan app, contains a fake Email `Activity` that performs a masquerade attack.

For our trojan to work, there are two more details we have to address:

- How can the trojan detect the victim app launch? Although the malicious accessibility service is able to receive events related to `Activity` launch, it still needs to distinguish which app is generating these events so that it can launch the corresponding fake app and perform the masquerade attack.
- How can the trojan display itself to the user while the victim app is hidden in the background? When the user touches an app, this app will be launched. Which app, our trojan or the victim app, will be displayed? How can our trojan win the race condition?

We address these two issues below.

3.2 Detecting Application Launch

A crucial piece of our masquerade attack is the ability to detect the launch of a victim application. There is no public API to allow a user application to be notified when another application is launched. By registering to receive `AccessibilityEvent` callbacks in the application manifest, our custom `AccessibilityService` is guaranteed to be notified when an application is launched by the user. The notification comes in the form of an `AccessibilityEvent` object that is delivered as an argument to our `AccessibilityService`'s `onAccessibilityEvent` method. The Launcher application, which is responsible for displaying icons and widgets on the home screen and maintaining the app drawer, populates the `AccessibilityEvent` when the Email application icon is clicked by the user.

From the information in the `AccessibilityEvent`, we can determine that the user clicked an icon in the Launcher, because the event type is `TYPE_VIEW_CLICKED` and the originating package name is com.android.launcher. We're able to identify which icon was clicked based on the Text field of the `AccessibilityEvent`. In the case of attacking Email, the Text field is a single element list containing the string "Email". Once the app launched by the user has been detected, the next step in the attack is to launch the malicious `Activity` instead

of the user desired `Activity`. To do this, an Intent that specifies the malicious `Activity` to be launched is created and passed to the `startActivity` method of our `AccessibilityService`.

3.3 Racing to the Top

The launch of the malicious `Activity` from our `AccessibilityService` does not prevent the Launcher app from also calling `startActivity` to start the legitimate `Activity`. Both Activities are created and dispatched to the `ActivityManager` to be displayed. As previously mentioned, only a single `Activity` may be displayed in the foreground at one time. This is a source of contention for our malicious `Activity`, which we want to be displayed instead of the victim `Activity` and without any suspicious screen flicker or transition animation that may alert the user to the presence of malware. Since the malicious `AccessibilityService` receives `AccessibilityEvent` and is able to detect launch of victim app, it has a chance to launch the malicious `Activity` before the victim `Activity` is launched.

There exists a source of contention for our malicious `Activity`. An attacker wants the malicious `Activity` to be displayed instead of the victim `Activity` without any suspicious screen flash, flicker, or transition animation that may alert the user to the presence of malware. To achieve this goal, the timing of launching malicious `Activity` should be carefully adjusted so that the malicious `Activity` is processed soon after the victim `Activity`. Therefore, the problem is how to derive an optimal delay for the malicious `Activity`. We present our analysis below.

We find that different delay of the malicious `Activity` produces four different statuses of the phone screen. Before introduce the four statuses, please note that when the malicious `Activity` is processed, a fake interface is created and displayed. Please also note that when the victim `Activity` is processed, a victim interface is created and displayed. Depending on the timing of processing each activity, there are four scenarios. (I) The malicious `Activity` is processed before the victim `Activity`. The fake interface will be displayed first, and then replaced by the victim interface. In this scenario, we can observe the victim interface showing up with a flash, and the status of phone screen is defined to be Ω_1. (II) The malicious `Activity` is processed before the victim `Activity`, but these two activities are processed nearly simultaneously. In this scenario, only the victim interface is displayed without a flash. The status of phone screen is defined to be Ω_2. (III) The malicious `Activity` is processed after the victim `Activity`, but these two activities are processed nearly simultaneously. In this scenario, only the fake interface is displayed without a flash. The status of phone screen is defined to be Ω_3. (IV) The malicious `Activity` is processed after the victim `Activity`. The victim interface will be displayed first, and then replaced by the fake interface. In this scenario, we can observe the fake interface showing up with a flash, and the status of phone screen is defined to be Ω_4. These statuses are listed in Table 1.

It is obvious that an attacker will want the phone screen status to be Ω_3. Therefore, we need to derive an optimal delay time for the malicious `Activity`.

Table 1. List of Phone Screen Statuses

	With flash	Without flash
Victim interface shows up	Ω_1	Ω_2
Fake interface shows up	Ω_4	Ω_3

However, for every victim app, the optimal delay time is different. To derive the optimal delay for a specific app, we can enumerate possible delay d to start the malicious `Activity` with delay d. The delay values producing the highest probability that status Ω_3 occur are the optimal ones.

Using the strategy above, we can derive an optimal delay for every victim app running on different phone models with different Android versions, and keep a record of these optimal delays in a remote server which provides all the malicious payloads we developed. When our malicious service tries to download a malicious payload from that server, the malicious service also sends a request to get an optimal delay to trigger that payload. Therefore, every time a victim app is launched, a fake interface is always shown without a flash.

4 Evaluation

To prove malware exploiting Android's accessibility framework is a new breed, we test our proof-of-concept implementation against four free Android mobile security applications from the Google Play Market: AVG AntiVirus, Norton Mobile Security, Lookout Mobile Security, and Trend Micro Mobile Security. None of the four detected any suspicious behavior or raised a flag during virus scanning while the `AccessibilityService` was enabled.

In the following, we provide the experimental results for detecting application launch and winning the race condition. The malicious `AccessibilityService` was tested on an Android emulator running Android 4.2, an HTC Nexus One running Android 2.3.6, and a Samsung Galaxy S2 running CyanogenMod 9.

4.1 Detecting Application Launch

In the Android versions we tested, application launch detection can be done for any application that defines a Launcher `Activity`. This is true because the objects that represent shortcut icons on the home screen and the app drawer are descendants of the `TextView` class, and do not override the behavior of the `onPopulateAccessibilityEvent` method. The `TextView`'s `onPopulateAccessibilityEvent` method adds the character string contained in its text field to the text list of character strings in the `AccessibilityEvent`. The `BubbleTextView` class used to represent shortcut icons will always store the title of the shortcut that is displayed in the home screen/app drawer in its text field. Therefore, the launch detection simply must match this title in order to detect application launch. Application shortcuts that belong to the hotseat, the horizontal space at the bottom of the default home screen that stays "docked" when navigating to

alternate home screens, do not display a title but all shortcut icons in the application drawer do. Although the hotseat applications do not display a title, the `AccessibilityEvent` that is dispatched on click does contain the title, however this is not the case in CyanogenMod 9.

We tested the launch detection capability on the HTC Nexus One for the following six Android applications: Messaging, Email, LinkedIn, Facebook, Bank of America, and Browser. For each application, a shortcut icon was created on the home screen. The launch detection was successful for all six applications.

4.2 Winning Race Condition

During the launch detection testing, we noticed that the malicious `Activity` was not displayed instead of the victim `Activity` 100% of the time, especially when the victim application was being launched for the first time since system boot. To test this, we performed two separate experiments. In the first experiment, we ensured the application we were launching was not running by pressing the *Force Stop* and *Clear Data* buttons under the corresponding *Settings -> Manage Applications -> All menu* for that application. These two operations effectively force the application to be reloaded from its initial state, as if the system had just booted. We then returned to the home screen, launched the application normally, and recorded which `Activity`, malicious or victim, was displayed. In the second experiment, we followed the same procedure, but instead of force stopping and clearing the data, we made sure that the victim application had previously been launched before relaunching from the home screen.

The two experiments were performed repeatedly for each of the aforementioned applications, and we observed the impact of the race condition discussed in Section 3.3. However, by setting the malicious `Activity` to sleep for specific time period before it is started, we can guarantee that the malicious `Activity` displays without a flash.

Now we evaluate our strategy of optimizing delay as discussed in Section 3.3. We choose Browser app and conduct two groups of tests on the HTC Nexus One running Android 2.3.6.

In the first group of tests, we test the Browser app with reloading. The procedure for our test is as follows. (I) We select a set of different delays, and choose each of these delays to launch our malicious `Activity`. (II) We press the *Force Stop* and *Clear Data* buttons under the corresponding *Settings -> Manage Applications -> All menu* for the Browser app. These two operations effectively force the application to be reloaded from its initial state when it is launched. (III) We launch the Browser app from home screen, and count the number of screen statuses we observe. (IV) For each delay, we repeat step (II) and (III) many times and calculate the rates of occurrences of those four statuses defined in Table 1. We present our experimental results in Table 2.

From Table 2, we make the following observations. (I) When delay is 0ms, the victim interface shows up without a flash (Status Ω_2) 100% of the time. (II) When delay increases from 1ms to 740ms, the fake interface shows up without a flash (Status Ω_3) 100% of the time. (III) When delay increases from 750ms to

Delay (ms)	Ω_1	Ω_2	Ω_3	Ω_4
0	0%	100%	0%	0%
1 - 740	0%	0%	100%	0%
750	0%	0%	90%	10%
760	0%	0%	70%	30%
770	0%	0%	50%	50%
780	0%	0%	30%	70%
790	0%	0%	30%	70%
800	0%	0%	20%	80%
810	0%	0%	20%	80%
820	0%	0%	20%	80%
830	0%	0%	10%	90%
840	0%	0%	0%	100%

Table 2. Status with Reloading

Delay (ms)	Ω_1	Ω_2	Ω_3	Ω_4
0	0%	20%	80%	0%
1-60	0%	0%	100%	0%
70	0%	0%	90%	10%
80	0%	0%	100%	0%
85	0%	0%	80%	20%
90	0%	0%	90%	10%
95	0%	0%	80%	20%
100	0%	0%	80%	20%
105	0%	0%	90%	10%
110	0%	0%	50%	50%
115	0%	0%	10%	90%
120	0%	0%	20%	80%
125	0%	0%	10%	90%
130	0%	0%	20%	80%
135 - 145	0%	0%	0%	100%

Table 3. Status without Reloading

840ms, the rate of (Status Ω_3) decreases to 0%, while the rate of fake interface showing up with a flash (Status Ω_4) increases to 100%. Therefore, the optimal range of delay is between 1ms and 740ms.

In the second group of tests, we test the Browser app without reloading. The procedure for our test is as follows. (I) We set the delay of our malicious `Activity` to different values. (II) We launch the Browser app from the home screen, and record the status of the screen we observe. (III) For every setting of delay, we repeat step (II) a large amount of times, and calculate the rates of those four statuses defined in Table 1. We present our experimental results in Table 3.

From Table 3, we make the following observations. (I) When delay is 0ms, 20% of the time the victim interface shows up without a flash (Status Ω_2). 80% of the time the fake interface shows up without a flash (Status Ω_3). (II) When delay increases from 1ms to 60ms, the fake interface shows up 100% of the time without a flash. (III) When delay increases from 70ms to 105ms, the fake interface shows up 80% to 90% of the time without a flash, and the fake interface shows up 10% to 20% of the time with a flash (Status Ω_4). (IV) When delay increases from 110ms to 145ms, the probability of Status Ω_3 decreases to 0%, while the probability of Status Ω_4 increases to 100%. Therefore, we derive a range of optimal delay, which is between 1ms and 60ms.

In the first group of experiments, the range of optimal delay is between 1ms and 740ms. In the second group of experiments, the range of optimal delay is between 1ms and 60ms. Combining the experimental results from these two groups, the optimal delay for the malicious `Activity` is between 1ms and 60ms.

5 Discussion

The attacks mentioned above are not foolproof. There are certain safeguards built into Android devices to thwart a full device takeover by a malicious user. Recovery Mode is one such safeguard that allows the user to install a clean OS, wiping any malicious apps from the data partition in the process. An experienced user could use the Android Debug Bridge (ADB) to obtain a command shell into the device and launch applications from the command line *am* tool or even locate and uninstall the malicious applications. Applications launched from the *am* tool do not invoke the Launcher application, therefore our technique is unable to detect this.

As a countermeasure to the malicious `Activity` payload, the logic which reorders the `Activity` history stack and determines the next `Activity` to be displayed should be analyzed and corrected. The two main Android files containing said logic are frameworks/base/services/java/com/android/server/am/ActivityManagerService.java and frameworks/base/services/java/com/android/server/am/ActivityStack.java. Ensuring that the next `Activity` to be displayed is indeed the `Activity` that was requested by `startActivity` could prevent the malicious `Activity` from being displayed instead of the legitimate `Activity`. However, this fix does not hinder launch detection and the attacker could simply delay the start of the malicious `Activity` for some short time, ensuring that the malicious `Activity` is displayed after all. The difference here is that there may be some obvious transition animation, if the developer of the legitimate app has not disabled it, from the legitimate `Activity` to the malicious `Activity`.

6 Related Work

In 2009, Schmidt *et al.* [20] made a survey of mobile malware and found that most malwares targets Symbian OS. Since Android OS was getting attention at that point and the authors investigated possibilities of malwares on Android, they explored "social engineering"-based Android malware, where the malicious functionality is hidden in a seemingly benign host app. Felt *et al.* [15] classify threats from third-party smartphone applications into *malware*, *grayware*, and *personal spyware*. Becher *et al.* [11] examine mechanisms securing sophisticated mobile devices in 2011. Threats were classified into four classes: *hardware centric*, *device independent*, *software centric*, and *user layer attacks* for the purpose of eavesdropping, availability attacks, privacy attacks and impersonation attacks. Existing security mechanisms are enumerated for various attacks.

Enck *et al.* [14] implemented *ded*, a Dalvik decompiler. *ded* transfers *.dex* file into Java source code. It was then used for analyzing security of 1,100 popular free Android applications. Zheng *et al.* [21] developed ADAM, an automated system for evaluating the detection of Android malware. ADAM uses repackaging and code obfuscation to generate different variants of a malware. Rastogi, Chen and Jiang [19] made similar effort to test state-of-the-art Andoroid commercial mobile anti-malware products for detecting transformed malware. Such

transformation techniques include polymorphism (where transformed code is still similar to the original code) and metamorphism (where transformed code is totally different from the original code, but with similar malware functionality). Zhou *et al.* [23] developed a system called DroidRanger, evaluating the health of Android markets, including the official Android Market, eoeMarket [6], alcatelclub [3], gfan [7], and mmoovv [10][1]. DroidRanger was able to find 211 malicious or infected apps out 204,040 apps from the five studied marketplaces, including two zero-day malware. Zhou and Jiang [22] made a one year effort and analyzed more than 1,200 malware samples, which covered a majority of state-of-the-art Android malware. Bugiel *et al.* [13] studied ways to defend against privilege-escalation attacks on Android. Such privilege-escalation attacks include confused deputy attacks and colluding attacks.

There are other survey works on mobile security and malware. Becher *et al.* [12] performs a comprehensive survey of mobile security from hardware to software. It is a comprehensive enumeration of existing wireless technologies and possible attacks against those technologies and devices. La Polla *et al.* [18] surveys mobile device security and complements the work in [12]. Peng *et al.* [17] performs a survey of malware on platforms such as Android, Windows Mobile and Symbian. The categorization of malware follows traditional jargons such as worms, viruses and trojans. They also survey malware propagation strategies. For our future work, we hope to have a system of categorization that systematically characterizes the underlying techniques of mobile malware.

7 Conclusion

This paper introduces a new type of malware that leverages Android's accessibility framework to activate its malicious payloads. The implementation of an example malicious application that uses the Accessibility APIs to masquerade as a legitimate application is detailed. Additionally, we describe how the emergence of this malware elicits the need for better malware categorization and characterization of malware. We describe results from the experiments we performed to quantify both the success of application launch detection and exploiting the logic error in the `ActivityStack` class. We show that, by adding a delay to the launch of the malicious `Activity`, we can guarantee 100% that the malicious `Activity` will be displayed. We also discuss possible strategies to mitigate this type of malware.

References

1. Accessibility. `http://developer.android.com/guide/topics/ui/accessibility/index.html`, 2013.
2. Accessibility services. `http://developer.android.com/guide/topics/ui/accessibility/services.html`, 2013.

[1] Link is no longer valid

3. Alcatelclub. http://www.alcatelclub.com/, 2013.
4. Android open source project. http://source.android.com/, 2013.
5. Application fundamentals. http://developer.android.com/guide/components/fundamentals.html, 2013.
6. eoemarket. http://www.eoemarket.com/, 2013.
7. Gfan. http://www.gfan.com/, 2013.
8. Juniper networks third annual mobile threats report. http://www.juniper.net/us/en/local/pdf/additional-resources/3rd-jnpr-mobile-threats-report-exec-summary.pdf, 2013.
9. Making applications accessible. http://developer.android.com/guide/topics/ui/accessibility/apps.html, 2013.
10. Mmoovv. http://android.mmoovv.com/web/index.html, 2013.
11. M. Becher, F. C. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proceedings of IEEE Symposium on Security and Privacy*, 2011.
12. M. Becher, F. C. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 96–111, 2011.
13. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.
14. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, 2011.
15. A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)*, 2011.
16. R. Hunt and S. Hansman. A taxonomy of network and computer attack methodologies. *Computers & Networks, Elsevier*, 24(1), February 2005.
17. S. Peng, S. Yu, and A. Yang. Smartphone malware and its propagation modeling: A survey. *Communications Surveys Tutorials, IEEE*, PP(99):1 – 17, July 2013.
18. M. L. Polla, F. Martinelli, and D. Sgandurra. A survey on security for mobile devices. *Communications Surveys Tutorials, IEEE*, 15(1):446 –471, February 2013.
19. V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Short Paper, Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.
20. A.-D. Schmidt, H.-G. Schmidt, L. Batyuk, J. H. Clausen, S. A. Camtepe, S. Albayrak, and C. Yildizli. Smartphone malware evolution revisited: Android next target? In *Proceedings of the 4th IEEE International Conference on Malicious and Unwanted Software (Malware 2009)*, pages 1–7. IEEE, 2009.
21. M. Zheng, P. P. C. Lee, and J. C. S. Lui. ADAM: An automatic and extensible platform to stress test android anti-virus systems. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2012.
22. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2012.
23. Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.