How Current Android Malware Seeks to Evade Automated Code Analysis

Siegfried Rasthofer[1], Irfan Asrar[3], Stephan Huber[2], Eric Bodden[1,2]

Center for Advanced Security Research Darmstadt (CASED)
[1] Technische Universität Darmstadt, Germany
siegfried.rasthofer@cased.de
[2] Fraunhofer SIT, Darmstadt, Germany
{stephan.huber, eric.bodden}@sit.fraunhofer.de
[3] Appthority, USA
iasrar@appthority.com

Abstract. First we report on a new threat campaign, underway in Korea, which infected around 20,000 Android users within two months. The campaign attacked mobile users with malicious applications spread via different channels, such as email attachments or SMS spam. A detailed investigation of the Android malware resulted in the identification of a new Android malware family Android/BadAccents. The family represents current state-of-the-art in mobile malware development for banking trojans.
Second, we describe in detail the techniques this malware family uses and confront them with current state-of-the-art static and dynamic code-analysis techniques for Android applications. We highlight various challenges for automatic malware analysis frameworks that significantly hinder the fully automatic detection of malicious components in current Android malware. Furthermore, the malware exploits a previously unknown *tapjacking* vulnerability in the Android operating system, which we describe. As a result of this work, the vulnerability, affecting all Android versions, will be patched in one of the next releases of the Android Open Source Project.

Keywords: Botnet, Android Malware, Code Analysis, Banking Trojans, Vulnerability

1 Introduction

According to a recent study [9], Android has reached a mobile-market share of 81%. There is an app for almost every need, provided by various app stores such as the Google PlayStore with 1.3M applications by July 2014 [36]. Besides apps that are mostly used for amusement, there are also more critical applications that handle confidential data such as mobile banking applications. According to a study of the Federal Reserve Board [28], more and more people switch from using cash and ATMs to using mobile banking with their smartphones. This makes phones a very attractive target for attackers who want to steal money from victims. Indeed, there is a big underground market for trading

stolen bank account credentials [38]. For instance, Symantec reported [38] that a single underground group made $4.3 million in purchases using stolen credit cards over a two-year period.

The Android operating system got enhanced with different security features, such as the 'Application verification' in version 4.2. Its goal is to protect the user against harmful applications. Despite those protection mechanisms, banking trojans are still actively spreading [6]; even worse, McAfee is predicting a rapid growth [18]. Very recently, we identified a new threat campaign underway in South Korea that emphasizes McAfee's prediction. The campaign stole, within two months, the credentials of more than 20,000 bank accounts of users residing in Korea. We identified a new malware family `Android/BadAccents` (named after the main component in the first stage of the trojan) that impersonates known banking applications in order to steal the user's credentials. Furthermore, it also steals incoming SMS messages, aborts phone calls and installs a fake anti-virus application.

In this paper, we describe in detail the techniques this malware family uses, and explain the current state-of-the-art of mobile malware development. The malware family clearly illustrates that mobile malware is becoming increasingly complex. In 2010, FakePlayer [11] was one of the first mobile malware families ever discovered. It implemented a simple premium SMS trojan, with only a few lines of Java code. As we show in this paper, however, current malware shows a highly complex structure comprising multiple malicious components and complex interactions between these components. In the case of Android/BadAccents, the complexity is further enhanced by an included zero-day-exploit (a vulnerability that was previously not known).

Many malware-detection frameworks, such as the one used in Google Play [13], or the ones used by anti-virus companies, however, aim at (semi-)automatically distinguishing between benign and malicious applications. To be able to initiate further actions, such as the take-down of a botnet, it is moreover crucial to be able to identify the actual malicious components. Given that every single day thousands of new apps, and versions of apps, are uploaded to the larger app stores, it is crucial that such an analysis can be conducted efficiently. Any manual analysis therefore must be supported by automated or semi-automated program-analysis tools. In this work we show, however, that current pieces of malware such as Android/BadAccents are raising significant challenges to static as well as dynamic code-analysis techniques. While we do not reiterate the well-known limitations from literature of both approaches [1,3], instead we demonstrate new challenges that are related to Android and which have to be considered on top of the well-known ones. For instance, the hiding of sensitive information in native code is no longer a theoretical problem for static analysis; it is already being exploited in the wild. The usage of multi-stage command and control (C&C) protocols is growing into a challenge for dynamic code-analysis techniques as well. Even malware-analysis frameworks that try to circumvent emulator-detection mechanisms [29] are not well prepared for current Android malware. There is still a big need for a proper environment setup, such as specific files on the SD

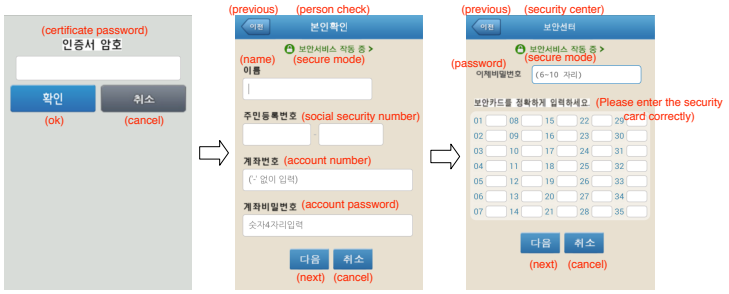

Fig. 1. Phishing of confidential banking credentials

card or specific apps installed, as otherwise the malicious behavior does not get triggered and hence cannot be observed. These are significant challenges that future code-analysis approaches will need to address.

The rest of the paper is organized as follows. Section 2 describes the details of the malware including an AOSP vulnerability. In Section 3 we identify the challenges for current state-of-the-art code analysis techniques and Section 4 describes the related work in the field of Android security while Section 5 concludes the paper.

2 Android/BadAccent Malware

During a threat-campaign investigation, we spotted an interesting malware sample that targets Korean users (more details in our technical report [31]). The threat campaign employed tactics such as social engineering to distribute Android malware. In particular, it distributed a new form of banking trojans that we designated as Android/BadAccents (named after the main component in the first stage of the trojan). Such mobile malware targeting Korea in many ways represents the best of breed practices when it comes to mobile malware development. In general, Android/BadAccents is a banking trojan that tries to steal bank-account credentials through a phishing attack. The victim is asked to enter her confidential data into a Graphical User Interface (GUI) that looks identical to the one of a benign mobile banking application. But the malware's GUI is designed by the attacker, and is instrumented to steal the credentials instead. Figure 1 shows such a fake GUI component which appears after a fake security message which prompts the user for some action.

Android/BadAccents demonstrates the complexity of current Android banking trojans. Different interactions, environment settings and conditions are necessary before a specific malicious behavior gets triggered. The malware sample uses different techniques to hide the malicious behavior as long as possible. Figure 2 gives an overview of the main components in the Android/BadAccents malware and shows the complexity of environment settings, workflow and external events that are involved. Especially the *Intercept SMS* components show that current attackers do not only rely on a single channel for transmitting sensitive data. Instead they use several ones, in this case e-mail and HTTP connections.

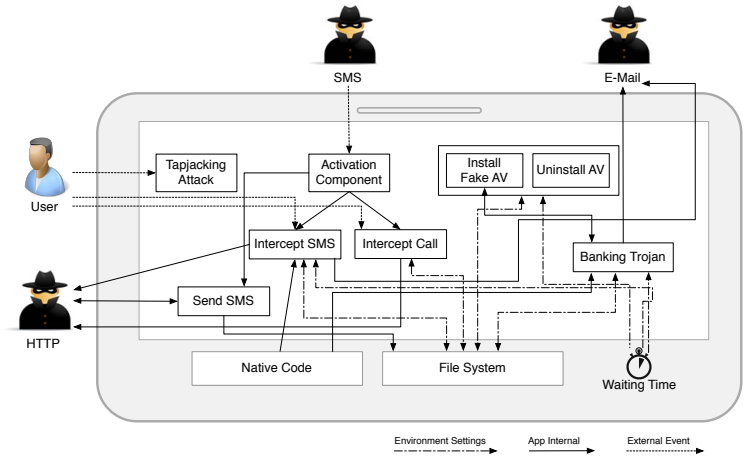

Fig. 2. Interactions and environment settings necessary for triggering malicious behavior in the Android/BadAccents Malware

In the following we will describe each component individually in detail, and its requirements for triggering a malicious behavior. The resulting requirements for code-analysis tools are described afterwards.

2.1 Send SMS

The *Send SMS* component gets activated at application startup time and is responsible for sending SMS messages to all contacts on the phone that have more than 5 digits as a phone number. It first initializes a connection to the C&C server, using the victim's device's phone number for identification, from which it receives the text for the SMS message. Additionally, it saves all phone numbers of the contacts into a global storage (*SharedPreferences* file). After receiving the text, the component immediately sends a message containing that text to all contacts. This mechanism is probably used for spreading the malware to all contacts. We assume that the text from the C&C server contains spam messages together with a download link to the Android/BadAccents malware. The attacker's aim is to infect the SMS receiver with additional malware by clicking on the link.

2.2 Activation Component

The *Activation Component* is responsible for receiving C&C messages via SMS. Using SMS as a protocol is an important design decision that is different from traditional IP-based approaches known from infected PCs. Zeng et al. [44] already illustrated this design in 2012. The main advantages of an SMS-based approach instead of IP-based are the fact that it does not require steady connections, that SMS is ubiquitous, and that SMS can accommodate offline bots easily.

The *Activation Component* is implemented as a broadcast receiver, which is active from the time the application starts. This broadcast receiver registers 63 different actions it can react to. However, it uses only a single one of them, the SMS-received action. It intercepts all incoming SMS messages and triggers the malicious behavior only if the message contains special commands. More concretely, it is responsible for activating the *Intercept SMS* and *Intercept Call* component (details below). The Android/BadAccents malware contains two specific checks on the incoming SMS number. It checks for '+84' and '+82' numbers, which indicates that the malware expects SMS from a C&C SMS server either located in China or South Korea. The message has to have a special format that contains either 'sd_$\langle MESSAGE \rangle$', 'ak40_0', 'ak40_1' 'call_0' or 'call_1' and can be concatenated with '_' (e.g., 'ak40_1_call_0'). The 'ak40' command is responsible for the *Intercept SMS* component and activates that component with 'ak40_1' and deactivates it with 'ak40_0'. The 'call' command is responsible for the *Intercept Call* component and 'call_1' activates and 'call_0' deactivates it. Activating a component is implemented by storing activation-flags (e.g. $\langle call, 1 \rangle$) into a SharedPreferences file, deactivating components is done by storing deactivation-flags (e.g. $\langle call, 0 \rangle$). The individual components get called in a specific time interval in which they first check for the appropriate activation-flag before running it. This is indicated as dotted arrows in Figure 2 from both components to *File System*. The 'sd_$\langle MESSAGE \rangle$' command is equivalent to the functionality of the *Send SMS* component (see section 2.1). The main difference is the communication channel. Instead of receiving the text of the message body via HTTP (*Send SMS* component), it uses only the SMS channel by taking the message from the incoming C&C SMS ($\langle MESSAGE \rangle$).

Intercept Call The *Intercept Call* component intercepts all incoming calls and checks whether the caller is stored as a contact on the device or not. If this is not the case, the call gets aborted and the entry in the call log gets deleted. We assume that the attackers want to abort calls from the bank which could have detected suspicious transactions caused by the banking trojan.

Intercept SMS This component intercepts all incoming SMS messages that do not contain any C&C command and leaks the information to the attacker via HTTP and E-mail. It uses two channels in parallel for a more reliable data theft. The credentials of the E-mail account are hidden in native code, which makes the detection hard for static analysis approaches that operate purely on the Dalvik bytecode. Listing 1.1 shows two native methods that return the constant username and password (original credentials are removed) that get called in the `onCreate` method (listing 1.2) and stored into a SharedPreferences file (`setValue` method). Before sending the email, the credentials are extracted from the SharedPreferences file in order to authenticate against the email server.

```
void Java_com_MainActivity_stringUser() {
  return "USERNAME";
}

void Java_com_MainActivity_stringPassword() {
  return "PASSWORD";
}
```

Listing 1.1. Methods in Native Code

```
public native java.lang.String
      stringPassword();
public native java.lang.String stringUser();

public void onCreate(Bundle b) {
...
user = stringUser();
setValue("musername", user);
pw = stringPassword();
setValue("mpass", pw);
...
}
```

Listing 1.2. Accessing Native Methods within Java

2.3 Install/Uninstall

The Install/Uninstall component first removes one particular app, the 'AhnLab V3 Mobile Plus 2.0'[1] app in case it is installed on the device. This is a malware-scanner application especially designed for detecting banking trojans. In the *Banking Trojan* component, a fake 'AhnLab V3 Mobile Plus 2.0' application gets installed which impersonates the original app and which contains malicious components similar to Android/BadAccents.

2.4 Banking Trojan

The *Banking Trojan* component tries to hide the application's icon from the launcher. This is possible with a singe API call (*setComponentEnabledSetting* in *PackageManager*) and does not require any permission. After a delay of 30 minutes, the malware looks for DER-formatted certificates stored under a specific folder on the SD card. If found, the malware checks whether the user has installed specific Korean banking applications such as Shinhan Bank, Woori Bank or NH Bank. This indicates that the threat campaign primary targets user from Korea. Next, if one of these applications is installed, it dynamically creates a new view impersonating this app. The 'fake' app uses social engineering in showing security warnings that should convince the user to provide the attacker her data.

After accepting the security messages, the attacker tries to steal the banking victim's credentials. Figure 1 shows the individual GUI fields the user has to go through. It is worth mentioning that input into the fields has to satisfy specific criteria such as the certificate password has to be entered twice or the password in the *security center* has to have more than 5 digits. If everything got filled out correctly, all the data, together with the certificate gets sent to the malicious e-mail account. Similar to the *Intercept SMS* component (see section 2.2), the e-mail-account credentials are loaded through native methods.

2.5 Gain Administration privilege

Besides the malicious components above, we also found a zero-day vulnerability of the AOSP abused by the malware. The Android/BadAccents malware

[1] https://play.google.com/store/apps/details?id=com.ahnlab.v3mobileplus

tries to obtain Android Device Administration privileges [39] without the user's knowledge.

The Android Device Administration API was introduced for applications to support enterprise features [39]. It provides functions on the system level with varying security impact. An application that is granted such privileges can, for example, lock the device screen, encrypt user data, or initiate a factory reset of the device. The full set of supported system functions is described in the developer documentation [40].

When an application requests administration privilege, the Android OS shows a warning message to the user, who then has to accept or deny the request. The malware abuses the mentioned vulnerability to trick the user into accepting the administration request by a so-called *tapjacking* attack [24] where the user clicks on a seemingly benign object, but instead activates the Device Administration. To the best of our knowledge this attack form is currently the only way to obtain administration privilege without resorting to some root exploit or without an explicit visible user confirmation.

Tapjacking Attack Summary The following subsection gives a short summary about the concept of tapjacking attacks. The formal name or most common name in research for a tapjacking attack is UI redressing [24] and subsumes tapjacking as a specific case.

The basic idea behind tapjaking on Android is not to directly exploit some system vulnerability, instead its focus is to force the user to an interaction without her knowledge and to hide the system or application information which is shown as a consequence of this hidden interaction. A harmlessly looking overlay window is brought to the foreground, hiding the real application behind the overlay window. The design of such an overlay window can be freely defined, for instance posing as a game or some generic application dialog (see figure 3).

The requirements for such an attack are all provided by the Android user interface (UI) design API. Such attacks can be performed in different ways, but the main premise is to generate a UI element which can be layered over applications and routes touch gestures to the underlying application. An additional requirement for successful tapjacking is the hidden start of the victim application or a part of the application [7] behind the overlay. Exported activities or defined `intent-filters` in applications can facilitate such hidden starts. System applications or Android settings can be accessed via system intents. To route taps through underlying applications, Android provides settings that make a widget transparent for touches.

Analysis of the Tapjacking Vulnerability After a detailed analysis of the malicious application, we isolated the code responsible for the tapjacking attack and reassembled it into a stand-alone proof-of-concept implementation. The malware uses the described tapjacking attack to obtain Android Device Administration privilege and thus the ability to lock the device screen. Another aspect is the uninstall protection. Once the admin privilege is granted, antivirus tools can

```
1 private void setupLayoutParams() {
2     layoutParams = new WindowManager.LayoutParams(WindowManager.LayoutParams.TYPE_SYSTEM_OVERLAY,
3                 WindowManager.LayoutParams.FLAG_FULLSCREEN,
4               WindowManager.LayoutParams.FLAG_SCALED);
5     layoutParams.flags = WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE;
6     ...
7 }
```

Listing 1.3. Settings for ovelay window layout paramters

no longer remove the malware. The attack can be illustrated as shown in figure 3. The victim only sees an application window requesting *"Please update to the*

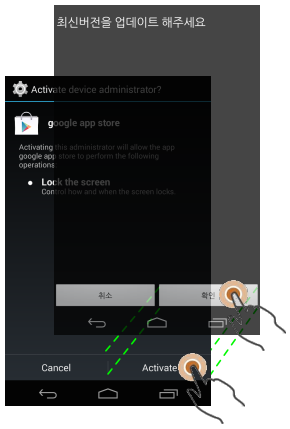

Fig. 3. Tapjacking Attack on Android Device Administrator App

latest version" with a *confirmation* and a *cancel* button. Pressing *confirmation* she activates the device administration feature.

Therefore the tapjacking attack at first starts the admin request dialog by sending the system intent `android.app.action.ADD_DEVICE_ADMIN`. Due to the asynchronous execution character of Android the application does not stop after calling the administration activity and executes a method showing the overlay window hiding the administration activity (see figure 3). The overlay window is an extended `LinearLayout` class defining specific layout properties (see listing 1.3). The first layout option is the overlay definition itself. The last option (`FLAG_NOT_TOUCHABLE`) is the crucial factor. It makes the window transparent for touches and therefore every touch gestures on it were received on the application behind it. Considering the malware example the victim assumes she confirms the update request, but in reality she activates the administration privilege. This form of attack is working to Android Kitkat version 4.4 and older Android versions.

In the newer Lollipop version (Android 5) this part of the malware is not working correctly anymore. Thus the victim would detect the attack. With a slight modification of the isolated proof of concept code we could show that

the attack is still possible and that there is no tapjacking protection for the administration activity. We informed Google about our discovery and provided a patch preventing such an attack.

Bug Fix and Counter-Measures As a counter measure against *tapjacking* Android provides some specific protection mechanism. It was introduced in API level 9 and is enabled by the method `setFilterTouchesWhenObscured()` which discards touches whenever the view's window is obscured by another visible window. As a result, clicking on the overlay window does not affect the underlying window. Alternatively, view elements can be protected on the level of XML declarations by defining the attribute `android:filterTouchesWhenObscured`.

Our provided patch introduces these functions to the accept- and deny-button for the administration activity. The patch code can be found here [2]. An attacker app thus can no longer trick the user into obtaining administrator privileges without her explicit consent. The described blacklisting approach of the Android OS is currently the only way to protect applications against such tapjacking attacks. To mitigate or completely prevent such attacks every critical android system application and also every provided Android application from the PlayStore should activate the protection. A better way would be some generic protection approach integrated in the Android OS.

Besides the counter measures from the AOSP, there already exists other mitigations from different researchers. For instance, Niemietz et al. [24] introduced an additional security layer into the AOSP consisting of a transparent layer over each foreground application. If a malicious application tries to get above the victim activity to set up a tapjacking attack, the security layer can catch all the touches trying to reach the protected app. We believe that a concept directly integrated into the AOSP, without further additions by the developer, would be simpler to maintain and should be integrated into Android.

3 Mobile Malware Analysis Challenges

The previous section describes in detail a representative malware family that shows the state-of-the-art for current Android banking malware. Mobile malware differs from PC malware in different aspects [25] resulting in the need for more complex analysis. One important aspect is the sensor-based event system of mobile devices, which allows the malware to react to incoming SMS, location changes etc., adding more complexity for automated malware-analysis approaches. Also the modular design of Android applications is an important factor for the need of more sophisticated analysis techniques, given that apps can use services and activities [10], and can combine different programming languages (e.g. JavaScript or native code) in one application.

Nevertheless, the goal in PC and mobile malware analysis always remains the same: *to identify the threat and take the necessary actions to eliminate the threat.*

[2] https://android-review.googlesource.com/#/c/127602/

In case of a trojan stealing personal information, it is necessary to know *what* data are stolen and *where* are they sent to. These are very important questions for security analysts in the case of active malware, because the analyst has to initiate further steps to remove the threat, for instance a C&C server takedown. The first question usually poses *dataflow questions* [3] whereas the latter one poses *reachability questions* [4]. Answering these questions in an automatic way would save a lot of time and money during investigation. Generally, two code-analyses approaches can be used: static or dynamic analysis techniques, or—more likely—a combination of both. Both approaches have well-known limitations [1, 3], but the Android OS itself introduces new additional challenges.

In the following we look more concretely into the different challenges for static and dynamic analysis approaches that will arise during an analysis of the Android/BadAccents example. Challenges such as emulator detection mechanisms, obfuscation techniques or packers are not covered, since they are already described in previous work [29, 37]. We use the Android/BadAccents malware as a representative for the complexity of current Android malware since it is implemented in a high-end engineering manner and contains various malicious components.

3.1 Static Analysis Challenges

In general, static analysis is a very powerful technique since one can reason about all execution paths in the application. This is especially useful to answer the *what* question in an investigation, i.e., what data are leaked.

Unfortunately, Android applications raise new challenges to static dataflow analysis, which are not only a theoretical problem anymore, as Android/BadAccents demonstrates. Recall that the malware sends sensitive data via e-mail where the origin of the data-source is stored in native code (Section 2). By answering the question *What is the username and password of the email account?*, one would either use a forward [3] or backward [14] dataflow analysis (*dataflow problem*) across language borders. The fact that the dataflow analysis has to deal with multiple code representations (Dalvik and native ARM) makes it more complex. Moreover, there is a need for new concepts how to handle *inter-language dataflows*. A new research direction could be the design of a common intermediate representation of Dalvik and native code which is not easy since both languages (Java and C/C++) have significant differences such as the pointer handling in native code. To the best of our knowledge, there is currently no real solution to this practical problem. But even an analysis of just the Java part raises new challenges for code-analysis approaches. The so-called *inter-component* dataflow tracking is well-known from literature [19, 27], but the approaches do not yet scale in practice, due to path-explosion problems [19]. Besides the inter-component problem, Android/BadAccents has shown another interesting problem, namely the dataflow through persistent storages (e.g., SharedPreferences) where the data-source flows to a persistent storage and gets read at some later point from it to continue the flow to the sink-method. The current solution for such cases is an over-approximation of the dataflows where all data

read from persistent storages are assumed to be 'sensitive' even if this is not the case. In practice, this produces too many false positives, which overwhelms an analyst with false-warnings. This is especially noticeable for Android applications, in comparison to applications in the PC world, since Android has a lot of API support for (temporary) storing data, which is actively used by developers as the Android/BadAccents sample shows (see Listing 1.1). Post-analysis approaches [43] that try to reduce false-positives after the main data-flow analysis are an interesting research area, but do not solve the main issue. Static code-analysis approaches for Android have to get advanced by adding new algorithms such as *quantitative information flows* [22] to reduce the false-positive problem.

As a summary, static analysis is very useful in general, but the analysis of Android applications include more challenges for which no concrete solution exists yet.

3.2 Dynamic Analysis Challenges

For all the above reasons, dynamic analysis or *behavior analysis* [33] has been advocated in the context of malware analysis [20, 35]. Furthermore, the answer to the *what* question is usually given by a dynamic analysis. To be complete however, dynamic analysis requires a set of execution traces that are representative of all the possible program behaviors. While observing all the program behaviors of a complex program is impractical, several coverage criteria have been proposed in the software testing literature to approximate full behavior coverage; their effectiveness however is still debated [15, 16]. Different facts in Android significantly hinder the triggering of malicious behavior by dynamically executing the code. For the example of Android/BadAccents we summarize the major problems in the following three categories: external events, environment settings and user interaction.

External Events The Android OS is a sensor-based event-driven environment that reacts to various events and executes the registered event handlers. For instance, an incoming phone call is modeled as an Android internal event, called intent [41], which can be intercepted through a corresponding callback defined in the application. This produces the first challenge: a simple dynamic analysis is insufficient if it fails to generate the proper events. Researchers have proposed several approaches [34, 45] for fuzzy testing Android components by sending abnormal/random intents to the components in order to identify security bugs. Nevertheless, section 2.2 shows that the malicious behavior gets only triggered if, for instance, the incoming SMS or HTTP request have the proper format. Furthermore, the ordering of events can also matter. For instance, the *Intercept SMS* component described in section 2.2 gets only activated if the attacker first sends an 'activation-command' and second the user sends an SMS to the victim. This makes a fully automated triggering of the original *Intercept SMS* component extremely difficult.

Environment Settings A successful analysis of Android malware with a *behavior analysis* requires a properly setup environment, since many malware families check for clues of an emulated environment before they trigger their malicious behavior. The environment thus must be set up in such a way that it *emulates* all aspects of a proper smartphone. To some extent, this is impossible. For instance, emulators will always expose timing and cache behavior that is clearly distinguishable from real phones [29]. But not only emulator checks complicate dynamic analysis. The problem of *time bombs*, were the malware waits for a specific time until it triggers its malicious behavior (see section 2.2) poses a serious problem to dynamic analyses. This problem is similar to malware in the PC world, but has a much higher impact as the Wall Street Journal reported this year[3]. The Android malware went undetected in the Google Play store due to a time bomb and infecting close to 10 million devices. Time bombs can be 'evaded' by speeding up the time in the environment. Unfortunately, this might still be insufficient with state-of-the-art malware samples. Android/BadAccents requires specific files in the file system (DER-formatted files), specific contact data stored on the device and specific apps installed on the device (Korean banking apps) before the banking trojan gets activated. Since there is an exponential amount of combinations for different settings, it is very difficult to come up with a proper setting of an environment that emulates all that.

User Interaction Mobile applications give a user a lot more possibilities for interaction since smartphones are in general an event-driven system. Interactions include the clicking on buttons, swiping objects, the reaction on incoming messages or filling out forms. Many of these interactions may need to be emulated to facilitate a meaningful dynamic analysis. Again, there has been a lot of research in the area of Android GUI testing [1, 8] but to the best of our knowledge none of these approaches would successfully work on Android/BadAccents. For instance, the first GUI in figure 1 requires the user to input her password two times. Randomly inserting some values and automatically clicking on the 'ok'-button would not result in a page switch. Also the password in the first and third screen page has to have more than 5 digits, otherwise the GUI will not switch to the next one and the malicious behavior of stealing the credential data (shown in figure 2) would not be triggered. Figuring out the right combination of inputs would require the most sophisticated techniques, such as symbolic execution, which are hard to scale in general. Further research in this field is clearly required.

4 Related Work

In this section, we describe a number of related work in the context of Android malware analysis that addresses attacks and threats.

[3] http://blogs.wsj.com/personal-technology/2015/02/04/android-malware-removed-from-google-play-store-after-millions-of-downloads/

Abusing the device administration privileges in order to make the uninstallation of applications more difficult is a common technique used in Android malware. For instance, the Android malware OBAD [42] requests administration privileges. Additionally it uses an Android vulnerability (fixed in Android 4) to hide its entry from the device administration list. This means it was also not possible for a user to manually revoke the admin privileges for uninstalling the malware. Another Android vulnerability [2], which got fixed in version 4.4.3, shows that it is even possible to prevent the installation of an arbitrary app on the device. Also different ransomware applications like Android/Koler [17] try to gather administration privileges to lock the device and encrypt the data storage. Another related malware in the context of banking trojans and C&C is the Zeus [23] trojan. This banking trojan exists despite of Android also for different mobile platforms like Blackberry, Windows Mobile or Symbian. The focus of the first Zeus trojan was to steal mTAN numbers through sms interception. Newer versions of Android trojans are aiming on stealing credit cards through wireless connection. Zhou et al. showed [47] a first global study about different types of Android malware. They showed that normal applications were enriched with malicious content and found different apps containing similar malware code. Depending of this payload they grouped them in different families.

Besides the internal threat detection framework of AV companies, there exist also other open-source approaches that crawl various app-stores for detecting malicious applications. Lindorfer et al. [21] propose a framework for discovering multiple instances of a malicious Android application in a set of alternative application markets. Based on some lightweight indicators, such as the package name or the hash of an application, they found various malicious applications in different markets. DroidSearch [30] is another framework that crawls different app stores and stores for each application meta-data into a database. The database can be queried afterwards for detecting vulnerabilities or malicious applications.

Isolated environments for analyzing and detecting Android malware are a well-established technique in the context of mobile malware analysis. Andrubis [20] or the Mobile Sandbox [35] are two examples. Usually, they use lightweight static analysis techniques to find concrete malware patterns [5] in combination with a lightweight dynamic code analysis approach that monitors the application in a secure environment. The results are used to detect suspicious behavior or evaluate the risk factor [26] of an application. Due to the nature of the lightweight analysis, the proposed techniques reaches its limitations when it comes to sophisticated malware that triggers malicious behavior only under specific circumstances.

Signature based approaches [46] are a well-known techniques used by many anti-virus applications. Zheng et al. [46] proposed a new signature methodology that was able to easily discover repackaged malicious applications or even zero-day malware samples. Apposcopy, a tool proposed by Feng et al. [12] improves signature based approaches by a semantic based approach that specifies signatures that describe semantic characteristics of malware families. Both approaches rely on static information extracted from the bytecode. Hardening or

even packers complicates the detection of malicious applications as shown by different researchers [32].

5 Conclusion

In this paper, we have described an investigation of a new malware family that infected more than 20,000 mobile devices in Korea. We described in detail the components of current state-of-the-art mobile malware development. Furthermore, we compared each individual technique of the malware with current state-of-the-art malware-analysis techniques. Our results show that current malware poses many challenges to malware analysis techniques in order to trigger malicious behavior, showing the need for further research in this area. We furthermore demonstrated a new tapjacking attack that is exploited by the Android/BadAccents malware. It causes a security threat, as the user can be tricked into clicking/tapping on objects that trigger unintended behavior. The Android Security Team confirmed the attack and our proposed patch will be integrated in the next major release of Android.

Acknowledgements This work was supported by the Deutsche Forschungsgemeinschaft within the project RUNSECURE, the BMBF within EC SPRIDE and ZertApps, by the Hessian LOEWE excellence initiative within CASED, and by the DFG Collaborative Research Center CROSSING.

References

1. Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th FSE*, FSE '12, New York, NY, USA, 2012. ACM.
2. Steven Arzt, Stephan Huber, Siegfried Rasthofer, and Eric Bodden. Denial-of-app attack: Inhibiting the installation of android apps on stock phones. In ACM, editor, *Proceedings of the Fourth ACM SPSM Workshop*, November 2014. https://github.com/secure-software-engineering/denial-of-app-attack.
3. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN PLDI*. ACM, June 2014.
4. David Basin, Sebastian Mödersheim, and Luca Vigano. *An on-the-fly model-checker for security protocol analysis*. Springer, 2003.
5. Thomas Bläsing, Aubrey-Derrick Schmidt, Leonid Batyuk, Seyit A. Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *5th International MALWARE'2010*, Nancy, France.
6. Carlos Castillo. Phishing attack replaces android banking apps with malware, June 2013. Blog.
7. Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.
8. Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN OOPSLA*, OOPSLA '13, New York, NY, USA, 2013.

9. International Data Corporation. Worldwide quarterly mobile phone tracker 3q12, November 2012. Blog.
10. Android Developer. Application fundamentals, April 2015. `http://developer.android.com/guide/components/fundamentals.html`.
11. Ken Dunham. *Android Malware and Analysis*. CRC Press, 2014.
12. Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT FSE*, FSE 2014, New York, NY, USA, 2014. ACM.
13. Google. Android security 2014 year in review, April 2014.
14. Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: Program slicing for smali code. In *Proceedings of the 28th Annual ACM SAC*, SAC '13, New York, NY, USA, 2013.
15. Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th ICSE*, ICSE '94, Los Alamitos, CA, USA, 1994.
16. Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th ICSE*, ICSE 2014, New York, NY, USA, 2014.
17. Bitdefender LABS. Reveton / icepol ransomware moves to android. blog.
18. McAfee Labs. Threats predictions, 2015.
19. Li Li, Alexandre Bartel, Tegawend Bissyande, Jacques Yves Klein, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th ICSE*, ICSE 2015. ACM, 2015.
20. Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the the 3rd International Workshop BADGERS*, 2014.
21. Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. AndRadar: Fast discovery of android applications in alternative markets. In *Proceedings of the 11th Conference DIMVA*, volume 8550, London, UK, July 2014.
22. Enrico Lovat, Johan Oudinet, and Alexander Pretschner. On quantitative dynamic data flow tracking. In *Fourth ACM Conference CODASPY*, 2014.
23. Denis Maslennikov. Zeus-in-the-mobile - facts and theories, October 2011. Blog.
24. Marcus Niemietz and Jörg Schwenk. Ui redressing attacks on android devices, 2014. BlackHat Asia.
25. Ruchna Nigam. A timeline of mobile botnets, April 2015.
26. NViso. `http://apkscan.nviso.be/`.
27. Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*, Washington, D.C., 2013. USENIX.
28. Board of Governors of the Federal Reserve System. Consumers and mobile financial services 2014, March 2014.
29. Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, New York, NY, USA, 2014.

30. Siegfried Rasthofer, Steven Arzt, Stephan Huber, Max Kohlhagen, Brian Pfretschner, Eric Bodden, and Philipp Richter. Droidsearch: A tool for scaling android app triage to real-world app stores. In *Proceedings of the IEEE SAI 2015*, July 2015.
31. Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. An investigation of the android/badaccents malware which exploits a new android tapjacking attack. Technical report, TU Darmstadt, Fraunhofer SIT and McAfee Mobile Research, April 2015.
32. Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC ASIA CCS*, ASIA CCS '13, New York, NY, USA, 2013.
33. Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, 19(4), December 2011.
34. Raimondas Sasnauskas and John Regehr. Intent fuzzer: Crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, WODA+PERTEA 2014, New York, NY, USA, 2014. ACM.
35. Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM SAC*, SAC '13, New York, NY, USA, 2013. ACM.
36. statista, jul 2014. `http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`.
37. Tim Strazzere. Android hacker protection level 0. Defcon 22.
38. Symantec. Symantec report on the underground economy, 2008.
39. Android Developer Team. Device administration. `http://developer.android.com/guide/topics/admin/device-admin.html`.
40. Android Developer Team. Device policymanager. `http://developer.android.com/reference/android/app/admin/DevicePolicyManager.html`.
41. Android Developer Team. Intent. `http://developer.android.com/reference/android/content/Intent.html`.
42. Emre Tinaztepe, Doğan Kurt, and Alp Güleç. Android obad. Technical report, COMODO, July 2013.
43. Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC CCS*, CCS '14, New York, NY, USA, 2014. ACM.
44. Yuanyuan Zeng, Kang G. Shin, and Xin Hu. Design of sms commanded-and-controlled and p2p-structured mobile botnets. In *Proceedings of the Fifth ACM WISEC Conference*, WISEC '12, New York, NY, USA, 2012. ACM.
45. Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC CCS Conference*, CCS '13, pages 611–622, New York, NY, USA, 2013. ACM.
46. Min Zheng, Mingshen Sun, and John C. S. Lui. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *Proceedings of the 12th IEEE International TRUSTCOM Conference*, TRUSTCOM '13, Washington, DC, USA, 2013. IEEE Computer Society.
47. Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE S&P*, SP '12, Washington, DC, USA, 2012. IEEE Computer Society.