BUILDING A TEST ENVIRONMENT FOR ANDROID ANTI-MALWARE TESTS

Hendrik Pilz
AV-TEST GmbH, Klewitzstr. 7, 39112 Magdeburg, Germany

Email hpilz@av-test.de

ABSTRACT

The growth of the Smartphone market over the past five years has attracted malware authors as well as security companies. Due to its open architecture and market share the *Android* system is very interesting for both sides. There are more than 40 commercial anti-malware products from different vendors and plenty of malware. From a consumer's viewpoint the situation is as follows: the user requires a security solution against the growing threat but in order to make a decision about which one to install he has to rely mainly on user comments and ratings. Hence professional comparative anti-malware tests are very important for consumers. However, vendors also benefit from these tests. This paper will show the different test scenarios, including on-demand and on-access detection as well as performance and battery impact. We will also cover the set-up of a sample test environment regarding the following key aspects:

- What hardware and software is required to perform tests?
- Since the number of test devices in a lab is limited, how can the load be managed?
- What must be considered when building a malware collection for tests?
- Testing is time consuming; is there a way to automate tests?

INTRODUCTION

Since cybercrime is a billion dollar business, these criminals are always looking for new opportunities to earn easy money. Because of the rapid market growth, Smartphones are an attractive target and allow efficient business models like premium SMS fraud. Just like in the world of personal computers, different mobile operating systems have different popularity among the malware authors. Targeting *Android* has several advantages over the other Smartphone systems besides its high market share. The open platform is available for all kinds of hardware and not limited to phones. The prices for *Android* devices vary from cheap to very expensive, which ensures a broad target group. The most important factor for *Android* in making it an appealing target for cybercrime is the possibility to install apps from unknown sources, which makes it easy to trick users into installing malicious apps.

Since January 2011 our *Android* malware collection has been rapidly growing to more than 20,000 unique samples. With the number of malware samples, the number of security products increased as well. Up to now, *AV-TEST* is the only testing lab that has performed a comparative malware detection test for more than 40 mobile security apps. *Android* as a new test platform requires the development of new testing methodologies. Regarding their anti-malware functionality, many mobile security apps are derived from traditional desktop products, e.g. most of them provide an on-demand scan and real-time protection. Looking at desktop products we know that the detection rates of the two functions are usually similar. We have often seen that this doesn't have to be the case on *Android*. That is why we can't simply use the same testing methodologies as we use for desktop PCs.

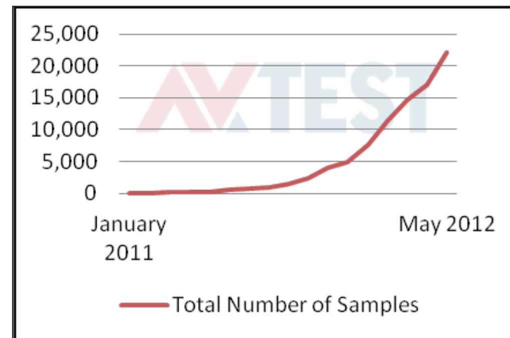

Figure 1: Android malware collection growth.

HARDWARE AND SOFTWARE REQUIREMENTS

Before the start of our first test we had to decide whether to run the tests in an *Android* emulator or on real devices. Regarding software tests in general, emulators have several advantages over real devices. Besides the possibility to easily switch between API versions and screen sizes or to reset the system to a clean state automatically, an emulator also provides root-access, which might be required to analyse scan reports without the need to exploit the system. There are also some disadvantages. While the testing tools that we have developed in-house work in an emulated environment, many anti-malware products don't. Depending on the set of tested applications it might be required to activate an app via SMS, which isn't possible without a real phone number. We've also experienced issues with cloud technologies in the emulator. The emulated 3G connection might have too high a latency for querying the cloud of some vendors. While the advantages of the emulator make testing more comfortable, the disadvantages limit the number of apps which can be properly tested.

Regardless of which test environment we choose, a host PC which is capable of running the *Android* SDK is required. As the *Android* SDK is available for *Windows*, *Mac OS X* and *Linux*, this requirement isn't hard to fulfil. If we want to run the tests in the emulator, the system requirements depend on the number of emulator instances.

We recommend the use of real *Android* devices for the tests, because no vendor can object to the use of a real device. In such a set-up the PC is primarily used to control one or more test devices via the Android Debug Bridge (adb). This may require an additional driver for specific devices. In order to be able to send and receive SMSs with the test device the phone is

Tests in the emulator
+ Cost efficient, scalable
+ Root privileges for scan report analysis
+ Easily switch between API versions and hardware configurations
+ Automatically reset the system to a clean state with snapshots

Tests on real devices
+ Real user experience
+ App activation via SMS possible
+ Vendors can't object to testing their software in a real environment

Table 1: Advantages of the emulator and real devices.

equipped with a prepaid SIM card. For Internet access we use a Wi-Fi connection. For performance testing we use a self-developed app on the device to monitor processes. The implementation of such a monitoring app will be described later in the paper. The software-based approach to measure performance is easiest to realize. For real measurements of the impact on the battery lifetime there are too many environmental factors, e.g. the room temperature or the age of the battery.

PRODUCT MANAGEMENT

Because there are more than 40 security products to be found in the *Google Play Store*, a comparative test may contain more products than the number of devices you have access to. Anti-malware tests require similar conditions among all programs tested for comparability. This includes a common baseline for malware signatures. Because many vendors use cloud technologies, it's also necessary to run all tests simultaneously to some extent. Due to the nature of the cloud, test results may vary if you retest at a later point in time. Thus it's very important to precisely document all test results to ensure the verifiability of the test.

For the reproducibility it's ideal to create a device image after an app has been installed and updated. The device can be restored and you can set up the next app. Currently most mobile security apps are updated once per day, so you ideally complete this procedure for all apps tested within one day. If you use just one device for 20 apps under test you have 24 minutes to install a single app on an eight-hour day for example. This is more than enough time. Usually the installation of an app takes just a matter of seconds. The problem is: you can't easily create device images before *Android 4.0*. The emulator provides a snapshot feature, so in an emulated environment you could easily create such images by saving the complete directory of your virtual device. Creating an image on a real device would require root privileges, but we want the tests to be run on stock devices. If a product doesn't score well, vendors could object to using rooted devices. And testers also can't rely on rooted devices as rooting may be prohibited by future *Android* updates.

Since *Ice Cream Sandwich* (*Android 4.0*), adb supports backup and restore of apps and data.

```
$: adb backup -f <file> -apk -shared -all -system
$: adb restore <file>
```

The backup includes all apps as APK files and the SD card content. When you run the backup command, the device asks you to enter an encryption password. If you don't want your backup to be encrypted, just leave the field blank.

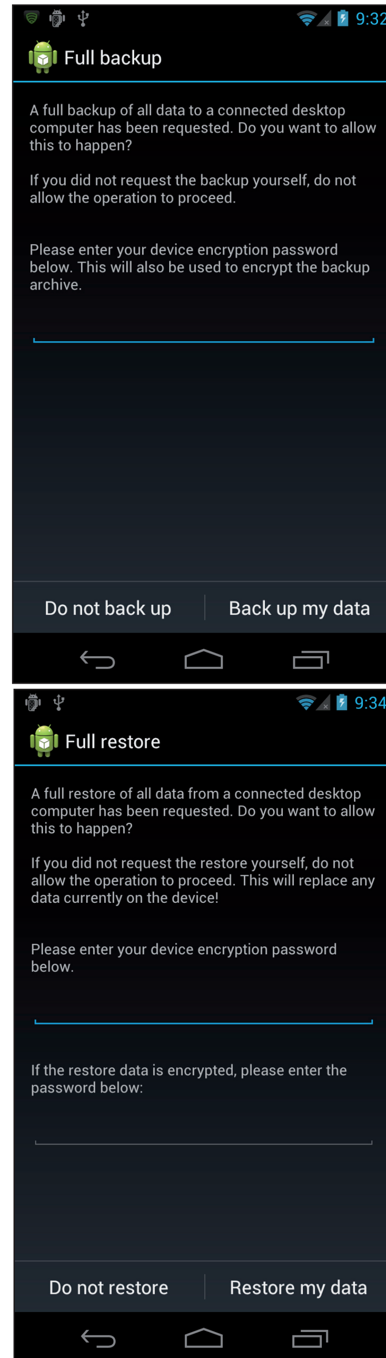

Figure 2: adb backup and restore requires user interaction on the device.

 © VIRUS BULLETIN LTD

TEST SCENARIOS

Due to our experience with tests of *Windows* anti-malware products we are familiar with many test scenarios for detection, repair and usability. Not all of these test scenarios can be performed in such a way on a mobile device. The possibilities of anti-malware products are rather limited, e.g. when it comes to removal of malware or behaviour-based detection. The main problems for such a test are the restricted permissions of the security app. Each app on a device has its own user id and apps are usually not allowed to access directories or files of other apps and the system. So a security app has a very limited view of the file system. Most apps detect and remove malware by scanning the installed APK files (which are world readable) and using the system's internal uninstall activity for each detected malicious APK file. Therefore, testing the clean and repair abilities isn't necessary as the cleaning is performed by the system. If the malware gains root privileges even the system might fail to completely remove it. This case won't be discussed here as most products aren't yet able to deal with these kinds of attacks.

Other security features like safe browsing, anti-spam and phishing protection aren't very common among mobile security apps (Figure 3). Tests of these functionalities will be performed as soon as these features are more widely spread among apps and allow a comparative review.

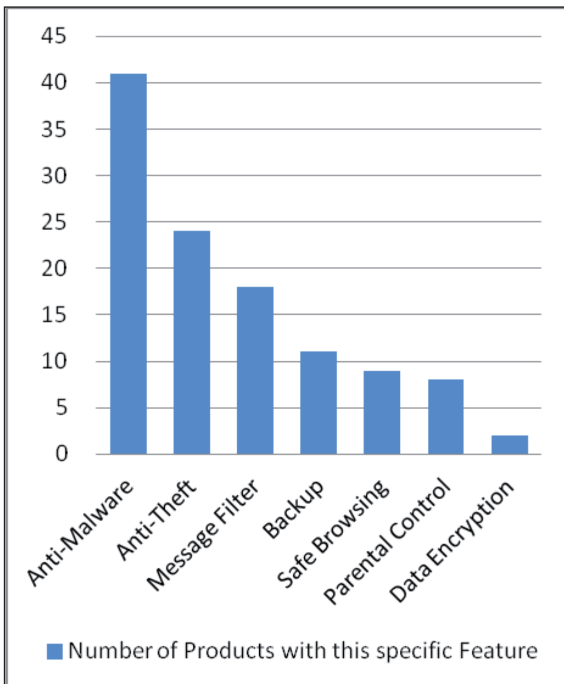

Figure 3: Features of 41 mobile security apps.

A current *Android* anti-malware test primarily contains detection rates, including false positive testing, and performance measurements. As described earlier, the on-demand and on-access detection rates may differ significantly. The on-access detection is what matters, so the on-demand detection test should only be used to reduce the test set for the on-access tests.

ON-DEMAND DETECTION

Testing the on-demand detection means that the tester explicitly starts a scan. Usually a scan can be configured to scan installed apps only or the complete system, which may include the SD card. A set of malware samples can be copied to the device, usually somewhere on the SD card to simplify testing. Then a full system scan is started. All detected files must be deleted and all remaining files will be saved as 'scan report'. Most apps we have encountered don't provide a function to save a scan report. Some apps create *SQLite* databases to store their scan results, but these files usually can't be accessed without root privileges. There are also apps which can't be configured to automatically delete all detected malware samples or to properly report them. In those cases we would have to deal with each and every detection one by one. For those apps we decided to skip the on-demand scan, because we wouldn't be faster than with the on-access test.

To copy samples to and from the emulator or device, use the following adb commands:

```
$: adb push <source> /sdcard/samples
$: adb pull /sdcard/samples <dest>
```

For many products the on-demand detection test doesn't cover the full detection capabilities, either because they don't scan the SD card or because they offer no on-demand scan at all. We use the on-demand test primarily to reduce the test set for the on-access test, which is obligatory to determine the real detection rates. Samples which have been detected by the on-demand scan don't have to be tested on access again, thus saving a lot of time.

ON-ACCESS DETECTION

An on-access test on *Android* is similar to a real-world test on a PC. It simulates the installation of a malicious app and determines whether the mobile security protects the user during the installation process. It requires the tester to choose an action when a malicious app has been detected. If a sample wasn't detected, it should be removed by the tester manually. Otherwise the device has to be restored, which is very time consuming. The samples are installed via adb. This approach doesn't reflect real user behaviour because real users wouldn't use adb, they would open the *Google Play Store* app to install new apps. As we can't guarantee to find malicious apps in the *Google Play Store* during the time of the test due to their limited lifetime in the store, the installation via adb is preferred to perform comparative reviews. One might argue that there is a problem with this installation process: anti-malware apps may use the installation source information as a hint for a malicious app or alternatively for false positive prevention. Such apps might get fewer detections or trigger more false positives. However, the installation source information isn't available through the *Android* API, thus it shouldn't matter whether a sample was installed with adb or via the *Google Play Store*. An anti-malware app might query the *Google Play Store* itself to gain information about the installed app, but that works independently of our installation method.

To install and remove samples with the emulator or device, the following commands can be used:

```
$: adb install <apk-file>
$: adb uninstall <package-name>
```

The package name can be obtained using this command:

```
$: aapt dump badging <apk-file>
```

Those three commands allow us to write a shell script which automatically iterates over all samples in a directory, installing and removing each sample one by one. The tester has to enter the detection result after each sample. An advanced way to automate such a test is given in the automation chapter.

FALSE POSITIVE TESTING

Whenever a detection rate test is performed, a false positive test is also important. The importance of such tests isn't that high yet on the *Android* platform. Due to the relatively small number of *Android* malware and their rather simple composition it's easy for security apps to detect them reliably once they know them. This limits the dangers of false positives. Nevertheless, it's possible to implement a false positive test.

The most important questions are the size of the test set and which apps to use for the false positive testing. In fact, the number of samples is just limited to the number of apps which can be installed on a device. The latest devices have enough internal memory to install hundreds of apps. The samples can be chosen from the top free apps in *Google Play*.

The tester can use the *Google Play* website to easily install all chosen apps on the device with the tested anti-malware product. This procedure covers the on-access detection. After all apps have been installed, a full system scan can be started to cover the on-demand detection.

Using only free apps is of course a limitation, as a normal user will most likely have free as well as paid apps. Covering this isn't really feasible for testers, especially when testing several dozen security apps vs. hundreds of clean apps for false positive testing purposes.

PERFORMANCE

App developers have to use the available resources economically on mobile devices. High CPU load may lead to a bad user experience as well as faster discharge of the battery. That means that an anti-malware app should avoid a long scan time and it should reduce background operations to a minimum.

Measuring the scan time isn't comparable because not all anti-malware apps allow the scope of a scan to be defined. Many scanners just scan the installed apps while others scan the SD card contents, too. The latter would require more time to perform the scan independent of their performance.

A direct approach would be to determine the power consumption during a scan or during idle times. This would require a comparable test set-up among all products, which is impossible to achieve. The battery would have to be in exactly the same state for all products, the background operations would have to be the same and there should be no influences from mobile networks etc. So you can never measure the impact of the security app only, you will always measure side effects as well. Therefore we opted for the following approach. We determined the resources used by the scan engine, which is an indicator of the impact on the battery life. A scan engine mainly uses the CPU to scan files and a network connection to query the cloud. An efficient scan engine requires fewer CPU cycles and generates less traffic than a heavier scan engine.

You should also consider that the malware scanner isn't the only feature which consumes battery power. If you talk about performance and impact on the battery life, you should always mention the supported features of the product and important configuration settings, e.g. how often are signature updates and full system scans scheduled?

The actual measurement of the CPU time consumption and traffic works as follows:

- Install a set of N regular apps
- Measure the traffic and consumed CPU time through the /proc file system on the device after each installation
- Repeat the procedure several times to build an average.

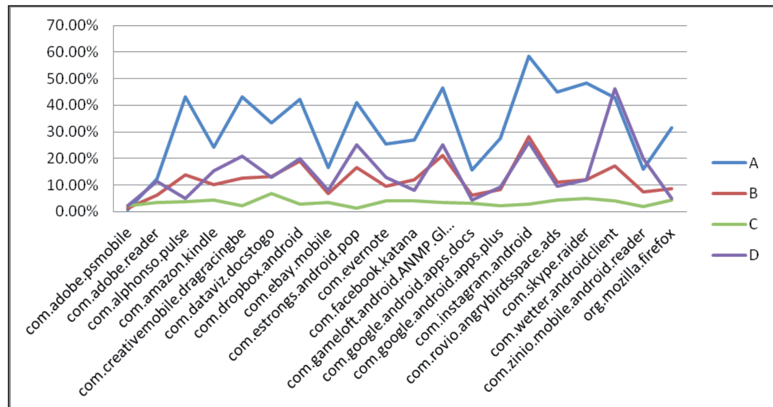

Figure 4: CPU usage analysis chart.

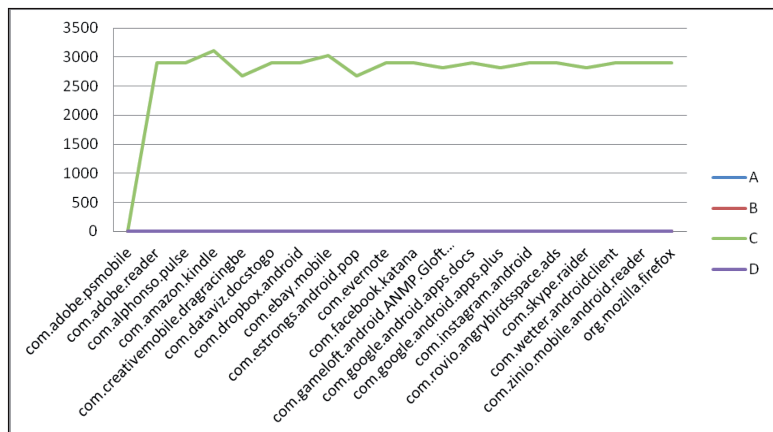

Figure 5: Traffic in bytes.

A sample analysis of the obtained data could look like Figure 4.

Figure 4 shows the CPU usage of four anti-malware products during the installation of 20 regular apps. As we can see, product C has a constant scan time. This might be the result of its cloud-only scan engine. The cost for a cloud query can be considered as constant. As our monitoring tool also measures the traffic per process, we can verify our assumption (Figure 5).

The traffic chart confirms our guess: product C is the only product which queries the cloud during its on-access scan.

So which product consumes the most battery power? The answer isn't that easy. Regarding the CPU usage, we could say that product A consumes most battery power as it requires most CPU time. But if we also consider the traffic, then product C would be a good candidate for consumption of the most battery power. As this also depends on the built-in Wi-Fi chipset, we're not able to clearly identify a 'winner' in this test.

BUILDING A MALWARE COLLECTION

There are some conditions which should be fulfilled by a malware collection in general. These conditions have been defined and discussed by AMTSO and apply for *Android* malware as well as for *Windows* malware:

- The samples must be validated.
- The age of samples and/or the age of their sources (in case of URL, domains as test objects) need to be taken into consideration.
- The samples in a test set are diverse and comprise a sufficiently large variety of files.
- Prevalence is important.

List 1: Collection qualities according to AMTSO [1].

A malware collection has to cover a broad range of prevalent malware families. A family-based analysis of detection results showed that even a product with very good detection rates can miss one or another malware family entirely. Therefore the malware set should cover all current prevalent malware families. The maximum size of the set should be limited by a maximum number of similar samples, e.g. samples with the same *Android* package name. Please consider that only an on-access test can provide meaningful detection results, so the bigger the sample set the more time is required to complete the test (Figure 6).

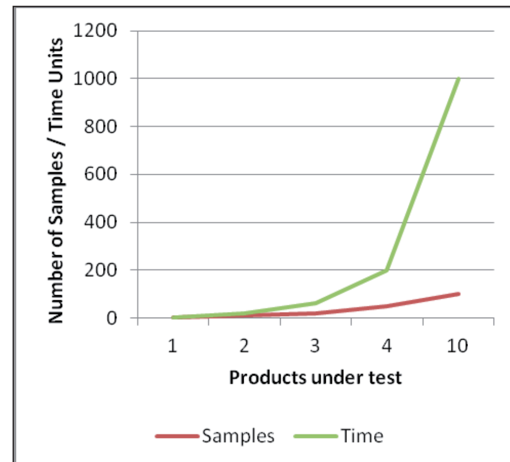

Figure 6: Dependency of number of samples and total test time.

Now we have a conception of the size of our collection. The next question is which file types must be included? As mentioned before, we have to perform an on-access test to gain a neat conclusion of the malware detection rates. So the file type of our malware set is limited to APK files. For pure on-demand tests you could also consider scanning Dalvik binaries (DEX) or native code, but that isn't recommended because Dalvik binaries can't be installed and executed by a user. The APK files need to be validated to be executable and verified to be malicious. A convenient way to validate APK files is to try to install the APK files on a test device or emulator. If the installation fails, the sample isn't suitable for the test, because it's not working and won't be scanned by the on-access scanner. The on-access scanner usually uses a broadcast receiver to listen for the 'PACKAGE_ADDED' action. When an installation fails, the broadcast for this file will not be sent. Corrupted signatures are the simplest case for an unsuitable APK file. The Android Package Manager requires valid signatures to proceed with the installation. You can use the adb install and adb uninstall commands to write a validation script for your sample set. adb install outputs whether the installation was successful or not.

To verify whether the samples are malicious or not, you can use static and dynamic analysis methods. Static analysis methods may include code review and analysis of the AndroidManifest file. Because *Android* apps are written in Java, they can be easily disassembled and decompiled. Tools to be mentioned here are *dex2jar* [2], which converts Dalvik binaries to Java archives, and *Androguard* [3], a powerful reverse engineering toolkit. Dynamic analysis can be done with *DroidBox* [4], which allows apps to be monitored in the emulator.

AUTOMATION OF TESTS

On-demand detection

Many mobile security apps use their own activity for the on-demand scan. Such activities can be directly started from the command line:

```
$: adb shell am start <intent>
```

The intent value can be obtained through the debug log:

```
$: adb logcat
```

With the information given in the debug log we can now compose our command line to start the activity:

```
$: adb shell am start -n com.avast.android.
mobilesecurity/.app.scanner.ScannerScanActivity
```

In this case, the scan starts automatically with the activity. No further user interaction is required. Starting a specific activity directly from the command line may require special permissions on a real device, so this approach may work only in an emulator where we have root privileges.

With this method it's not possible to automate a complete on-demand detection test, but it may help in a lab environment.

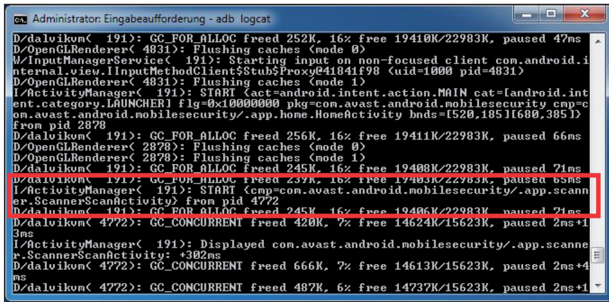

Figure 7: A manual scan was started with its own activity.

On-access detection

A simple approach for semi-automation was given in the description of the test scenario. A more advanced version would include automated screenshots of the notifications and management of multiple devices which are tested simultaneously.

Figure 8 shows how such an application could look, depicting the *AV-TEST Android* testing environment. The program installs all samples of the test set one by one. If multiple devices are used for the test, each sample will be tested on all devices before the next sample is tested. The user just has to decide whether the sample was detected or not. After the user has made his decision, a screenshot is automatically taken and the sample is uninstalled.

The user interface allows for the selection of one or more devices which are connected to the PC. The table shows the samples to be tested and the respective results per device (1 = detected, 0 = not detected). The total number of samples in the test and the total number of detected samples per device are shown in the first row. Log messages are shown at the bottom.

Automated decisions, whether a sample was detected or not, could be obtained through observation of the debug log.

You can try to implement such automation in a shell script with the command line tools provided by the SDK or you can have a look at the ddmlib.jar Java library, which is also supplied by the SDK. This library includes a high-level API to control the ADB.

Other automation approaches

If you need more complex automation techniques you should have a look at *Robotium* [5]. With *Robotium* you have full

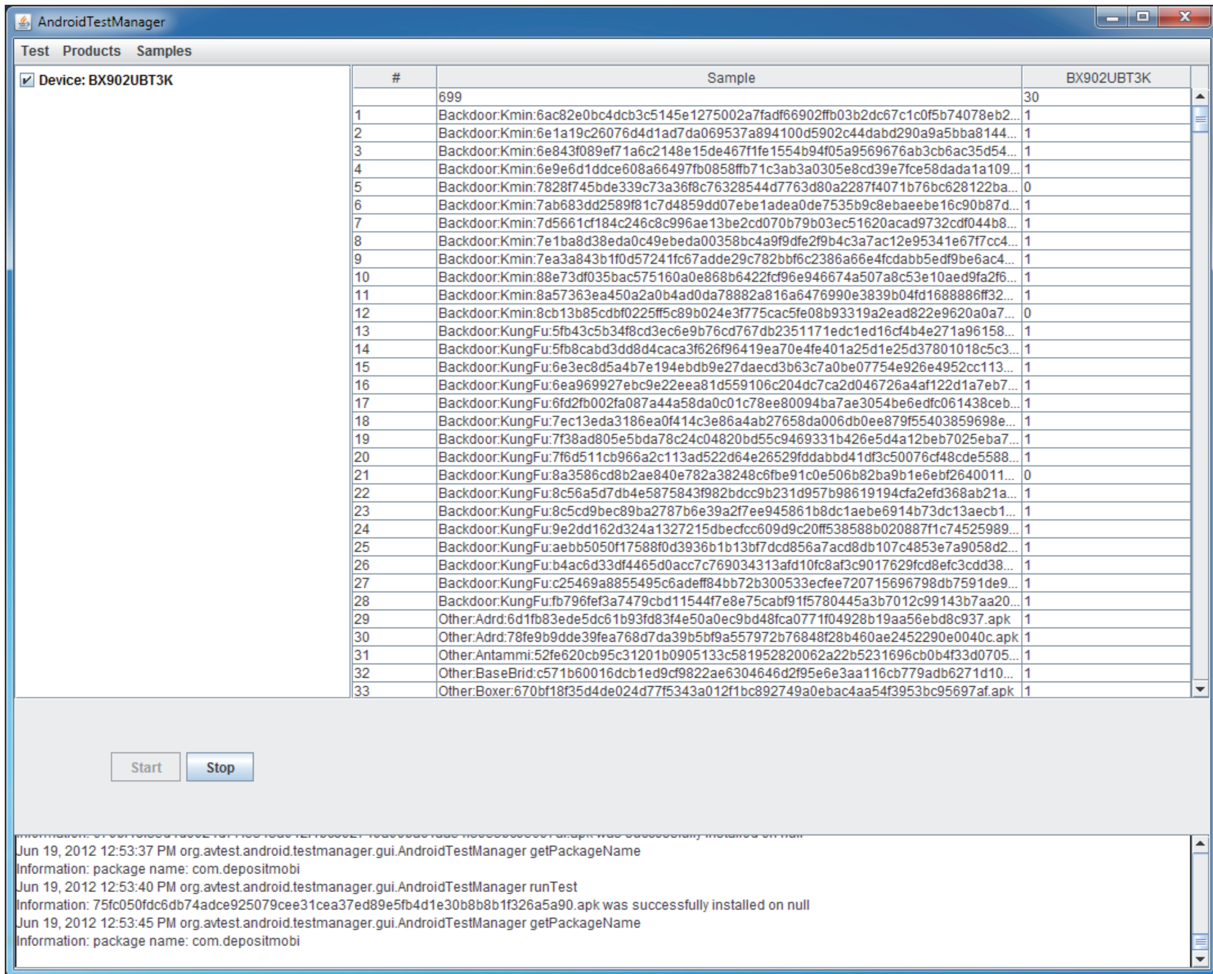

Figure 8: A self-developed application to perform on-access tests.

control over the GUI of an app, but it requires that the automated app is signed with the same key as the 'controller app'. If you have the source code of the app you want to automate, signing is no problem. Otherwise you have to re-sign the app under test, e.g. with the debug key [6].

Automation conclusion

There are several ways to perform simple and advanced automation. The *Robotium* framework is perfect for your development team, but you shouldn't use it for public tests, because the APK files have to be re-signed. The automation of the time-consuming on-access tests is very useful, but you should never run such automation unattended, as with an unattended test run you have to verify each result in your report afterwards.

SUMMARY

Anti-malware tests on *Android* don't seem to be too difficult at first sight. Due to the small threat landscape and limitations of the devices and OS there aren't that many test scenarios. The *Android* system itself was designed with security in mind. Each app runs in its own sandbox; even the mobile security apps have only limited access to the system. This also reduces the number of malware-related features that are included, e.g. a safe browsing feature is included in fewer than 25% of the available mobile security apps. With a rising number of malware targeted at mobile devices the support of such features may increase. New testing methodologies will be introduced accordingly. The tricky parts are the things that are done differently from the way they are on the desktop and the big differences among the products, which require a strict testing methodology that will cover all apps equally. There are still some open questions regarding testing methodologies and best practices. We will have to see how the threats evolve and which security features will remain in the products.

ACKNOWLEDGEMENTS

I'd like to thank our CTO Maik Morgenstern and our CEO Andreas Marx for their feedback and comments.

REFERENCES

[1] AMTSO: Sample Selection for Testing. http://www.amtso.org/amtso--download--sample-selection-for-testing.html.

[2] dex2jar: Tools to work with android .dex and java .class files. http://code.google.com/p/dex2jar/.

[3] Androguard: Reverse engineering, Malware and goodware analysis for Android applications. http://code.google.com/p/androguard/.

[4] DroidBox: Android Application Sandbox. http://code.google.com/p/droidbox/.

[5] Robotium: It's like Selenium, but for Android. http://code.google.com/p/robotium/.

[6] Robotium: Test Android apk file with Robotium. http://robotium.googlecode.com/files/TestAndroidapkfileUsingRobotium.pdf.