Scalable Testing of Mobile Antivirus Apps

Andrea Valdi, Eros Lever, Simone Benefico
{andrea.valdi,eros.lever,simone.benefico}@mail.polimi.it

Davide Quarta, Stefano Zanero, Federico Maggi
Politecnico di Milano
{davide.quarta,stefano.zanero,federico.maggi}@polimi.it

Scientific evaluation of antivirus applications is a long-debated problem. Essentially, reproducing the same exact conditions an antivirus meets when running on a user's device presents conflicting requirements, which in the mobile world are exacerbated even further.

We implemented AndroTotal, a scalable system to conduct mobile antivirus evaluations with a rigorous approach. It is a publicly available service and open research dataset that we operate since early 2013. We describe the conceptual and technical challenges that we faced in its design and implementation, and compare it against existing approaches. Furthermore, we report our experience after one year of operation and data collection.

Keywords: Mobile malware; Antivirus testing, Android.

1 The Rise of Mobile Malware

Mobile devices are becoming more and more ubiquitous. A leading role is played by Google's Android mobile operating system, which holds 75% of the market [IDC, 2013]. The Google Play App Store reached a total of 50B downloads in July 2013, with over 1M applications [Welch, 2013], not to mention the tens of unofficial, alternative markets (e.g., Amazon, Samsung Apps, AndroLibs, AppBrain), which offer exclusive apps, or apps banned from Google Play Store.

For most people, mobile devices have become their digital wallet, which holds sensitive, personal and business information. This makes mobile devices attractive for cyber criminals, whose current tactics include the development and distribution of malicious applications, with the final goal of stealing sensitive data, or performing fraudulent monetary transactions.

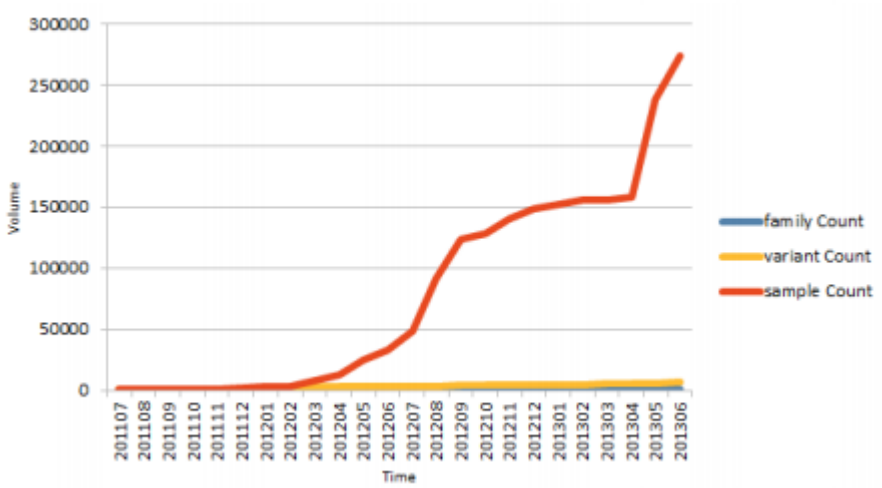

Figure 1: Android malware growth as seen in the Symantec sample collection (drawn from [Uscilowski, 2013]).

All threat reports in the past two years (e.g., [Uscilowski, 2013]) agree that the amount of malware samples (Figure 1), along with their diversity and complexity [Lab, 2012], have been growing at an alarming rate. As a result, for the past two years, mobile security has been a very active research and development topic.

The security measures currently provided by the Android system include code signing and verification of the signature upon installation. Moreover, the Google App Verify sends the hash of each installed application to Google servers: in case the hash corresponds to a known malware, the user is warned. Furthermore, if making use of the official Google Play Store, developer verification is enforced, and a system called Google Bouncer provides a first screening of submitted applications. This has been shown, however, to be easily circumvented [Poeplau et al., 2014].

In addition to these security measures, antivirus (AV) apps play a key role in providing client-side protection for mobile devices. Between late 2013 and early 2014 we found approximately 100 AV apps in the Google Play App Store (approximately, because some of them were available only for a limited period of time). According to our estimates, approximately one half of the Android user population installed an AV app, as Figure 2 shows. We considered the public download counts of the top 20 AV apps as of late 2013. More interestingly, approximately 70% of the AV apps we surveyed are Android-specific, meaning that no desktop equivalent app exists. Thus, in addition to established vendors, the majority of apps that hit the market are from new players.

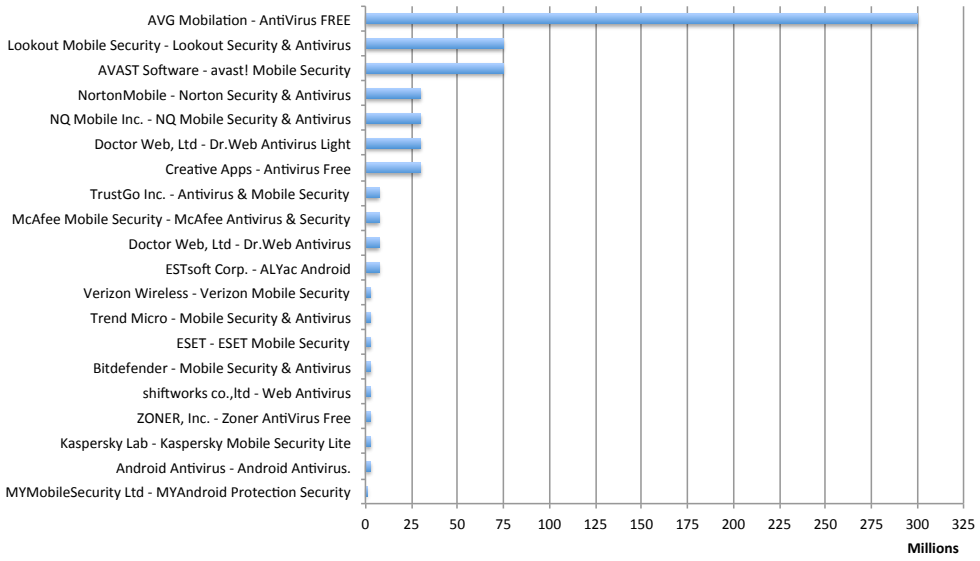

Figure 2: Top 20 Android antivirus products by average installations count on the Google Play Store.

2 Mobile Antivirus Testing Challenges

A natural question is how effective mobile AV apps are. Thorough, scientific evaluation of security products, particularly of AVs, is a long-debated and confusing problem.

2.1 Conceptual Challenges

The essence of the problem is that it is very difficult to conduct such a test under the same exact conditions (e.g., network connectivity, operating system version, system load) of the AV running on the user's device. This problem becomes technically challenging if the tests need to scale. The most common approach for desktop products consists of creating scripts that automatize command-line versions of the AVs. Besides the fact that the command-line versions of the AVs are not as well maintained as their GUI companion, their inherent limitation is that detection relies solely on signature matching, and not, for instance, on behavioral heuristics, or on techniques triggered by network communication or inter-process communication anomalies. In other words, not only this approach does not consider the AV's working environment, it does not even exercise the entire AV. Indeed, using this strategy for AV testing has been heavily criticized [Harley, 2012], because the test does not reflect the actual AV capabilities in real-world conditions. The authors of VirusTotal, which implements the aforementioned approach, carefully point out this limitation[1].

[1] `https://www.virustotal.com/en/about/best-practices/`

In the mobile world, these same challenges apply. Behavioral heuristics are in use in several products (e.g., SRTAppScan), and some products analyze network traffic anomalies (notably, e.g., Kaspersky). Another remarkable example of why a completely different approach is needed for mobile AVs are those anti-viruses (e.g., Trendmicro, SourceFire) that use in-the-cloud scanning (i.e. they verify the samples not against a local database only, but by interrogating a remote service).

2.2 Technical Challenges

Mobile AVs, with their highly interactive GUIs, are technically difficult to automatize. For instance, Figure 3 shows the gestures needed to perform a device scan with the Zoner antivirus. As basic as it seems, automating these steps requires emulating tapping operations programmatically, and waiting for the results to be displayed. This is even harder when dealing with custom view components or other custom graphical elements not provided by the Android SDK itself. Another challenge is that AVs work in multiple detection modes, which must be supported. The two most popular ones in mobile AVs are *on demand*, where the user requests a device scan, and *on install*, where the AV waits for an APK to be installed (i.e., registers a broadcast receiver for the `ACTION_PACKAGE_ADDED` or `ACTION_PACKAGE_INSTALL` intents) and analyzes it on the spot. Another problem is how to capture detection results. The Android GUI is asynchronous, which makes this operation technically complex: For instance, many AVs use the notification bar. Unfortunately, it is not natively allowed to access those notifications by other applications.

Some of the above challenges could be solved by modifying the Android framework or the AV application. However, this would not meet the requirement of preserving the original working conditions as much as possible.

2.3 Existing Approaches

Researchers, practitioners and vendors have proposed methodologies and published several reports in this direction (e.g., `http://www.av-comparatives.org/mobile-security/`).

Unfortunately, all the proposed methodologies require time-consuming manual actions, tedious tasks such as preparing a clean testing image (about 2–3 minutes each), installing the AV and the suspicious application (about 2–5 minutes), restoring a clean state of the system (up to 10–15 minutes). We estimate that such approaches would allow to scan with a single AV up to 25-30 applications per man/day of effort. Although some mechanisms can automate the creation of the clean testing image and the installation of the suspicious applications, even state-of-the-art tools (i.e., [Pilz, 2012, Ramachandran et al., 2012]) still require an operator to check the outcome of each single scan, which significantly limits scalability.

Other researchers concentrated on complementary aspects of the AV-testing problem. For instance, [Zheng et al., 2012] focus on the resiliency to malware mutations, by applying multiple transformations (e.g., repackaging or obfuscation) to the samples before testing

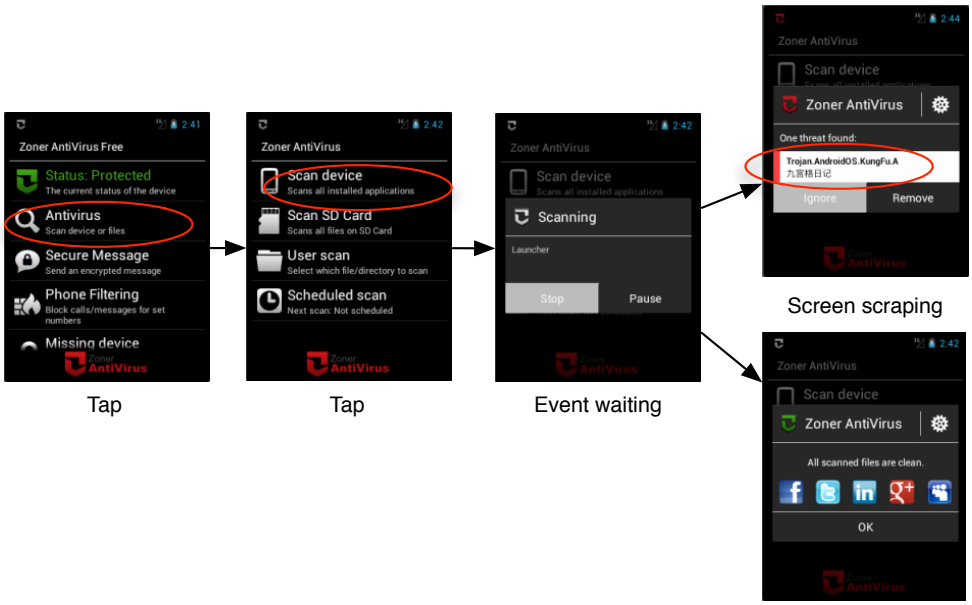

Figure 3: User interaction needed to perform an on-demand device scan with Zoner AntiVirus Free 1.7.0.

them with AVs. While the work is very interesting, the authors choose not to focus on ensuring that the tests are run under reproducible conditions, which we believe is an essential aspect for the scientific AV comparison tool we want to build.

3 An Approach and Platform for Mobile Antivirus Testing

Considering the variety and the complexity of mobile malware, and the above challenges, in this paper (an early version of which appeared in [Maggi et al., 2013]) we present a generic approach for mobile AV testing, we implement it in a tool, openly accessible as a web service, that allow researchers to conduct thorough, precise and repeatable experiment to efficiently test Android AV apps. In addition to [Maggi et al., 2013], after one year of operation of our AndroTotal service, we describe the dataset we are collecting (which is made available to other researchers) and discuss several quantitative measurements and observations we performed.

As motivated in Section 2, conducting scientific evaluation of (mobile) AVs requires full emulation of the entire application stack. To solve this conceptual challenge, AndroTotal creates reproducible, self-contained testing environments for each AV-malware pair, while ensuring a high throughput thanks to its inherent scalability. To increase scalability, previous work proposed to use multiple AV apps against multiple samples by installing a set of AV apps and samples in a single emulator instance or device at the same time. However,

we do not follow this approach because it may lead to unwanted interactions both between the AVs, and between the samples, which could result in biased outcomes. Instead, our approach is that of abstracting the environment preparation and testing procedure, so that tests can be created programmatically and can scale by simply adding new hardware. `AndroTotal` does not require any software or hardware modification, and works on both emulated and physical hardware.

`AndroTotal` supports both on-demand and on-install detection modes, exposing a Python API to automatize test procedures and to gather results via GUI scraping. The key contribution is that AndroTotal *runs* the actual AV in a real or emulated Android device. During each test, AndroTotal also captures screenshots from the application, the log file produced, and the network dump. Note that we provide the log file as an additional result, but we do not rely on it to derive the detection label because not all mobile AVs log that information. A GUI-, screen-scraping-based approach is more general, flexible and less constrained than a log parser.

As additional information sources, AndroTotal gives the user access to the analysis (if any) generated by VirusTotal, CopperDroid, ForeSafe and SandDroid. We have set up data sharing agreements with the first two services mentioned.

To tackle both the conceptual and technical challenges highlighted in Section 2, we need to reproduce the actions of a user that interacts with the AV GUI. Considering our goals, we derived the following requirements, which AndroTotal fulfills entirely:

Req 1 (User Input Simulation). We must be able to stimulate the device GUI by reproducing the typical gestures a user would perform when using an AV.

Req 2 (User Interface Feedback). We must be able to obtain a feedback about the displayed views and activities on a device. This is needed to synchronize the testing procedures with the state of the running AV. We must also be able to scrape information from the display in order to retrieve possible useful information (e.g., name of an identified threat).

Req 3 (Multi-application Testing). We must support testing procedures over more multiple applications. This is needed not only to develop complex testing procedures, which may involve more than one application (e.g., AV and browser), but also for basic operations (e.g., notification management), which still require accessing different Android application contexts.

Req 3 (Avoiding Antivirus Tampering). Any modification to the AV package may alter their true behavior, which in turn may bias the results of our tests. For this reason, we must avoid modifying the AV package by injecting new code, changing its signature, or repackaging it.

Req 4 (Support any Android Version). We must support any Android version.

Req 5 (Support for Android Notifications). We should (natively) support Android notifications operations (e.g., waiting for notification to appear, open notification bar, checking for notification existence), because some AV apps rely on notifications as the only feedback.

4 Approach Implementation

As described in Section 4.1, based on the above requirements, we implemented AndroPilot, the first pure-Python Android AV automation library that supports all the functionalities needed to conduct AV tests. As described in Section 4.2, we used AndroPilot as the basic block for creating AndroTotal's scalable architecture.

4.1 Test Automation Implementation

We analyzed the six most-promising, publicly available libraries and, as discussed in depth in our previous work [Maggi et al., 2013], we found out that none of them was suitable. In the remainder of this section, we summarize the essential points of Table 1 from [Maggi et al., 2013]. Robotium, while excelling in white-box testing, it presents some shortcomings due to the need for application resigning and application sandboxing limitations (**Req 3** and **4** not met). Android provides monkey and monkeyrunner, which perfectly meet all requirements but they do not support retrieving data from a running device or emulator (**Req 2** not met). Android UI Automator is the most suitable library, but it only support Android SDK API 16 or higher (**Req 4** not met). AndroidViewClient and Apk-view-tracer are a good trade off, as they both effectively simulate typical user inputs and can retrieve information about displayed activities. Under the hood, they rely on on monkeyrunner and monkey. They support all the existing Android versions and do not need any package modifications to interact with an application. Unfortunately, we noticed that the notification support, as well as most of the other implemented functionalities, were unstable and slow.

Therefore, we extended Apk-view-tracer and obtained AndroPilot: We re-implemented part of the existing code to improve its stability, fixing several bugs and suboptimal design choices. More specifically, by further leveraging the Android ViewServer component we introduced new procedures to properly manage application synchronization during testing stages, including functions that wait for an arbitrary view, text or notification to appear on the screen. We improved the view management to correctly report when a view is shown on the running Android instance and implemented a new function to retrieve the screenshot from an attached device or emulator. Overall, we improved the code stability and enhanced the library speed.

Developing a new AndroPilot adapter module for an AV is simple. The coding effort is low, as we use a high-level language (Python), and our libraries make it possible to implement an adapter in just 36 lines of code on average (as measured with CLOC (`http://cloc.sourceforge.net/`) on 10 adapters currently implemented). Two examples of such implementation are as follows:

```
class TestSuiteProductX(base.BaseTestSuite):
    def detection_on_demand(self, sample_path):
        """ Test the AV's capability of detecting malware by scanning
        the whole device.
        """
        # Connect to a running device
        p = self.pilot
        # Install application sample
        p.install_package(sample_path)

    def detection_on_install(self, sample_path):
        """Test the AV's capability of detecting malware upon installation.
        """
        p = self.pilot
        p.install_package(sample_path)
```

```
class TestSuiteProductY(base.BaseTestSuite):
    """ Test suite for ProductY
    """
    def detection_on_install(self, sample_path):
        """Test the AV's capability of detecting malware upon installation.
        """
        # Connect to a running device
        p = self.pilot
        if sample_path: # Install sample on the running device
            p.install_package(sample_path)

        time.sleep(2)  # Sleep
        self.__check_popup()  # Check if detected

    def detection_on_copy(self, sample_path):
        """ Test the AV's capability of detecting a malware when the
        sample is copied on the device.
        """
        p = self.pilot
        if sample_path:
            p.push_file(sample_path)

        time.sleep(2)
        self.__check_popup()

    def __check_popup(self):
        """ Check if the alert popup is displayed
        """
        p = self.pilot
        # Wait for the antivirus's screen to appear
        if p.wait_for_activity(
                "com.kms.free.antivirus.gui.AppCheckerAlert", 10):
            # Tap on button to start scan
            p.tap_on_coordinates(120, 210)

            if p.wait_for_activity("com.kms.free.antivirus.gui.AppCheckerVirusAlert",
                                   30, critical=False):
                p.refresh()
                threat_view = p.get_view_by_id("ObjectType")
                # Extract the threat name
                self.result['detected_threat'] = threat_view.mText.strip()
            else: # No threat found
```

```
            self.result['detected_threat'] = config.NO_THREAT_FOUND
    else: # No threat found
        self.result['detected_threat'] = config.NO_THREAT_FOUND
```

4.2 Architecture Implementation and Deployment

In the high-level workflow of AndroTotal a user submits an Android application (APK package) to the web interface of AndroTotal. If the application has not been analyzed yet, it is pushed into the analysis queue. Whenever a worker is available, an emulator with a clean image is started, the application sample is installed and the required tests are performed. A test is basically a Python script written on top of AndroPilot, which runs a given AV in either on-demand or on-install mode, and retrieves the results via GUI scraping. The results are then stored in the database and returned to the user. AndroTotal also exposes a REST API, which ensures interoperability with external services.

AndroTotal's scalability is ensured by the fact that each testing procedure is self contained. In this way, each test job is taken and analyzed by a single worker which starts an Android emulator, installs the needed packages, performs the required tests and stores the results into a database.

By leveraging the snapshot functionality of the Android emulator, AndroTotal can run a test in 1–3 minutes on average. Each test requires about 1GB of temporary disk space, to store the snapshot image, which is de-allocated once the test terminates. The time needed to scan an application or malware sample against a set of AV apps grows linearly with the number of AV apps. This ensures linear scalability through parallelization.

Differently from similar services, such as VirusTotal, AndroTotal maintains multiple versions of the same AV over time. This allows testing new samples against older AV versions, and computing evolution statistics. Accessing the scan results allows users to visualize the data associated with each AV test (i.e., logcat dump, network dump, screenshots) and to download it. By aggregating data from various reports, AndroTotal can also give an insight of how a sample is likely to be malicious.

A daily procedure monitors the Google Play Store for new AV releases. When one is found, AndroTotal alerts the maintainer, who will take care of initializing the clean image of the new AV release, plugging it to the AndroTotal system. This task is asynchronous and does not affect the throughput of AndroTotal.

5 Evaluation

The most important evaluation was the feedback that we obtained from the community. As of now, 2234 users have requested access and are actively using AndroTotal and allowed us to collect 35209 distinct samples of malicious and benign Android applications. We were contacted by five major antivirus vendors to access the dataset and the service, besides a number of researchers and groups.

Full Emulation vs. Partial Emulation A successful use case of our service by a third party is described by Poeplau et al. [2014], who used both VirusTotal and AndroTotal to evaluate the detection rate of mobile AVs against an advanced proof-of-concept malware that contain call-home functionalities to download additional malicious code after the first execution. None of the AV tests executed by VirusTotal triggered the call-home functionality, demonstrating that VirusTotal's testing environment did not create the necessary conditions for the malware to work, which in turns mean that the AVs were not tested under realistic conditions. The authors confirm that during AndroTotal tests, their malware was able to complete its procedure as if working on the real device.

Mobile-desktop AV Mismatch The aforementioned example shows how sophisticated types of malware are simply not compatible with the approach adopted by VirusTotal. The same can be said, in turn, for AV apps which adopt in-the-cloud or behavioral approaches, as we mentioned earlier.

For these reasons, a direct, side-by-side comparison between the two approaches does not make a lot of sense. However, we automatically and manually compared the analysis of VirusTotal (which supports desktop AVs) and AndroTotal (which supports mobile AVs) on 300 randomly chosen malware samples, recognized by both. We then focused on the set of five vendors in common (which we will anonymize as V1–V5).

For each sample and for each vendor, we calculated the edit distance between the detection label obtained by VirusTotal (e.g., `INI:SMSSend-A [Trj]`) and the detection label obtained by AndroTotal (e.g., `Android:FakeNotify-A [Trj]`). Although one may expect that the same vendor labels the same samples consistently, we found out that this is not the case for the majority of the vendors (validating similar results in the desktop applications, reported in [Maggi et al., 2011]). Indeed, as Figure 4 shows, only one vendor achieves (almost) zero discrepancy. The discrepancy is calculated by dividing the edit distance of each comparison for the length of the longest of the two labels, so to obtain a number in $[0, 1]$.

In addition to showing the intra-vendor discrepancies problem, this experiments demonstrate once again the need for full-emulation AV testing approaches.

Time Requirements A worker takes between 30 seconds and 3–4 minutes to run a test on 10 malware detectors, including the time required to launch all the emulators. On average a single test takes 1–3 minutes to complete, with a wide standard deviation that does not allow us to firmly state that there are vendors that are considerably slower or faster than others.

Table 1 shows the most popular threat labels detected, where the popularity is assumed to be proportional to the number of distinct (i.e., in terms of MD5) sample APKs uploaded with the that label. We obtained this table by querying the AndroTotal database for the number of tests of each APKs, grouped by the output label. As it can be seen, several of these labeled threats are Adware (i.e., applications aggressively pushing ads to users) as

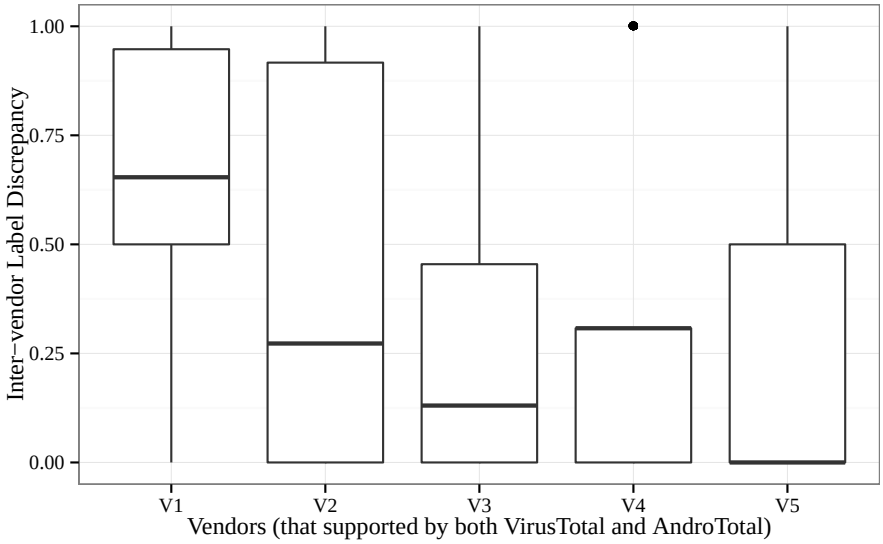

Figure 4: Mobile-desktop AV mismatch. We calculated the discrepancy of the mobile vs. desktop product of the same vendor. The discrepancy is calculated, for each detected sample, by calculating the edit distance of the detection labels. This is mapped to a number in $[0, 1]$ by dividing the edit distance by the lenght of the longest label. Only one vendor shows zero discrepancy.

opposed to general malicious applications. Where the line is drawn for this class of apps is difficult to define, as discussed e.g. in [LookOut, 2012]. This is an expanding gray area which we would like to investigate more in the future.

6 Future Challenges

AndroTotal works on physical devices and already allowed us to start working on measuring the AV battery consumption. However, performing tests on physical mobile devices presents some significant challenges. Creating self-contained, repeatable tests on physical hardware is expensive in terms of the amount of time required for "freezing" the state and restoring the same testing conditions. Reflashing a chosen NAND partition on an Android device can take a time ranging from 2 to 10 minutes depending on the device model, the partition to reflash, and the need to reconfigure the system after flashing has completed. We plan to overcome these challenges through use of the recent hardware virtualization support of ARM processors. In the long run, AndroTotal will thus be able to give us information about accurate performance evaluations of the AV on real devices.

Moreover, AndroPilot exhibits some slight performance limitations while retrieving the displayed view tree, taking up to 40 seconds for a complete screen dump in some extreme, rare cases. This delay is due to Android's ViewServer component, and we are

Label	#APKs
UDS:DangerousObject.Multi.Generic	6372
HEUR:Trojan-SMS.AndroidOS.Opfake.bo	2157
Adware.Airpush.origin.7	2138
AndroidOS_Opfake.CTD	938
Android.SmsSend.origin.281	916
Android:FakeNotify-A [Trj]	907
Adware.AndroidOS.Airpush-Gen	892
HEUR:Trojan-SMS.AndroidOS.Opfake.a	809
HEUR:Trojan-SMS.AndroidOS.FakeInst.a	786
Android.SmsSend.origin.629	664
Android.SmsSend.origin.315	637
AndroidOS_Leadblt.HRY	584
HEUR:Trojan.AndroidOS.Plangton.a	570
Adware.Startapp.origin.8	503
not a virus Adware.Startapp.origin.0	493

Table 1: Top 15 labels overall as of December 23rd, 2013

currently testing the stability of patching it (as shown in `http://code.google.com/p/android-app-testing-patches/`), thus obtaining a 20–40× speedup.

Thanks to its modular architecture, AndroTotal could also be a valuable starting point to build a more generic testing framework for mobile applications, beyond the specific scope of mobile antivirus applications. Mobile applications testing is a growing field and having an independently developed tool available to researchers, in contrast to the commercial and closed solutions available, to execute repeatable tests at scale, simplifying application validation and verification, can definitely be a big opportunity to investigate on.

7 Conclusions

AndroTotal is a publicly-accessible web application that allows any user to submit an Android application and check how a set of commercial mobile antivirus products classify it.

AndroTotal advances the state of the art by allowing scalable testing of antivirus products, whereas existing researches focused on human-assisted tests and reviews. AndroTotal can scan hundreds of suspicious applications per day against all of the major antivirus versions with a single machine running in a completely automated fashion, as opposed to approximately 25–30 application scans over a single antivirus, per man-day.

The research community reacted very positively to the release of AndroTotal. Several antivirus vendors are automating submission of samples to our system, and are retrieving samples for their own analysis. We also cooperate with other well known Android malware research projects such as CopperDroid and Andrubis.

Acknowledgments

The authors are grateful to the anonymous reviewers who pointed out weaknesses and significantly contributed to the improvement of this research. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement nr. 257007, as well as from the TENACE PRIN Project (nr. 20103P34XC) funded by the Italian Ministry of Education, University and Research.

References

David Harley. There's testing, then there's VirusTotal. `http://blog.isc2.org/isc2_blog/2012/12/theres-testing-then-theres-virustotal.html`, November 2012.

IDC. Android and ios combine for 92.3shipments in the first quarter while windows phone leapfrogs blackberry, according to idc. `http://www.idc.com/getdoc.jsp?containerId=prUS24108913`, 5 2013.

ESET Latin America Lab. Trends for 2013, Astounding growth of mobile malware. `http://go.eset.com/us/resources/white-papers/Trends_for_2013_preview.pdf`, November 2012.

LookOut. Uncovering Privacy Issues With Mobile App Advertising. `https://blog.lookout.com/blog/2012/07/09/mobile-privacy-app-advertising-guidelines/`, July 2012.

Federico Maggi, Andrea Bellini, Guido Salvaneschi, and Stefano Zanero. Finding non-trivial malware naming inconsistencies. In Sushil Jajodia and Chandan Mazumdar, editors, *Information Systems Security*, volume 7093 of *Lecture Notes in Computer Science*, pages 144–159. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-25559-5. doi: 10.1007/978-3-642-25560-1_10. URL `http://dx.doi.org/10.1007/978-3-642-25560-1_10`.

Federico Maggi, Andrea Valdi, and Stefano Zanero. Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '13, pages 49–54, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2491-5. doi: 10.1145/2516760.2516768. URL `http://doi.acm.org/10.1145/2516760.2516768`.

Hendrik Pilz. Building a Test Environment for Android Anti-Malware Tests. *AV-TEST Report*, October 2012.

Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *21st Annual Network and Distributed System Security Symposium*. The Internet Society, Feb 2014.

R Ramachandran, T Oh, and W Stackpole. Android Anti-Virus Analysis. In *Annual Symposium on Information Assurance (ASIA)*, 2012.

Bartlomiej Uscilowski. Mobile adware and malware analysis. Technical report, Symantec, October 2013. URL `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/madware_and_malware_analysis.pdf`.

Chris Welch. Google: Android app downloads have crossed 50 billion, over 1M apps in Play. `http://www.theverge.com/2013/7/24/4553010/google-50-billion-android-app-downloads-1m-apps-available`, July 2013.

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)*, April 2012.