AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors

Federico Maggi
fmaggi@elet.polimi.it
DEIB, Politecnico di Milano

Andrea Valdi
andrea.valdi@mail.polimi.it
DEIB, Politecnico di Milano

Stefano Zanero
zanero@elet.polimi.it
DEIB, Politecnico di Milano

ABSTRACT

Although there are controversial opinions regarding how large the mobile malware phenomenon is in terms of absolute numbers, hype aside, the amount of new Android malware variants is increasing. This trend is mainly due to the fact that, as it happened with traditional malware, the authors are striving to repackage, obfuscate, or otherwise transform the executable code of their malicious apps in order to evade mobile security apps. There are about 85 of these apps only on the official marketplace. However, it is not clear how effective they are. Indeed, the sandboxing mechanism of Android does not allow (security) apps to audit other apps.

We present AndroTotal, a publicly available tool, malware repository and research framework that aims at mitigating the above challenges, and allow researchers to automatically scan Android apps against an arbitrary set of malware detectors. We implemented AndroTotal and released it to the research community in April 2013. So far, we collected 18,758 distinct submitted samples and received the attention of several research groups (1,000 distinct accounts), who integrated their malware-analysis services with ours.

KEYWORDS

Android; Malware; Malware detectors; Testing.

CATEGORIES AND SUBJECT DESCRIPTORS

D.4.6 [**Security and Protection**]: Invasive software; D.2.5 [**Testing and Debugging**]: Testing tools

1. INTRODUCTION

With 75% market share, Android is the most popular platform for mobile devices according to the most recent surveys [6, 7]. Along with the number of Android devices, the prevalence of the official app market increased last year hitting 25 billion downloads in September 2012 [14].

The popularity of Android devices turned them into a very attractive target for cybercriminals [16]. Although various types of threats against Android devices have been spotted in the wild, the most popular one consists of malicious apps distributed through both official and unofficial markets. These apps perform actions without the user's consent, which ultimately end up in financial loss for the user (and financial gain for the malware developer) and/or data stealing. According to McAfee's most recent report [10], malicious spyware and targeted attacks are becoming more prevalent.

After the seminal research work by Zhou and Jiang [19] in early 2012 (based on data collected in the preceding 2 years), we observed a continuous growth in the numbers of malicious Android apps. However, it is difficult to define what should be considered as one "instance" of a malicious app. For this reason, some security companies like McAfee report that their database contain about 60k distinct malicious Android apps as of Q1 2013 [10], whereas other companies like Trendmicro counted up to 590k distinct "threats" against Android devices. More conservative estimates report numbers between a few hundreds up to less than four thousands [15]. Regardless of the actual value, which may be difficult to define, security companies, researchers and practitioners all agree that there is an ongoing increasing trend of malicious Android apps spotted in the wild, which indicates that the criminals consider this as a viable market opportunity[1].

This of course creates a market for security products on the Android platform. Given the restrictive security model of Android, in which apps are sandboxed, questions arise on how these security applications actually work. For instance, how can a malware detector scan another app's files if Android has no primitives for allowing the former to inspect the filesystem folder of the latter? Some malware detectors work around this limitation by checking the APKs against a list of static signatures at installation time. Unfortunately, this is the best that the Android security model allows them to do (without requiring root privileges, which in turn would increase the attack surface even more).

Malicious developers generate malware variants to evade these basic security checks, even just by changing the package name or repackaging the APKs to alter just a few bytes. Two recent works [13, 18] studied the robustness of current Android malware detectors against evasion attempts such as repackaging, encryption or obfuscation (basically implementing methodologies proposed before the Android era [2, 11]). Unsurprisingly, most products failed at detecting variants of known malware families.

In our vision, considering this trend and the reliance on evasion techniques to circumvent static signatures, instead

SPSM'13, November 8, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2516760.2516768.

[1] `http://cwonline.computerworld.com/t/8652955/807570490/619941/0/`

of focusing on demonstrating that a specific malware detection app has low robustness, the research community would benefit more from scalable approaches to evaluate malware detectors against an arbitrarily large collection of malware samples. Indeed, malware detector testing must be performed on large collections of malware samples, otherwise its significance is low [2, 11].

These crucial aspects are not really taken into account by state-of-the-art malware detector testing approaches [12, 13, 18]. The only attempt at scalability and automation is Google's VirusTotal service[2], which appears similar in spirit to our research vision. However, it uses command-line versions of desktop-based malware-detection applications, modified and extended with the signatures for mobile malware where available (as it is unusual that a desktop-based malware detector looks for malicious code that targets mobile devices). For this and other reasons, Google explicitly discourage the use of VirusTotal for evaluating or comparing malware detectors. Contrarily, we believe that testing the original, unmodified malware detectors is important to guarantee unbiased evaluation and to study their mechanisms of detection on the mobile device itself.

This objective poses a challenge, as the user interface of mobile apps is designed for touchscreens, and thus difficult to automate. Even when feasible, executing an app against a list of malware samples is insufficient without a programmatic method to determine the outcome of each scan. In fact, even the most advanced approach to mobile malware detector testing we could find [12] falls short on this aspect, because it assumes that a human oracle manually verifies the result of each scan, which is clearly unfeasible.

We present AndroTotal, a publicly available tool, malware repository and research framework that aims at mitigating the above challenges, and allow researchers to automatically scan Android apps against a collection of malware detectors. After a careful review of the advantages and disadvantages of existing libraries for user-interface scripting of Android apps, we designed and implemented a generic SDK for automating malware detector testing tasks, which we plan to release publicly. Our SDK offers a high-level interface, and can adapt to virtually any Android malware-detection app with less than one hundred lines of Python code. Our approach is agnostic with respect to the specific Android version and works on both emulated and physical devices.

We deployed and released AndroTotal to the research community in April 2013. As of August 30, 2013, we collected 18,758 distinct submitted samples thanks to a simple yet comprehensive JSON API. We were contacted by several research groups: As a result, AndroTotal is integrated with VirusTotal, CopperDroid, ForeSafe, SandDroid and Anubis.

In summary, we make the following contributions:

- In Section 3 we present an approach to programmatically test Android mobile malware detectors.
- In Section 4 we propose an implementation and low-cost deployment of a publicly available web service that shows how our approach works in practice.
- We provide *bona fide* researchers with free access to our deployment for research purposes. Up to now, we granted access to 1,000 accounts.

[2]`http://virustotal.com`

We envision AndroTotal to become a starting point for creating a data-sharing and research platform, to which mobile malware analyzers can be "plugged in" through open APIs.

2. MALWARE DETECTORS

Android adopts an app-centric security model, where each app runs within a separate user process. Isolation is guaranteed natively by the underlying Linux kernel, which prevents distinct user processes from interfering with each other: As a matter of fact, each user process has a separate memory space. The security of IPC is enforced through a custom permission model, where apps and (sensitive) resources (e.g., phone, networks, SMSs) are allowed to communicate only if the user has granted specific static permissions.

This app-centric security model has both positive and negative consequences on malware development and detection. On the one hand, it prevents, at least in theory, malicious apps from interfering with a benign app (e.g., to read its memory content). On the other hand, as highlighted by Fedler et al. [3], it limits the auditing capabilities of security apps, such as anti-malware products. Indeed, it would need to request all the existing permissions, audit other apps, and intercept all network traffic.

A special set of fine-grained privileges would be necessary for the next generation of malware detectors. Fortunately, given the great research interest around the Android permission model [1, 4, 5, 8, 17], it is realistic that future versions will incorporate special policies for certified auditing security applications. However, given the current state of the art, questions arise on how anti-malware apps currently work, and how effective they can possibly be with such a restrictive security framework.

2.1 Testing Malware Detectors

Testing malware detectors is an extensively investigated problem. In this work we do *not* use the term *testing* to indicate "comparative analysis", which results are often abused for marketing purposes.

The seminal work by Christodorescu and Jha [2] proposed to test the robustness of malware detectors by generating variants via code-transformation. Morales et al. [11] adapted such techniques to mobile malware detectors of the pre-Android era (e.g., Symbian). Recently, Zheng et al. [18] and Rastogi et al. [13] continued along the same line by proposing and testing code-transformation procedures for the Dalvik VM and ARM. The assumption behind this research line is that malware authors will rely on (similar) code-transformation procedures to evade static signatures: Therefore, evaluating malware detectors against mutants is a good estimation of their robustness to future malware. Although [13, 18] have some technical limitations (i.e., corner cases that, according to our experience, produce APKs that do not run properly on some Android versions), we believe that they cover extensively this research direction.

Research Gap. We notice a lack of tools that support researchers in creating experiments to test malware detectors against a given set of samples. VirusTotal is the only initiative in this direction. However, as stated by the maintainers themselves, VirusTotal does not use the original products: It uses command-line versions of the applications (instead of the GUI based ones) along with the same signature database shipped and updated by the vendor. For traditional, PC-based malware and detectors this approach is completely

Feature	Robotium	Monkey Runner	UI Automator	ViewClient	apk-view-tracer	**AndroPilot**
① **User input simulation**	✓	✓	✓	✓	unstable	✓
② **User interface feedback**	✓		✓	slow	slow	slow
③ **Multi-app context**		✓	✓	✓	✓	✓
④ **No anti-malware modification**		✓	✓	✓	✓	✓
⑤ **Version agnostic**	✓	✓		✓	✓	✓
⑥ **Notification framework support**					unstable	✓

Table 1: Comparison of Android user-interface testing libraries with respect to AndroTotal requirements.

reasonable, even if it focuses on signature detection and does not take into account heuristics or behavioral components.

Mobile malware and detectors are different, as also pointed out in part by Pilz [12]. First, there are vendors (e.g., Lookout, Creative Apps, NQ, GFI) that produce malware-detection apps only for mobile platforms (and therefore have no command line clients available). Second, mobile malware detectors may also need to be tested for power efficiency, performance impact or network data usage, which is evidently impossible in an approach similar to VirusTotal's. Finally, when testing mobile malware detectors multiple test scenarios must be taken into account (e.g.,on-demand vs. on-install detection).

Our research solves the scalability limitations of [12] and overcomes the technical obstacles of the existing user-interface testing tools, allowing researchers to fully explore the peculiarities of the mobile settings.

2.2 User-interface Testing

User-interface testing poses a barrier to researchers that need to create scalable procedures to test mobile applications. Pilz [12] does not consider this aspect and assumes that either the malware detector logs the recognized threats in a parsable format, or a human operator manually retrieves such information. These assumptions imply that some malware detectors (i.e., those that support no logging) must be modified or discarded, or that only a limited number of experiments can be carried out because they become labor-intensive.

Automating mobile malware detector testing experiments, in a way which is agnostic to the specific product, is essential to build significant, extensive evaluations.

We examined and tested 6 publicly available libraries and, as summarized in Table 1, none of them are fully suitable for automating modern Android malware detectors with respect to our requirements:

① **User input simulation** to reproduce the gestures needed to control the app.

② **User interface feedback** to inspect the displayed views and activities, to synchronize the testing procedures with the state of the running app and to scrape text (e.g., the threat name) from the interface.

③ **Multi-app context** is needed not only to support complex testing procedures, which may involve more than one application (e.g., malware detector and browser), but also for basic operations (e.g., notification management) across multiple application contexts.

④ **No anti-malware modification** to avoid altering their original code (e.g., instrumentation), which in turn may bias the results of a test.

⑤ **Version agnostic.**

⑥ **Notification framework support** since many malware detectors use it as their only form of output.

3. PROPOSED APPROACH

Our goal is to provide a framework to streamline parallel testing of malware detectors.

3.1 Overview

Our approach consists in installing the original, unmodified anti-malware apps in their native, clean environment. Then, we install the suspicious app under analysis and we register the *system changes* (e.g., user interface, log files, network traffic). This allows us to determine whether or not the malware detector recognizes the *attempt of installing* the malicious app (so-called "on-install detection"). If the anti-malware app permits it, we also perform an explicit scan (including the external SD card) after the infection. This allows us to determine whether or not the anti-malware detects the *presence* of the malicious app on the system storage (so-called "on-demand detection"). We then post-process the system changes that we recorded to extract the label of the detected threats.

Characteristics. Our approach advances the state of the art [12] because it requires no manual work to perform tests. For this, we leverage a custom user-interface automation library that we developed as described in Section 4.2. This library, called AndroPilot, allows to pilot any Android app by emulating a user activity (e.g., press a button, copy the content of a text field) and abstracts the task of running an anti-malware test as a self-contained testing unit. In addition, as described in Section 4.2, we support both emulated and physical devices. This is essential because some malware families refuse to exhibit their maliciousness when they realize that they are being run within an emulated device. Finally, our approach is inherently scalable because each testing procedure is self contained and can run on modest, general-purpose hardware, as described in Section 4.3.

3.2 Definitions and Design

We define a *test* as a function that applies a recipe to a running instance of a given malware detector installed on a chosen environment. A test takes as input a *malware sample* as an APK, the *anti-malware* as the APK of the product version under analysis, and the *environment*, which is the Android platform. The *recipe* is a list of interactions that need to be executed in order to obtain an outcome from the anti-malware, which is the output of the test function.

4. SYSTEM DESCRIPTION

As depicted in Figure 1, AndroTotal is based on open-data formats to ensure easy integration with existing systems. This choice was successful, because we received many contacts from other research groups that wanted to integrate their systems with ours.

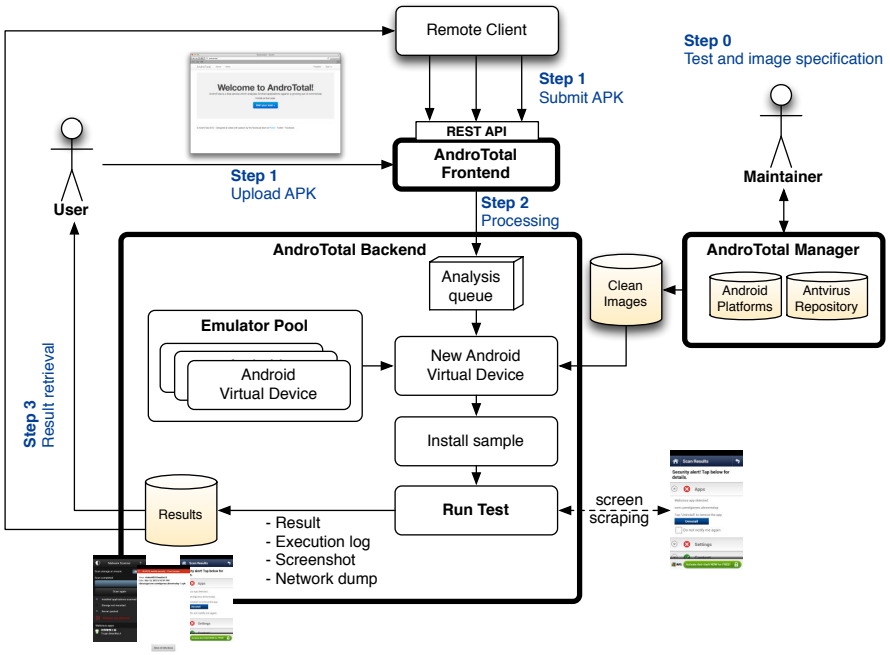

Figure 1: Overview of the logical architecture of AndroTotal.

4.1 Workflow

The *maintainer*'s role is to provision anti-malware apps and write one recipe (as detailed in Section 4.2) for each of them through the AndroPilot framework (**Step 0**). As of now, AndroTotal supports the 10 most popular Android malware detectors[3] (we received explicit permission from the vendors). The maintainer acts only when new malware detectors must be incorporated. Therefore, once a recipe exists for a given anti-malware app, no further manual actions are required to execute any number of tests. Although several pre-packaged Android system images exist as part of the Android SDK, the maintainer can also package customized system images.

The *user* of AndroTotal can be an actual human that uploads (**Step 1**) an APK through an HTML form, or an automated client that sends JSON messages according to a RESTful API. In both cases, an *analysis task* (i.e., execution of a test, as defined in Section 3.2) is enqueued in the AndroTotal's backend (**Step 2**). Whenever a worker, which is a remote or local machine, is available, it pulls a task from the queue and creates a copy of a given clean image (e.g., as requested by the user), mounts a block storage to emulate the SD, runs it within a virtual or physical device and infects either the system image or the block storage with the APK of the malware sample. The infection, which for state-of-the-art Android malware consists in installing an APK, is performed automatically for each test and needs not to be part of the recipe. This step is executed for each anti-malware app, thus creating a set of testing tasks.

[3]The public service exposes 7 out of the 10 supported products as of August 30, 2013.

At this point (**Step 3**), the recipe is executed, and the output is written asynchronously on a database and delivered back to the user (as a JSON or HTML payload).

4.2 Implementation Details

We implemented AndroTotal on top of the Flask Python web framework, leveraging SQLAlchemy for storage abstraction, and Celery for asynchronous task management. In the remainder of this section we describe how we implemented the AndroPilot library and show how a recipe looks like. Also, we detail how the anti-malware upgrade procedures work, and provide some details about our current production deployment.

Recipes: User-interface Automation. The AndroPilot library overcomes the limits of the current user-interface testing tools described in Section 2.2. We selected the library that met most of our requirements (i.e., Apk-view-tracer) and extended it.

More specifically, we leveraged the Android `ViewServer` component to introduce new procedures that 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 a running device or emulator. Additionally, we improved the overall code stability. Finally, AndroPilot includes a Java library that supports the retrieval of screenshots while running a test.

AndroPilot interacts with a running Android instance via `adb` to send commands to the `monkey` server and to the `ViewServer` processes.

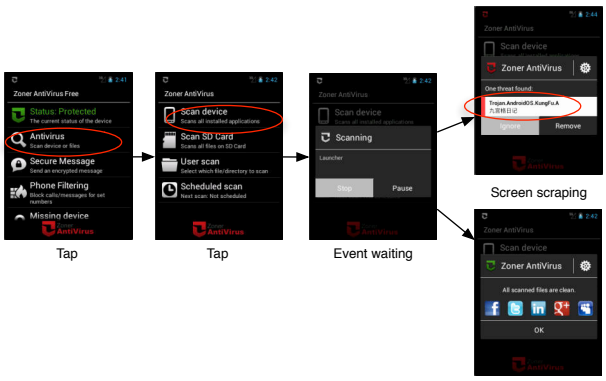

Figure 2: Example user interaction needed to perform an on-demand device scan with Zoner AntiVirus Free 1.7.0.

AndroPilot is written in Python and leverages `monkey` and `ViewServer` as the main tools for interacting with the Android system. Thus, it works on any Android system that supports them, which includes basically any Android OS version on either a virtual or a physical device. The fact that the `ViewServer` requires root privileges is not a limitation because the rooting procedure must be performed only once, when adding a worker for a new (physical) device. Creating a new worker requires no effort.

To illustrate AndroPilot, in the following we summarize the user-interface interactions (depicted in Figure 2) needed to scan a device with the free version of the Zoner AntiVirus and to scrape the threat label (if any is found):

```
from andropilot.pilot import AndroPilot

ap = AndroPilot('device-5554', 'localhost', 4939, 10000)

ap.start_activity(
    "com.zoner.android.antivirus", ".ActMain")
if not ap.wait_for_activity(
    "com.zoner.android.antivirus.ActMain"):
      raise Exception('Activity not found!')

ap.click_view_by_text("Antivirus")
if not ap.wait_for_activity(
  "com.zoner.android.antivirus.ActMalware"):
    raise Exception('Activity not found!')

ap.click_view_by_text("Scan device")
ap.wait_for_activity(
  "com.zoner.android.antivirus_common.ActScanResults")
    raise Exception('Activity not found!')

# define an event to periodically check
event_checker = lambda:(ap.exist_view_by_text("Clean")
  or ap.exist_view_by_text("problem found"))

if ap.wait_for_custom_event(
  event_checker, timeout=45, refresh=True):
  if ap.exist_view_by_text("problem found"):
    threat_name = ap.get_view_by_id("scaninfected_row_virus").
          mText
    result = threat_name
  else:
    result = 'NO_THREAT_FOUND'
else:
  result = 'SCAN_TIMEOUT'
```

Malware Detector Upgrades. After examining the 85 existing anti-malware products we found that they support *app* and *signature* upgrades. In the first case, the app is itself upgraded with a new release. For some products, this type of upgrade implies that also a signature upgrade is bundled with the new release of the app. In the second case, the product has an in-app procedure to upgrade the signatures.

AndroTotal handles both cases with an app-specific recipe written with AndroPilot, which performs the necessary gestures to issue an upgrade command. In our deployment, we run these recipes daily.

A special case are anti-malware apps that use a remote database of signatures: In this case obviously no signature upgrade takes place.

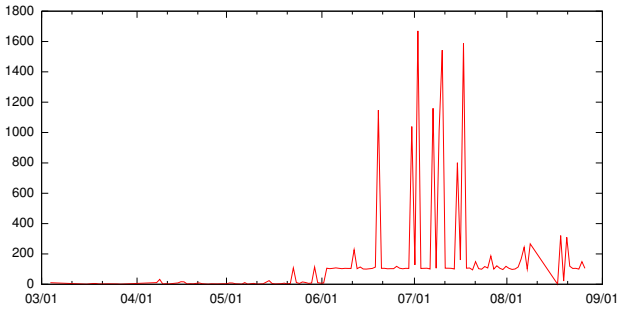

Figure 3: Daily submissions. The spikes beyond the average 100 daily submissions are due to a series of batch submissions for workload testing. As it can be seen, we experienced a downtime August 9–20th, 2013.

Label	#APKs
UDS:DangerousObject.Multi.Generic	3963
HEUR:Trojan-SMS.AndroidOS.Opfake.bo	1252
not a virus Adware.Airpush.origin.7	701
AndroidOS_Opfake.CTD	700
HEUR:Trojan-SMS.AndroidOS.Opfake.a	628
Android.SmsSend.origin.281	620
Android:FakeNotify-A [Trj]	620
HEUR:Trojan-SMS.AndroidOS.FakeInst.a	512
Android.SmsSend.origin.315	485
HEUR:Backdoor.AndroidOS.KungFu.a	466
Android.SmsSend.origin.585	462
Android.SmsSend.origin.629	461
Adware.AndroidOS.Airpush-Gen	432
HEUR:Backdoor.AndroidOS.BaseBrid.a	390
AndroidOS_Opfake.CTC	386

Table 2: Top 15 labels overall.

4.3 Service Deployment

AndroTotal service is currently in beta phase. We deployed it on desktop-class hardware: a dual-core machine with 4GB of RAM handles the frontend and data-storage part, a dual-core and a four-core machines implement 6 parallel workers. In spite of the current hardware limitations, we receive and process a substantial flow of daily and weekly submissions, as Figure 3 shows.

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. As shown in Figure 4, the wide standard deviation does not allow us to firmly state that there are vendors that are considerably slower or faster than others.

Table 2 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

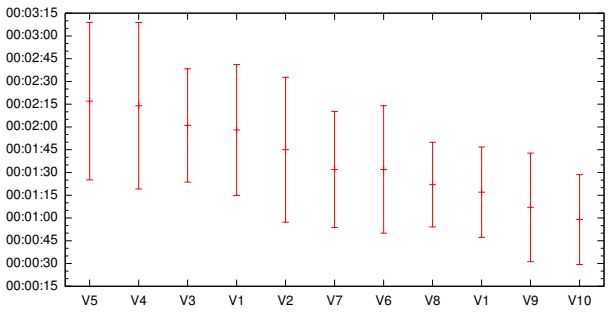

Figure 4: Average time to run a test. Vendors are anonymized, as our goal is not currently that of performing comparative analyses. V1 is repeated because it released two malware detectors.

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.

5. LIMITATIONS AND FUTURE STEPS

Our approach is limited to malware detectors that have a "scrapable" user interface that exposes the details of the detected threat. This assumption holds for all of the 85 commercial and free apps we analyzed.

Some security products include extra features such as browser protection, phone call filtering, remote wiping, or SMS scanning. Such features are not common to every anti-malware product, and fall outside the scope of malware detection, and thus of this paper.

Also, we tested the current implementation on virtual ARM devices over Intel x86 machines: ARM-to-x86 emulation is known to be slow. However, the availability of ARM-as-a-service platforms[4] makes scaling AndroTotal to physical hardware feasible. To this purpose, we are currently investigating a solution to perform hot snapshots-rollbacks of a running machine and its filesystem (similarly to what [9] does for malware analysis on the bare metal). This would allow us to obtain VM-like flexibility and bare-metal-like speed. In addition, running automated tests on physical devices opens up new measurement possibilities, including power-efficiency studies, for instance.

We are also planning to refactor AndroTotal and release it as a framework to automate user-interaction tasks and retrieve results from a running Android instance. Hopefully, this will allow other researchers to contribute to it and, more importantly, to take advantage of AndroPilot for custom experiments.

6. CONCLUSIONS

We implemented and deployed AndroTotal as a web service and infrastructure open to the research community. Our vision is to make it the first component of an independent, collaborative observatory of the mobile malware phenomenon. To this end we are working on creating a new, more powerful deployment and, more importantly, collecting interest from various research groups to federate our respective analysis systems in a larger, cooperative, worldwide infrastructure.

[4]`http://www.boston.co.uk/solutions/viridis/viridis-cloud.aspx`

REFERENCES

[1] S Bugiel, L Davi, A Dmitrienko, and S Heuser. Practical and lightweight domain isolation on android. In *SPSM*, 2011.

[2] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *ISSTA*. ACM, July 2004.

[3] R Fedler, J Schütte, and M Kulicke. On the Effectiveness of Malware Protection on Android. Technical report, Fraunhofer AISEC, Berlin, 2013.

[4] A P Felt, E Ha, S Egelman, A Haney, and E Chin. Android permissions: User attention, comprehension, and behavior. In *SOUPS*, 2012.

[5] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *CCS*. ACM, October 2011.

[6] Gartner. Gartner says asia/pacific led worldwide mobile phone sales to growth in first quarter of 2013, 5 2013.

[7] 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.

[8] J Jeon, K K Micinski, J A Vaughan, and A Fogel. Dr. Android and Mr. Hide: fine-grained permissions in android applications. In *SPSM*, 2012.

[9] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. BareBox: efficient malware analysis on bare-metal. In *ACSAC*. ACM, December 2011.

[10] McAfee Labs. Mcafee threats report: First quarter 2013. Technical report, McAfee, 2013.

[11] Jose Andre Morales, Peter J Clarke, Yi Deng, and B M Golam Kibria. Testing and evaluating virus detectors for handheld devices. *J. of Computer Virology*, 2(2): 135–147, September 2006.

[12] Hendrik Pilz. Building a test environment for Android anti-malware tests. In *VB Conference*, pages 1–7, 2012.

[13] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *ASIA CCS*, New York, NY, USA, 2013. ACM.

[14] Jamie Rosenberg. Google play hits 25 billion downloads. `officialandroid.blogspot.com/2012/09/google-play-hits-25-billion-downloads.html`, 9 2012.

[15] Symantec. Internet security threat report. Technical report, Symantec, 4 2013.

[16] TrendLabs. Android under siege: Popularity comes at a price. Technical report, Trend Micro, Inc., 2013.

[17] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: practical policy enforcement for Android applications. In *USENIX Security Symposium*. USENIX Association, August 2012.

[18] 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 *DIMVA*, April 2012.

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