ThinAV: Truly Lightweight Mobile Cloud-based Anti-malware

Chris Jarabek
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, AB, Canada T2N 1N4
cjjarabe@ucalgary.ca

David Barrera
School of Computer Science
Carleton University
1125 Colonel By Drive
Ottawa, ON, Canada K1S 5B6
dbarrera@ccsl.carleton.ca

John Aycock
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, AB, Canada T2N 1N4
aycock@ucalgary.ca

ABSTRACT

This paper introduces ThinAV, an anti-malware system for Android that uses pre-existing web-based file scanning services for malware detection. The goal in developing ThinAV was to assess the feasibility of providing real-time anti-malware scanning over a wide area network where resource limitation is a factor. As a result, our research provides a necessary counterpoint to many of the big-budget, resource-intensive idealized solutions that have been suggested in the area of cloud-based security. The evaluation of ThinAV shows that it functions well over a wide area network, resulting in a system which is highly practical for providing anti-malware security on smartphones.

Keywords

Android, Malware, Cloud computing, Anti-virus

1. INTRODUCTION

The exponential rise in malware has caused countless research papers to begin with vacuous statements about the exponential rise in malware. Typically these are backed up by token citations to Gartner and Symantec reports, and occasionally a Department of Justice publication for good measure. We will omit this particular ritual.

Massive numbers of malware signatures and related updating issues have caused many anti-malware vendors to move parts of their product into the cloud over the last few years (e.g., [4, 18]). The answer to the question of how big anti-malware can get is therefore limitless. Few people seem to be asking the opposite question, however: how *small* can anti-malware be? This is an especially relevant question for mobile devices. We should note that by "small" we are naturally referring to a small amount of anti-malware software running on end hosts, but also a small amount of supporting infrastructure (ideally none). In other words, a tiny piece of software on the end host plus a massive local cloud infrastructure to maintain does not equal small.

We began answering this question for desktop computers.[1] We wrote a small Python program to intercept file accesses under Linux; we thus had our small desktop footprint, but somehow we had to check the files being accessed for malicious content, *without* creating or maintaining our own cloud infrastructure.

We observed that cloud-based anti-malware exists already, and it is freely-available in the sense that anyone can query it on the Internet. Specifically, we used Kaspersky, VirusChief, and VirusTotal.[2] All of these are similar insofar as a user of the service can upload any type of file, and receive a report about the malware (if any) that might be contained in the file. These WAN-based services acted as our anti-malware scanners; any access to a file that had not already been scanned by our system would be sent for scanning.

Our system was designed in a modular fashion, so it was able to leverage all these existing anti-malware services easily. Scanning requests were sent via the site's API if one existed (VirusTotal), otherwise they were made via HTTP requests and the results scraped from the HTTP responses. (We note that only VirusTotal had terms of service listed, which we abided by in addition to making attempts to minimize our traffic to all these services during testing and evaluation.)

On the desktop, we found that two factors conspired to create an underwhelming user experience. First, the multitude of different files being accessed resulted in poor local cache performance and many files being uploaded. Second, these files would often be accessed several at a time, and in rapid succession, aggravating latency issues. Furthermore, an orthogonal problem is that the files being uploaded could potentially contain sensitive data.

While things look grim on the desktop, the situation for mobile devices – notably Android – is such that our idea fits into the ecosystem better and works well. That is the topic of this paper.

A survey of malware encountered in the wild on Android, iOS and Symbian devices [9] found that all instances of malware for Android devices used application packages as their vector, meaning that users were unknowingly installing the malware on their device. This is not surprising, as Android applications can be installed from an arbitrarily large number of places, unlike the one-stop (and one-stop only!) shopping for iOS applications. There are multiple major An-

---

[1] We only summarize our desktop implementation and experiments here due to space constraints; full details are in [14].
[2] kaspersky.com; viruschief.com; virustotal.com.

droid app markets, even more minor markets, and apps can be downloaded and installed from the Internet or via USB. Furthermore, it is relatively trivial to construct and release a Trojanized version of a legitimate application.

Clearly there is a need for anti-malware protection on Android. In fact, Google has announced that due to the spate of malware on their market, they have developed their own internal anti-malware scanning system called Bouncer, which performs automated scanning of apps submitted to the market [16]. But this is just one application source of many, and this is where our system, ThinAV, fits in.

ThinAV provides lightweight cloud-based anti-malware protection for Android devices, combining a small Android client with the ability to leverage multiple anti-malware services on the Internet. Android effectively forces each application to run as a different user, thus limiting what we need to scan, as one app cannot modify another. We can reasonably constrain ThinAV to look at apps as they are installed, combined with a "killswitch" to manage cases of *post hoc* detection. This addresses file access latency, but also privacy: only apps, not data, need to be scanned, and ThinAV's design proxies scan requests so that individual users cannot be easily profiled by their IP addresses.

Section 3 presents ThinAV's architecture, after the related work in Section 2. Section 4 extensively evaluates ThinAV, and is followed by a discussion of ThinAV's limitations and our conclusions in Sections 5 and 6, respectively.

2. RELATED WORK

Cloud-based malware scanning as posited in [19, 20] was a significant source of inspiration for ThinAV. Their system, CloudAV, has end hosts run a lightweight client (300 LOC in Linux, and 1200 LOC in Windows) which tracks and suspends file access requests until a file has been scanned. This is the only lightweight part; CloudAV relies on a local cloud service consisting of twelve parallel VMs, ten of which run different anti-virus engines, and two run behavioral detection engines. These dedicated scanning servers run in a LAN environment, where the performance hit from network latency and system load is minimal. Even an extension of the CloudAV work to a mobile setting [21] failed to provide any information on how fast their solution operated in the lower-bandwidth / higher-latency mobile realm. ThinAV, by contrast, is truly lightweight. It has a small client that runs on the end host and it relies on already-existing anti-malware services on the Internet.

Many cloud-based anti-malware systems (e.g., [17, 5, 6, 4]) are an exercise in load balancing. Well-provisioned cloud servers perform intensive processing, in concert with clients that handle less demanding processing or operations that require client-side context. (In some cases, just extra processing power and not a cloud is needed: one system validates the contents of a mobile device when it is connected to a desktop or laptop computer via USB [7].) Some systems [13, 22, 2] tilt the balance and make the client a straightforward source of security data for the cloud to process, but again this needs resources on the server side if not the client side. ThinAV sidesteps this by combining a lightweight client with the ability to leverage existing services.

At the other extreme is anti-malware that is based on the end host, per the traditional anti-malware model. This presents a problem on resource-constrained mobile devices, and consequently these mobile systems tend to employ various generic detection heuristics, such as battery consumption [15], memory consumption [12], and heuristic (mostly permission-based) rules [8]. Running solely on a mobile device is neither lightweight nor does it allow use of existing services, though.

Finally, Meteor [1] draws on existing information sources such as developer registries, application databases and remote application killswitches to provide single-market security guarantees in a multi-market environment. While Meteor could potentially provide a framework in which ThinAV could operate, the server-side component is as yet unimplemented.

3. SYSTEM ARCHITECTURE

In this section we present the ThinAV anti-malware system. We begin with an overview, followed by a threat model, and then describe each ThinAV component.

3.1 Overview

ThinAV is an anti-malware system for Android which offloads the chore of scanning to existing third-party malware scanning services. ThinAV was designed to be lightweight, modular, extensible and has been tested with freely available online services such as Kaspersky, VirusChief, VirusTotal and ComDroid [3]. These scanning services all behave similarly in receiving cryptographic hashes of files or the binaries themselves as queries, and returning a scan result.

As shown in Figure 1, ThinAV consists of two main components: (1) an Android client, which submits applications for scanning and (2) a server which submits received files to third-party scanning services and notifies the client in the event of malware detection. The client software consists of modifications to the Android OS package manager as well as a client app for user notification of the scan result. A Killswitch module acts as a periodic post-installation scanner for installed apps. For performance reasons (described in Section 4), the server caches scan results in order to return them faster to clients.

Aside from performance and power consumption reduction, a clear benefit of splitting the client and server is the ability to update scanning modules without having to update the client code. This enables on-demand addition or removal of scanning services.

3.2 Threat Model

Android provides strong application isolation by assigning each application an unprivileged unique UNIX user id (UID). Once apps are installed, the underlying Linux kernel ensures isolation between apps and grants privileges according to the pre-approved permission request (displayed to the user at install-time). Under this security model, the key threat to the user and OS originates primarily from malicious apps that are installed voluntarily by a user, rather than from traditional vectors for malware such as drive-by-downloads and malicious executables [9]. Our threat model assumes:

1. Side-loaded[3] apps cannot be executed without being installed through the OS-provided PackageInstaller,

---

[3]Side-loaded apps are installed through mechanisms other than the official Google Play Store (e.g., third-party markets, file downloads).
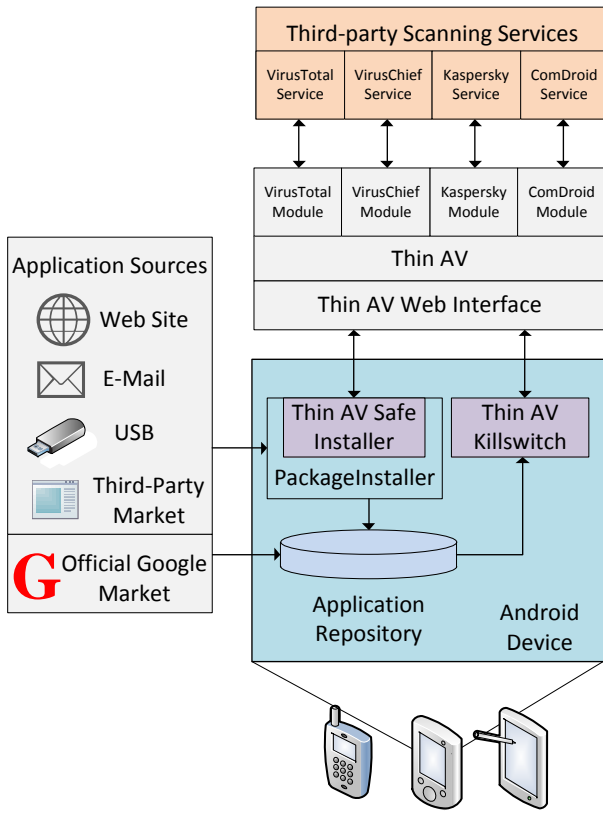
Figure 1: System architecture diagram for ThinAV.

a low-level framework responsible for completing the installation process.

2. Once installed, applications cannot modify their code dynamically. The only way to modify code or functionality is through an application update, which in itself is a new installation (that preserves user data) bound to Assumption 1.

3. The OS and pre-installed system applications are trusted.

3.3 Server

The ThinAV server component is responsible for receiving scan requests and submitting them to the scanning modules on behalf of the client. The ThinAV server is currently implemented in Python using the Flask[4] 0.8 web application micro-framework and runs on Linux. The server is designed in a modular object-oriented fashion, where a parent class provides scanning modules (i.e., subclasses) with all the functionality needed for searching and updating the local cache, as well as uploading binaries via HTTP `POST` requests.

To improve performance, the ThinAV server uses a local cache, which is implemented as a flat file containing previous scan results. While most online malware scanning services cache scan results, the latency in returning results (even those which are cached) was found to be unacceptable in our test conditions (see Section 4). Thus, whenever a scanning module is instantiated, the local cache is first checked.

The cache holds an MD5 hash of each scanned file, the full path to the file, the number of times ThinAV has been asked to analyze the file, the last time such an access has occurred, the infection status of the file, a note for additional scan details, and the module that was used when the file was analyzed.

3.4 Safe Installer

Safe Installer is the ThinAV component on the client responsible for preventing malicious applications from being installed. To build Safe Installer, we modified the Android Package Installer framework,[5] the system code in charge of parsing Android packages to verify integrity, and later completing the installation or update process by creating a new UID (if necessary) and placing files in the appropriate directories. All side-loaded applications must go through the Android Package Installer for installation, making this a ideal choke-point for placing our ThinAV client.

Specifically, we modified the `PackageInstallerActivity` class to make use of `ThinAvService`, a new service class which communicates with the Thin AV server described in the previous section. The service provides a single public function `checkAPK`, accessed via an interface defined using the Android Interface Definition Language. The `checkAPK` function takes the file system path of the Android app package (APK) being installed, reads the file and creates a cryptographic hash of the APK. This hash is then sent to the ThinAV web application, which returns a scan report, if such a report exists. If no scan report exists, the APK is uploaded to ThinAV where it is passed off to one of the third-party scanning services. When a scan result is returned, that result is passed back to `ThinAvService` and `checkAPK` then returns a Boolean value indicating whether or not the installation should be allowed to proceed. The `PackageInstallerActivity` then allows or prevents the installation of the application, displaying the appropriate information dialogs to the user, where necessary.

3.5 Killswitch

Safe Installer can prevent the installation of applications known to be malicious. However, Safe Installer will be unable to prevent the installation of malicious apps in two cases. First, when a malicious application was installed on a device prior to the installation of ThinAV; and second, when an application was installed on a device but was not flagged as malicious at the time of installation. A Killswitch was developed to address these two scenarios. The Killswitch operates independently of any specific application installation mechanism, making it ideal for the multi-market ecosystem available on Android devices.

The Killswitch was developed as a standalone Android application capable of communicating with the ThinAV server, similar to the Safe Installer, but invoked on-demand rather than automatically at install-time.

The Killswitch has three different functions available to the user. (1) It can upload all applications to ThinAV for analysis (if those applications are not in the ThinAV local cache); (2) it can manually check if any non-system applications on the device have been flagged as malicious; and (3) it can regularly check the device for malicious applications

---

using a scheduled event. In the current implementation the killswitch is scheduled to run every 15 minutes.

When the Killswitch is checking for malicious apps, it uses the `PackageManager` class to locate all Android packages installed on the device. For each package, the Killswitch reads the meta-data, creates a hash of each app's byte contents, and a collection of all package hashes is sent to the ThinAV server. If a package has already been hashed by the Killswitch, then the hash is stored in a file which is only accessible to the Killswitch. This hash can then be retrieved much more quickly than recomputing the hash every time the device is fingerprinted. If any of the hashes sent to the ThinAV server are found to be from a malicious app, the user is notified of the infection, and presented a list of applications suspected to be malicious. The user can then choose to initiate the removal of those applications.

3.6 Scanning Modules

ThinAV can be configured to offload scanning to any third-party malware scanning service with a public API or web interface. The system currently has modules for four scanning services that are freely available online: Kaspersky, VirusChief, VirusTotal, and ComDroid. These services all behave similarly insofar as a user can upload any type of file (executable, data, etc.) through the website of the service and receive a report as to any malware that might be contained in that file. Unfortunately, these scanning services are based on proprietary anti-malware engines, and as such, the exact details of the engines underpinning these services are very closely held trade secrets. Therefore, the exact capabilities and limitations of these services with respect to threat detection are not publicly known.

The currently implemented modules and their descriptions:

- Kaspersky – Offers a free service that uses a proprietary anti-malware engine for scanning files that are 1MB or smaller in size.

- VirusChief – A multi-engine anti-malware scanning service with a 10MB file size limit.

- VirusTotal – Scans uploaded files up to 20MB in size with 42 different scanning engines. VirusTotal has a public API to interact with its services.

- ComDroid - While not an anti-malware engine, ComDroid can identify potential vulnerabilities in Android apps by performing static code analysis. The ComDroid module identifies scanned applications as being "at risk" as opposed to being "infected".

The ThinAV server is currently configured to select scanning services based on the average amount of time each module takes to scan a file. We found the fastest service to be Kaspersky, followed by VirusChief, ComDroid, and Virus-Total. If any scanning module returns an error from an attempted online scan, then the next module in the priority sequence is selected. If all four scanning modules fail, a general error code is returned to the device that originated the request. Performance measurements for the scanning modules are discussed in the following section.

4. THINAV EVALUATION

This section presents the results of our evaluation of ThinAV. All development and testing was done on the Android

Number of Apps	1022
Mean App Size	2.65 MB
Median App Size	1.78 MB
Minimum App Size	0.02 MB
Maximum App Size	37.06 MB
Proportion of Apps <1 MB	34.64 %
Proportion of Apps <10 MB	97.16 %
Proportion of Apps <20 MB	99.51 %

Table 1: General file size characteristics of the Android test data set.

emulator provided in the SDK. The emulator enabled rapid development on different versions of the Android operating system, and allowed for changes to be made to the Android source code.

Working on the emulator presents evaluation issues with respect to network performance. Because ThinAV is heavily reliant on the network, link speed has a direct impact on performance. On a mobile device like a cell phone, the speed of the cellular connection can vary based on the location of the user, radio interference, the load on the cellular network, as well as other factors. Due to the challenges involved in cellular network measurements, we use the results of Gass et al. [10].

4.1 Data Set

The evaluation process was performed with a collection of apps downloaded from the official Google Play Store (known as the Android Market at the time of data collection) using a custom crawler. We downloaded the top 50 free apps (as ranked by user ratings) in each application category on January 3, 2012. The majority of package downloads were successful, with 28 downloads causing repeated failures. This resulted in 1,022 apps spread across 21 application categories, with each category having between 46 and 50 packages. Table 1 summarizes the key file size statistics of the data set.

4.2 Malware Detection

We first uploaded the entire data set to the VirusTotal scanning service to to confirm that VirusTotal, and therefore other scanning services are capable of correctly receiving and scanning Android applications. This initial scan also allowed us to obtain a baseline of detection. That is, to see how many apps in our initial data set contain malware.

VirusTotal flagged several possible instances of malware in the data set downloaded from the Google Play Store. Of the 1,022 apps uploaded, 1,019 were scanned (three apps were skipped due to size restrictions) and 27 were flagged as malware by at least one scanning engine. One package was flagged as malware by four different engines, nine packages were flagged by two engines, and the remaining seventeen packages were flagged as malware by a single engine. Table 2 provides details on some of the commonly flagged samples. The most commonly identified sample was from the `Adware.Airpush` family. However, the majority of these samples were identified by a single scanning engine (DrWeb), which raises the possibility of this being a false positive. The next most common sample was `Plankton`, which was identified by a variety of scanning engines. The remaining

Sample Name	Malware Type	Count	Detection Engine(s)
Adware.Airpush(2, 3)	Adware	15	DrWeb, Kaspersky
Plankton (A, D, G)	Trojan	6	Kaspersky, Comodo, NOD32, Trend Micro
SmsSend (151, 261)	Dialer	2	DrWeb
Rootcager	Trojan	2	Symantec

Table 2: Most frequent samples of malware detected in Google Market data set. Detection engine refers to which VirusTotal scanning engines detected the sample.

malware samples had far fewer occurrences in the data set.

While the test data set only suggested that 6 of the AV engines used by VirusTotal are capable of detecting Android malware, we later confirmed that as many as 26 (more than half of the VirusTotal scanning engines) are capable of detecting some form of Android specific malware.

4.3 AV Scanning Module Performance

We developed a testing program which uploaded files of different sizes to each of the scanning services at specific time intervals. This program was designed to measure the response time of each service. The program submitted 12 files (of sizes 0 KB, 1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB and 1023 KB) for scanning in random order over an 8 day window. The files were created by a script which produces files of a specified size filled with pseudo-random bits with the expectation that files generated in such a way would have an extremely low probability of being flagged as malware by one of the scanning services, or exist in the service cache. Test files were uploaded in a pseudo-random order every time the test program was run to overcome any penalty that might be incurred against the first file being uploaded due to DNS lookups.

Results.

For each of the three scanning services, several hundred response time measurements were recorded. A cursory review of the data showed a handful of extreme outliers for each service. Any measurement beyond two standard deviations of the mean was classified as an outlier. This threshold was chosen because it eliminated the most extreme results, while retaining the vast majority of the data.

A comparison of the measurements from the three services shows a clear difference of nearly an order of magnitude between the performance of VirusTotal and the other two scanning services. The average response times from Kaspersky and VirusChief range from 1.54 – 14.49 seconds and 6.82 – 28.70 seconds respectively, while the response times from VirusTotal range from 1.21 – 229.28 seconds (though the latter range becomes 148.92 – 229.28 seconds, when only non-zero file sizes are considered). The upload portion of VirusTotal shows response times similar to Kaspersky with response times ranging from 1.94 – 11.74 seconds.

With the outliers removed, the response time data was plotted (see Figures 2, 3, and 4) in an attempt to determine the correlation between file size and service response time

for the three scanning services. Each of the figures shows the upload file size plotted versus the response time for each of the three scanning services, and Figure 5 graphs the upload and response speed of VirusTotal. Kaspersky and VirusChief both show a similar positive correlation between file size and response time, with the VirusChief data being positively shifted on the y-axis (and therefore slower) by roughly fifteen seconds. VirusTotal, on the other hand, shows little if any relationship between file size and response time. The trend of the VirusTotal response time data is slightly negative and has a much larger y-intercept than either of the other two scanning services. Conversely, the upload portion of the VirusTotal scan shows a trend very similar to Kaspersky. With this data, we produced a set of linear equations which approximate the performance of the scanning services as a function of the number of bytes in the file (see Table 3).
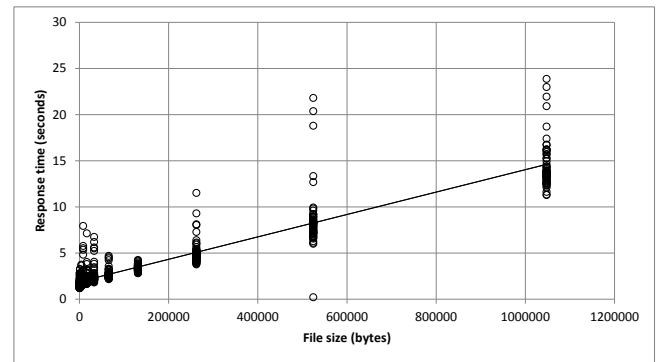
Figure 2: Scan response time versus file upload size for the Kaspersky virus scanner service.
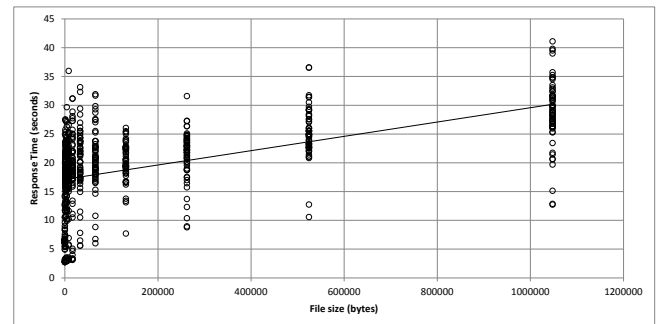
Figure 3: Scan response time versus file upload size for the VirusChief virus scanner service.

Discussion.

It is not surprising that Kaspersky, which only scans with a single anti-virus engine, returns the fastest results, and VirusChief, which scans with six anti-virus engines is roughly fifteen seconds slower than Kaspersky when scanning a similarly sized file. We attribute the erratic VirusTotal performance to the company's prioritization of scanning requests. VirusTotal assigns the lowest priority to requests that are sent via their formal API, and the response appears to not be dependent on the size of the uploaded file, but rather on how busy the VirusTotal scanning service is at any given
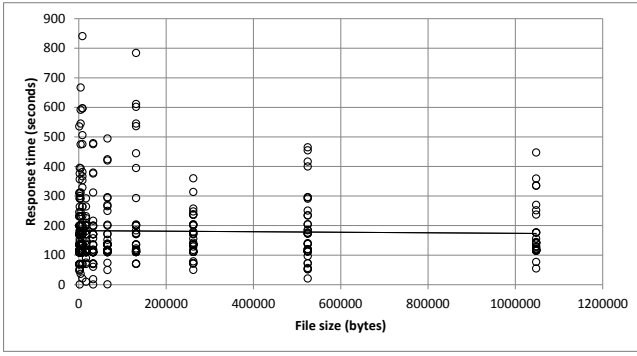
Figure 4: Scan response time versus file upload size for the VirusTotal virus scanner service.
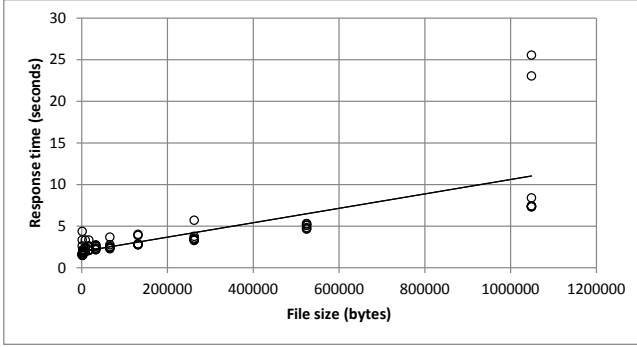

Figure 5: Upload response time versus file upload size for the VirusTotal virus scanner service when uploading a file and not polling for a scan result.

moment. This distinction is made much clearer when comparing the time required to scan a file with VirusTotal with the time required to merely upload a file to VirusTotal.

4.4 ComDroid Performance

To evaluate ComDroid, we uploaded each of the 1,022 apps in our data set to the ComDroid scanning service over a two day period. For each upload, we recorded the time required for a scan report to be returned.

Results.

Of the 1,022 packages uploaded to ComDroid, 993 were scanned, with the remainder being rejected by the server due to a 10 MB size limitation. Of the 993 packages scanned by ComDroid, 8 returned a scan error, resulting in 985 valid scan results. The mean response time was 40.67 seconds (σ

Kaspersky	$f(x) = 10^{-5} \times x + 1.891$
VirusChief	$f(x) = 10^{-5} \times x + 17.133$
VirusTotal	$f(x) = -9 \times 10^{-6} \times x + 182.98$
VirusTotal (Uploads)	$f(x) = 9^{-6} \times x + 1.947$

Table 3: Linear equations for each of the three scanning services derived from Figures 2, 3, 4, and 5. Equations calculate the response time for each scanning service for a file x bytes in size.
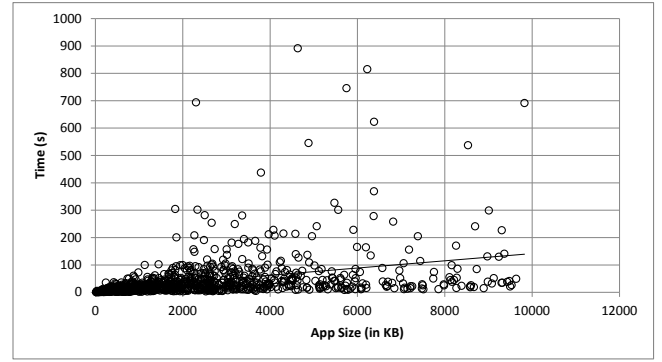

Figure 6: Response time of the ComDroid service as a function of package size.

ComDroid	$f(x) = 0.0132 \times x + 9.6893$

Table 4: Linear equation for the ComDroid scanning service.

= 77.60 seconds), and the median response time was 18.63 seconds. Figure 6 shows the response time plotted as a function of the package size, and the exact function is specified in Table 4. There is a clear positive linear relationship between package size and scan time, although numerous outliers are present.

Discussion.

The performance of the ComDroid service is somewhat similar to both the Kaspersky and VirusChief services (Section 4.3). However, the linear trend is much less prominent. The most likely explanation for this lies in the nature of the analysis performed by ComDroid. ComDroid is a static code analysis tool, and as such, it is safe to assume that the time required to analyze an Android app is more directly influenced by the amount of code in the package, than the total size of the package. Given that many apps contain numerous resource files (images, sounds, video, etc.) which are not scanned by ComDroid, it is easy to imagine how a package might have a large size, but a relatively small amount of code. It is quite likely that the observed linear trend is much more a result of the upload time and not the code-to-resource-file ratio of the package.

ComDroid detects a wide variety of "programming errors". In our experiment, 971 of 985 apps (98.6%) were flagged by ComDroid as containing some type of code that is vulnerable to exposed communication. This suggests that, at least in its current form, simply flagging an application as being "at risk" if there is any instance of exposed communication would effectively cripple the ability of users to install apps on their device. Chin et al. [3] (the ComDroid authors) suggest that in their data, after manual inspection of warnings, only about 10–15% were genuine vulnerabilities.

We believe that there is a place for ComDroid (and other tools like it) in the ThinAV architecture. However, the behavior of this ThinAV module would likely have to be adjusted over time to prevent excessive false positives. This could be done by creating thresholds which would flag a package as vulnerable if it had significantly more exposed surfaces than average for a given type of warning.

Network Con-figuration	Upload Speed (KBps)	Download Speed (KBps)
Typical 3G	16.25	84.13
Ideal 3G	1792.00	1792.00
Typical WiFi	190.38	155.38
Ideal WiFi	76800.00	76800.00

Table 5: Network speeds used for evaluating the mobile implementation of ThinAV.

Network Configuration	Small File	Medium File	Large File
Ideal 3G	0.034 s	0.232 s	0.293 s
Typical 3G	0.041 s	0.239 s	0.300 s
Ideal WiFi	0.034 s	0.231 s	0.293 s
Typical WiFi	0.035 s	0.233 s	0.294 s

Table 6: Time required to check an package in ThinAV, for three different file sizes, and four different network configurations, assuming the scan result is already cached by the ThinAV server

4.5 Safe Installer Performance

The performance of the Safe Installer is based on three factors: the size of the package being scanned, the speed of the network to which the device is connected, and whether or not the package being installed has already been scanned by ThinAV. To evaluate the Safe Installer, we use three different file sizes: 0.76 MB (small), 1.78 MB (medium), and 3.56 MB (large), corresponding to the median size of apps in the category with the smallest median size (medical apps), the median size for the entire data set, and the median size of apps in the category with the largest median size (educational apps). We also use the "ideal" and "typical" 3G and WiFi speeds from prior work [10, 11] (see Table 5).

Results.

The best case scenario for the performance of the safe installer is when the package being installed has already been scanned by ThinAV. In this case, the cost for performing an install time check is equal to the time required to hash the installing application, send the hash to ThinAV, look up the scan result, and return the scan result.

We measured the time required to hash a small, medium, and large application on the Android emulator, and took the average of five runs for each size. The emulator was able to calculate hashes of small files in 0.033 seconds, medium files in 0.231 seconds, and large files in 0.293 seconds. We recorded the amount of data uploaded and downloaded during transmission of the hash to the ThinAV server, as well as the reception of the result. This was approximately 200 bytes (100 up, 100 down), although this amount varied slightly with the file being scanned. Finally, the cost of the ThinAV server performing a cache lookup was 0.0002 seconds. Table 6 summarizes the results for this best case scenario. In general, even the largest file over the slowest network only takes 0.3 seconds to check with ThinAV.

The worst case scenario is when the application being installed has not been scanned by ThinAV, and therefore the whole package must be uploaded to ThinAV, which must then upload the package to one or more of the third-party scanning services. Using the formulæ in Tables 3 and 4,

Network Configuration	Small File	Medium File	Large File
Ideal 3G	36.56 s	98.13 s	170.29 s
Typical 3G	84.66 s	210.00 s	394.14 s
Ideal WiFi	36.13 s	97.14 s	168.31 s
Typical WiFi	40.23 s	106.68 s	187.39 s

Table 7: Time required to check an package in ThinAV, for three different file sizes, and four different network configurations, assuming the scan result is not cached by ThinAV.

and the file sizes and network speeds above, it is possible to compute the time required to upload and scan these files at install time. We note that when calculating the time required to scan a package, we add both the time to scan the package with an anti-virus module as well as the time for scanning with ComDroid.

Table 7 summarizes the results for this worst case scenario. In general, the time required to upload and scan an Android package ranges between 36 seconds and 394 seconds, depending on the size of the file and the speed of the network.

Discussion.

The best case scenario, where ThinAV already has a cached scan result, is extremely fast. At 0.3 seconds, this check would be unnoticeable to a user. On the other hand, if the file must be uploaded and scanned, this process could take as long as seven minutes. This could be seen as a serious inconvenience to the user, but considering that this check would only take place when a user is installing an app that has never been seen by the ThinAV server, large-scale deployment should make this an infrequent occurrence. Additionally, given that ThinAV could be primed with packages from a variety of sources, including regular downloads of applications from various application markets, upload of applications by developers, and the upload of applications by other users running ThinAV, the chance that a user would have to upload a package for scanning at install time could be made very rare. Of course, only the evaluation of a wide-scale deployment can provide confirmation of our intuition.

4.6 Killswitch Performance

During normal operation, we expect the most frequently used functionality of ThinAV to be the killswitch service which is periodically activated and checks for revoked apps. To evaluate the performance of the killswitch, we examine several factors: (1) the cost of hashing apps to generate a system fingerprint; (2) the network cost associated with uploading the fingerprint; (3) the cost of looking up the hashes on the ThinAV server; and (4) the network cost associated with returning those hashes to the client. We also consider a manual upload feature, in which all packages currently installed are submitted for scanning. This is required to scan applications that were installed prior to enabling ThinAV.

In general, the time required for the killswitch to perform a check for revoked apps without uploading any full packages to the ThinAV server will be:

$$h_1 = \text{time to hash all packages}$$
$$s_u = \text{hash upload size}$$
$$sp_u = \text{link upload speed}$$
$$c_1 = \text{cache lookup time} \quad (1)$$
$$s_d = \text{response download size}$$
$$sp_d = \text{link download speed}$$
$$t_1 = h_1 + \frac{s_u}{sp_u} + c_1 + \frac{s_d}{sp_d}$$

Because the cost of performing a manual upload of missing packages is dominated by upload and scanning costs (similar to the safe installer above), we include only these costs in the calculation. The time required for the killswitch to manually upload missing packages is:

$$s_u = \text{package upload size}$$
$$sp_u = \text{link upload speed}$$
$$t_s = \text{time to scan all applications} \quad (2)$$
$$t_2 = \frac{s_u}{sp_u} + t_s$$

To test the performance of the hashing function, we installed the top five apps from each of the 21 Google Play app categories (on top of the 5 default non-system apps) on the Android emulator. After installation, we generated a complete system fingerprint (i.e., a hash for each app installed on the user partition) ten times and recorded the average time. This represents the worst case scenario in which none of the apps on the device have been hashed before, and all hashes must be computed. Next, we generate another ten fingerprints and record the average time. The cache created in the previous test was left intact, however. This represents the best case scenario in which all of the apps on the phone have already been hashed and the phone fingerprint is stored locally.

Under normal use it is likely to expect that the typical scenario would in fact be the best case scenario, or very close to it. After the first fingerprint has been generated, the only time an app will have to be hashed is when it has not been seen by the killswitch, meaning it has just been installed. Unless a user installs numerous apps between the scheduled runs of the killswitch, it is likely the number of apps that need to be hashed would be near zero.

Combining the hashing performance with the file size data for the data set, the scanner performance functions in Tables 3 and 4, and the experimental network performance measurements from [10], we calculate the cost of performing manual uploads, as well as the cost of fingerprinting based on Equations 1 and 2.

Results.

Figure 7 shows the best (cached) and worst (uncached) case scenarios for the fingerprint generation time as a function of both the number of packages on the device and the total size of those packages.

It is clear that time to generate a system fingerprint grows linearly with both the number and size of packages on the device. In the worst case, with 110 apps on the device, it only takes 29.95 seconds to generate a system fingerprint. The best case scenario is better, with a fingerprint being

(a)

(b)
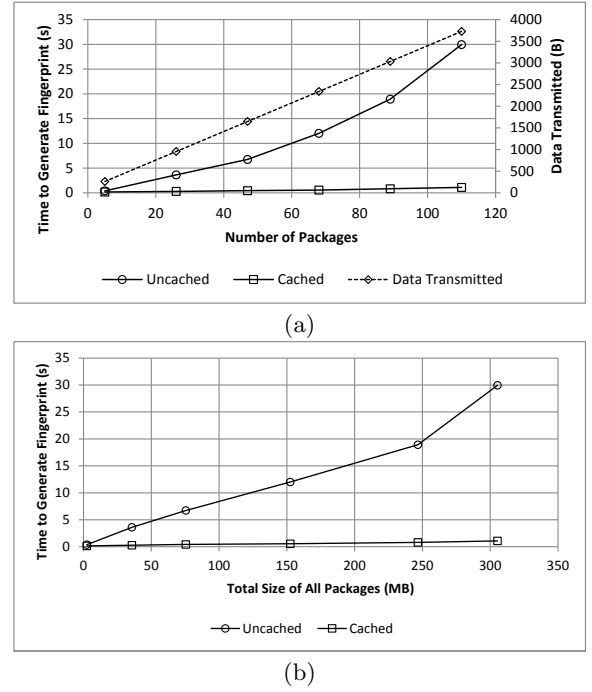
Figure 7: Time required to generate a complete system fingerprint as a function of the number of packages installed on the device (a) and the total size of those packages (b). Both figures show the average time when all of the package hashes have been stored (cached) and when none of the package hashes are stored (uncached). Figure (a) also includes the number of bytes sent and received when communicating the fingerprint to the ThinAV server.

generated in 1.09 seconds for the same 110 apps when the fingerprint has been cached.

Data usage grows linearly with the number of packages on the device. The data consumption ranges from 3.64 KB for 110 apps, down to 261 bytes for 5 apps. The majority of this transmission is in the form of the uploaded fingerprint, as the response from ThinAV only downloads 70 bytes from the server when the fingerprint contains no hashes corresponding to malicious apps.

The current implementation of the ThinAV client is scheduled to run the killswitch service every 15 minutes. Table 8 shows how much data would be consumed by ThinAV (under the current configuration) over different lengths of time

Interval	Data Consumption (5 Apps)	Data Consumption (110 Apps)
1 Day	24.47 KB	349.41 KB
1 Week	171.28 KB	2.39 MB
1 Month	5.19 MB	74.04 MB

Table 8: Data consumption of ThinAV killswitch over different time periods, for 5 and 110 apps installed on the device, assuming the killswitch is scheduled to run every 15 minutes.

Scenario	Time (seconds)
110 apps / ideal 3G / no hashes cached	26.206
110 apps / typical 3G / no hashes cached	26.430
110 apps / ideal WiFi / no hashes cached	26.204
110 apps / typical WiFi / no hashes cached	26.223
110 apps / ideal 3G / all hashes cached	3.424
110 apps / typical 3G / all hashes cached	3.478
110 apps / ideal WiFi / all hashes cached	3.423
110 apps / typical WiFi / all hashes cached	3.428
26 apps / ideal 3G / no hashes cached	1.034
26 apps / typical 3G / no hashes cached	1.258
26 apps / ideal WiFi / no hashes cached	1.032
26 apps / typical WiFi / no hashes cached	1.051
26 apps / ideal 3G / all hashes cached	0.285
26 apps / typical 3G / all hashes cached	0.339
26 apps / ideal WiFi / all hashes cached	0.285
26 apps / typical WiFi / all hashes cached	0.290

Table 9: Time required to complete the fingerprinting operation for different numbers of applications, network performance, and caching scenarios.

and with two sets of installed apps (5 and 110).

Using the same network measurements from Section 4.5, the measured fingerprint generation times, and data transmission totals, it is possible to compute a variety of potential running times for the entire fingerprinting operation of ThinAV killswitch using Equation 1. These values are summarized in Table 9.

	Total Data Uploaded (MB)		
Scenario	10 Apps	25 Apps	50 Apps
Small Apps	7.643	19.108	38.216
Medium Apps	17.775	44.438	88.875
Large Apps	35.570	88.925	177.850

Table 10: Total upload sizes used for calculations of manual scanning performance.

For calculating the cost of manually uploading missing packages, we use the package size averages from Section 4.5 and use three sets of apps (10, 25, and 50 apps). Table 10 summarizes the total amount of data that would be uploaded for different numbers of apps of different sizes. The upload times for the different numbers and sizes of apps are summarized in Table 11. Using the size and quantity of each app, the scanning time could then be computed using the equations in Tables 3 and 4. These results are summarized in Table 12. Finally, referring to Equation 2, it is possible to compute the time required to upload and scan missing apps under different scenarios.

The best case scenario is when ten small apps are uploaded and scanned over an ideal WiFi connection. This takes 289.2 seconds, or just under five minutes. The worst case scenario is where 50 large apps are uploaded and scanned over a typical 3G connection. This operation takes 17351.2 seconds, or nearly five hours. However, if the same operation is performed over a typical WiFi connection, the time required to complete this one-time operation drops by more than half, to 1.95 hours.

		Upload Time (Seconds)		
Scenario		10 Apps	25 Apps	50 Apps
Ideal 3G	Small Apps	4.367	10.919	21.837
	Medium Apps	10.157	25.393	50.786
	Large Apps	20.326	50.814	101.629
Typical 3G	Small Apps	485.360	1213.400	2426.801
	Medium Apps	1128.781	2821.953	5643.907
	Large Apps	2258.833	5647.082	11294.164
Ideal WiFi	Small Apps	0.102	0.255	0.510
	Medium Apps	0.237	0.593	1.185
	Large Apps	0.474	1.186	2.371
Typical WiFi	Small Apps	41.111	102.777	205.553
	Medium Apps	95.609	239.023	478.046
	Large Apps	191.326	478.315	956.630

Table 11: Upload times for the values in Table 10, for four different network configurations.

	Scanning Time (Seconds)		
Scenario	10 Apps	25 Apps	50 Apps
Small Apps / Kaspersky	161.021	402.552	805.104
Medium Apps / VirusChief	634.032	1585.080	3170.159
Large Apps / VirusChief	1104.893	2762.232	5524.465
Small Apps / ComDroid	107.224	252.563	494.796
Medium Apps / ComDroid	120.919	266.259	508.491
Large Apps / ComDroid	144.972	290.312	532.544

Table 12: Scan times for different numbers of apps with small, medium and large sizes, using conventional scanning engines (Kaspersky and VirusChief) as well as the Android-specific scanner, ComDroid.

Discussion.
During long-term use of ThinAV, fingerprinting installed apps is the only operation that would likely take place frequently. In the best case, the killswitch requires about 1 second of computation followed by less than 4 KB of data transmission for a set of 110 installed apps. This operation would be unnoticeable to a user, especially on a physical Android device, which is typically more powerful than the Android emulator.

In terms of data consumption, the 74 MB per month for uploading the fingerprint of 110 apps is non-trivial, particularly for users with pay-as-you-go type data plans. We note, however, that data consumption can be lowered by reducing the frequency with which the killswitch is run. Compressing the data for fingerprint submission and response retrieval is possible and would likely also reduce data consumption.

5. LIMITATIONS

We have shown that despite being in an early prototype form, ThinAV could be realistically deployed on actual devices as a free and lightweight anti-malware system. We now discuss some of the limitations of the prototype and the system overall.

OS modification. Because the Package Installer is part of the core Android OS, ThinAV cannot be installed on any Android device as an application. Instead, the underlying OS must be replaced with a ThinAV-enabled version. Alternatively, Google and other phone manufacturers could incorporate ThinAV directly into their own builds.

Test environment. All tests in our experiment were performed on the Android emulator. While we believe the em-

ulator provides accurate technical feasibility metrics, it also provides a lower bound on speed measurements. Physical devices are generally more powerful, but also have battery consumption concerns. Future work will evaluate battery consumption by ThinAV.

Third-party scanning services. ThinAV relies on the continual existence of third-party scanning services in a production capacity. The terms of service of these services may change, and services may also cease to exist. The decision on whether or not to support HTTPS connections is also out of ThinAV's control, as are denial-of-service attacks (which are possible for any cloud-based anti-malware). Fortunately, the modular design of ThinAV should help transparently replace scanning modules without updating clients.

6. CONCLUSIONS

Keeping malware in check for Android is a difficult problem; Android has a chaotic multi-market app environment and the ability for users to side-load apps of unknown provenance. We address this problem not by imposing a massive anti-malware regime, but by just the opposite. Our ThinAV system combines a lightweight footprint on an Android device, consisting of a safe installer and killswitch, with the ability to leverage multiple free, already-existing anti-malware services on the Internet. As only apps are scanned and requests are proxied through a ThinAV server, no personal or IP address data is leaked to outside services. Our experiments with performance and data consumption have shown that small is practical, especially if ThinAV is fully integrated into the Android app ecosystem and is already primed with scan results for popular apps.

7. REFERENCES

[1] D. Barrera, W. Enck, and P. C. van Oorschot. Meteor: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems. In *IEEE Mobile Security Technologies*, 2012.

[2] J. Cheng, S. H. Wong, H. Yang, and S. Lu. Smartsiren: virus detection and alert for smartphones. In *5th International Conference on Mobile Systems, Applications and Services*, pages 258–271, 2007.

[3] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *9th International Conference on Mobile Systems, Applications, and Services*, pages 239–252, 2011.

[4] M. Chiriac. Tales from cloud nine. In *19th Virus Bulletin International Conference*, pages 83–88, 2009.

[5] B.-G. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *12th Workshop on Hot Topics in Operating Systems*, 2009.

[6] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *8th International Conference on Mobile Systems, Applications, and Services*, pages 49–62, 2010.

[7] B. Dixon and S. Mishra. On rootkit and malware detection in smartphones. In *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 162–163, 2010.

[8] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *16th ACM Conference on Computer and Communications Security*, pages 235–245, 2009.

[9] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14, 2011.

[10] R. Gass and C. Diot. An experimental performance comparison of 3G and Wi-Fi. In *Passive and Active Measurement*, volume 6032 of *LNCS*, pages 71–80, 2010.

[11] IEEE Computer Society. Wireless LAN medium access control (MAC) and physical layer specifications enhancements for higher throughput, Oct. 2009. IEEE Std 802.11n-2009.

[12] M. Jakobsson and K.-A. Johansson. Retroactive detection of malware with applications to mobile platforms. In *USENIX HotSec*, 2010.

[13] M. Jakobsson and A. Juels. Server-side detection of malware infection. In *2009 New Security Paradigms Workshop*, pages 11–22, 2009.

[14] C. Jarabek. Towards cloud-based anti-malware protection for desktop and mobile platforms. Master's thesis, University of Calgary, 2012.

[15] L. Liu, G. Yan, X. Zhang, and S. Chen. VirusMeter: Preventing your cellphone from spies. In *Recent Advances in Intrusion Detection*, volume 5758 of *LNCS*, pages 244–264, 2009.

[16] H. Lockheimer. Android and security. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`, Feb. 2012.

[17] L. Martignoni, R. Paleari, and D. Bruschi. A framework for behavior-based malware analysis in the cloud. In *Information Systems Security*, volume 5905 of *LNCS*, pages 178–192, 2009.

[18] C. Nachenberg, Z. Ramzan, and V. Seshadri. Reputation: A new chapter in malware protection. In *19th Virus Bulletin International Conference*, pages 185–191, 2009.

[19] J. Oberheide, E. Cooke, and F. Jahanian. Rethinking antivirus: executable analysis in the network cloud. In *USENIX HotSec*, 2007.

[20] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *17th USENIX Security Symposium*, pages 91–106, 2008.

[21] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian. Virtualized in-cloud security services for mobile devices. In *1st Workshop on Virtualization in Mobile Computing*, pages 31–35, 2008.

[22] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: versatile protection for smartphones. In *26th Annual Computer Security Applications Conference*, pages 347–356, 2010.

[23] D. Rowinski. More than 50% of Android devices still running Froyo. ReadWrite Mobile, 6 September 2011. `http://www.readwriteweb.com/mobile/2011/09/more-than-50-of-android-device.php`.