Sweetening Android Lemon Markets: Measuring and Curbing Malware in Application Marketplaces

Timothy Vidas and Nicolas Christin

November 16, 2011

(revised June 8, 2012)

CMU-CyLab-11-012

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
08 JUN 2012		**00-00-2012 to 00-00-2012**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Sweetening Android Lemon Markets: Measuring and Curbing Malware in Application Marketplaces	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,CyLab,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Application marketplaces are the main software distribution mechanism for modern mobile devices but are also emerging as a viable alternative to brick-and-mortar stores for personal computers. While most application marketplaces require applications to be cryptographically signed by their developers, in Android marketplaces, self-signed certificates are common, thereby offering very limited authentication properties. As a result, there have been reports of malware being distributed through application ?repackaging.? We provide a quantitative assessment of this phenomenon by collecting 41,057 applications from 194 alternative Android application markets in October 2011, in addition to a sample of 35,423 applications from the official Google Android Market. We observe that certain alternative markets almost exclusively distribute repackaged applications containing malware. To remedy this situation we propose a simple verification protocol, and discuss a proof-of-concept implementation, AppIntegrity. AppIntegrity strengthens the authentication properties offered in application marketplaces, thereby making it more difficult for miscreants to repackage apps, while presenting very little computational or communication overhead, and being deployable without requiring significant changes to the Android platform.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**21**	

Sweetening Android Lemon Markets: Measuring and Curbing Malware in Application Marketplaces

Timothy Vidas[a] *Nicolas Christin*[b]
[a] Carnegie Mellon ECE/CyLab
tvidas@cmu.edu

[b] Carnegie Mellon INI/CyLab
nicolasc@cmu.edu

Original revision: November 16, 2011 – This version: June 8, 2012.

Abstract

Application marketplaces are the main software distribution mechanism for modern mobile devices but are also emerging as a viable alternative to brick-and-mortar stores for personal computers. While most application marketplaces require applications to be cryptographically signed by their developers, in Android marketplaces, self-signed certificates are common, thereby offering very limited authentication properties. As a result, there have been reports of malware being distributed through application "repackaging." We provide a quantitative assessment of this phenomenon by collecting 41,057 applications from 194 alternative Android application markets in October 2011, in addition to a sample of 35,423 applications from the official Google Android Market. We observe that certain alternative markets almost exclusively distribute repackaged applications containing malware. To remedy this situation we propose a simple verification protocol, and discuss a proof-of-concept implementation, AppIntegrity. AppIntegrity strengthens the authentication properties offered in application marketplaces, thereby making it more difficult for miscreants to repackage apps, while presenting very little computational or communication overhead, and being deployable without requiring significant changes to the Android platform.

1 Introduction

Online application stores or "markets" are becoming an increasingly important vector of software distribution. For instance, Apple's flagship MacOS X operating system is, since version 10.7, only distributed through the Apple App Store, thereby entirely forgoing the traditional distribution channel – packaged optical media sold in brick-and-mortar stores. Likewise, the Google Chrome Web Store is a consolidated place to download all extensions to the Chrome browser.

While their importance is growing, for personal computers application markets are a relatively recent development,[1] and still merely represent one of several alternatives. On the other hand, application markets have been the primary (if not the only, for most users) means of acquiring and installing software on advanced mobile devices such as smartphones and tablets.

"Official" application markets for mobile devices, such as Google Play or the Apple App Store act as a centralized software distribution point for a given platform, and allow users to find, download and install applications through a single interface.

Besides official markets, a large number of third-party (or *alternative*) markets exist. Users may rely on these alternative markets, for a variety of reasons, including the unavailability of the official market in a particular country, name-brand recognition (e.g., Amazon's Appstore), or to freely obtain applications that require payment in the official market. Some markets are also locale specific, where existing applications are modified and redistributed for localization purposes. For instance, popular applications may be translated in languages that they do not natively support.

Markets adopt several techniques to provide users with confidence that they are downloading safe applications. First, usually, applications must be cryptographically signed so that their providers are authenticated. Second, markets enforce policies to deal with malicious applications. Some markets (e.g., Apple AppStore) vet applications prior to publication [11]. Others, such as Google Play, allow relatively unmoderated publication, but react to identified malware by removing it both from the market and from all (connected) devices that have already installed the malicious application.

Unfortunately, these techniques fall short of providing strong security guarantees. When application signatures are certified by the market proprietor (e.g., Amazon and Apple markets), the user has to completely trust the market proprietor to manage and secure the certificates. The fact that existing centralized vetting systems have shown to be imperfect in keeping malware at bay [7, 8, 9] seems to indicate that the security guarantees provided by such centralized systems are relatively weak.

In Google Play, the security guarantees are even weaker. Certificates are typically self-signed and, thus, are not bound to any particular identity. Almost anybody can upload applications into the market; and it may take time to realize that some harmful applications have been uploaded. Worse, some of the third-party Android markets may not police malware at all. In fact, it may even be in a market's best interest *not* to do so, as the market operators could enjoy revenue from infected applications.

In other words, existing authentication mechanisms for market applications appear insufficient. For instance, grafting viruses onto pirated software is certainly not a new attack; yet, the lack of proper authentication allows miscreants to use such techniques to distribute malware through application markets.

In this paper, we focus on Android application markets, and present two main contributions. First, through measurement experiments, we evaluate to which extent existing markets for Android devices facilitate malware installation. We build crawling mechanisms that identify a large number (195) of existing Android application markets, and gather a total of 76,480 applications from these markets, including Google Play (35,423 applications). From this application corpus, we show that *application repackaging*, in which

[1] The AppStore first appeared on MacOS 10.6.6 in Jan 2011.

miscreants disseminate malware posing as legitimate, well-known applications, presents a significant threat. By analyzing signing strategies used in alternative marketplaces, we show that some malicious markets extensively reuse certificates to provide valid signatures on maliciously repackaged applications.

Second, we propose a simple authentication protocol for market applications, that can be immediately deployed on Android, piggy-backs on the naming conventions used for Android packages and applications, and would make it significantly more difficult for an attacker to perform application repackaging.

The remainder of this paper has the following structure. We first discuss application repackaging techniques in section 2. In section 3, we describe our measurements on the incidence of malware in current Android marketplaces, and show the threats posed by application repackaging. Then, in section 4, we provide a novel application authentication mechanism and present a proof-of-concept mobile application, AppIntegrity, which implements such a verification mechanism. We provide a security analysis of our mechanism in section 5, outline its limitations, and propose extensions to overcome these limitations. We discuss related work in section 6. Finally, we conclude with a discussion and directions for future work in section 7.

2 Application Repackaging in Android

We next summarize how application repackaging is performed in Android. To do so, we first describe the contents of an Android application, before turning to repackaging mechanics.

2.1 Android applications

In Android, applications are usually written in Java (although some have "native" C calls), and are distributed as APK (Android package) files. Those APK files are in fact Zip archives, which contain compiled Java classes (in Dalvik DEX format), application resources, and an `AndroidManifest.xml` binary XML file containing application metadata. The APK also contains a public key and its associated X.509 certificate, bundled as a PKCS#7 message in DER format.

Naming conventions. When creating a new project, the Android developer documentation dictates that a full "Java-language-style" package name be used, and that developers "should use Internet domain ownership as the basis for package names (in reverse) [4]." This creates package names such as `com.google.maps` for the mobile Google Maps application. To avoid name conflicts, package names must be unique across the entire universe of applications. Using reversed domain names theoretically limits potential namespace conflicts to a developer's own domain.

Signing applications. All Android applications must be cryptographically signed by the developer; an Android device will not install an application that is not signed. Typical Java tools, such as `keytool` and `jarsigner`, may be used to create a unique keypair and sign the mobile application.

In Android, the only key distribution mechanism used consists in bundling developer's public key with the application. Further, Android has no requirement for a keypair to be certified by a Certificate Authority (CA). In fact, we observed that more than 99% of the 76,480 applications we gathered as part of this study (see Section 3) use self-signed certificates.

In other words, the primary purpose of the keypair is to distinguish between application authors, but not to provide any stronger security properties. In practice, keypairs are also used to 1) ensure that applications allowed to automatically update are signed by the same key as the previous version, 2) potentially allow applications signed by the same key to share resources, 3) grant or deny permissions to a family of applications signed by the same key, and 4) to remove all applications signed with the same key from the Android market and potentially from all connected devices when one of these applications is flagged as malware [32].

On the other hand, due to the absence of any certification authority or PKI, signatures on Android do not provide any assurance about the identity of the signer. Shortly stated, Android ensures that the Facebook application is correctly signed by somebody, but cannot prove the Facebook company actually signed the Facebook application.

2.2 Application Repackaging

An existing application redistributed with a different signing key, often with functionality not present in the original version, is said to be *repackaged*. Some, all, or none, of the application's existing functionality can be preserved in the repackaged version.

Applications can be repackaged for many reasons other than to distribute malware. For instance, a repackager may simply wish to add advertising to an existing application to profit from somebody else's application. Application repackaging falls broadly in two classes: spoofing and grafting.

Spoofing. Mobile applications can simply be published under false pretenses, *spoofing* little or none of the features a legitimate application would possess. To deceive the user, a malicious program may advertise to be an existing application, or a nonexistent application that may plausibly exist, yet provide none of the expected functionality. As previously shown in peer-to-peer networks [15] and search-engine result poisoning [23, 27], this type of attack could flood a market with enough false positives to attract users.

As an example, in July 2011, the legitimate Netflix application only supported specific devices and versions of Android. Unsupported devices could not locate and install the official application in the market. In October 2011, a fake version of the Netflix application was published in the official Android Market, claimed to "support" all devices, and thus appeared to owners of devices that could not download the legitimate application. The fake application displayed a plausible login screen, but then simply stole service credentials. Once credentials were entered, the application uninstalled itself [10].

Grafting. To achieve the desired functionality of a legitimate application, an attacker may elect to graft malware onto an existing application, and subsequently republish the modified application.

The attacker starts by downloading and extracting an existing application. To do so, she unzips the APK archive to extract the application components (class files and manifest).

Then, she adds malware to the application, and repackages it. Adding malware may require to reverse the DEX-formatted Java classes. While not entirely straightforward, tools such as `undx` [31], `baksmali`, `dedexer`, or `ded` [18] can often successfully decompile `.dex` files to source code. DEX can also be converted to a typical Java jar collection of classes using the `dex2jar` utility, at which point a typical Java decompiler can be used.

In the quite common case in which the `.dex` file does not need to be fully reversed to source code, much of the disassembly and repackaging process can be automated. For instance, `apktool` [3] can unpack and repackage an existing `.apk` with two commands. `apktool` has several side effects that result in non-required changes to the repackaged `.apk`. For instance, some files may be compressed in the repackaged application regardless of whether or not original file was compressed. With automatic compression the repackaged file may actually be smaller than the original despite the addition of malicious code. These side effects may be undesirable for an attacker that wishes for the application to remain as similar as possible to the original application.

In addition to the class files, the attacker may need to modify the `AndroidManifest.xml`, since this is where application-level permissions are specified. This can be done using a tool such as AXML-Printer2 [5]. For instance, the malware to be added to the existing application may require the `INTERNET` or `SEND_SMS` permissions, even if the permission is not specified in the original application.

Last, prior to publishing a the new application to one or more markets, the attacker must cryptographically sign the application. The signing can easily be performed with standard Java tools, e.g., using `jarsigner`. Since Android uses self-signed certificates, such signatures will pass installation-time checks.

3 Measuring the prevalence of malware in markets

Each market may have different policies for policing applications. Google maintains a reactive policy in the official Android market, but alternative markets may have a less effective policing policy or no policy at all. To demonstrate the threat of application repackaging we investigated the presence of repackaged applications in existing markets. In this section we discuss our measurement methodology to create a corpus of alternative market applications, how we created a corpus of official market applications, and a description of the resulting corpora.

3.1 Collecting applications in alternative marketplaces

In order to create a corpus of applications from alternative markets, we first conducted an experiment to observe alternative distribution mechanisms.

We identified 194 alternative marketplaces by popularity based on search engine results. First we created a list of candidate site seeded with search results for "alternative market android","third party android market","free android applications","android app store", and simply "android market." We then expanded the list of candidates to include the same strings translated to all 63 languages currently supported by Google Translate. We manually inspected search results to prune candidate sites that did not actually deliver mobile applications (many only offer meta-data, directing an interested user to then download the application from the official market). During manual inspection of search results, we appended obvious links to other marketplaces to the list of markets. Perhaps the most important observation from the search results inspection is that, unlike the official market, applications from some alternative markets can currently be downloaded using common, unauthenticated HTTP methods. In many cases the URL for applications is highly predictable and can facilitate complete coverage, as in `http://yadroid.com/?download=n` where `n` is between 1 and 2696.

While an Android application is a Zip archive packaged in a certain way (see Section 2), there is no guarantee that a given marketplace delivers applications to the device in this form. In fact, we observed many other methods. For example, one site delivers applications as expected, but the file extension is "`.ipa`" instead of "`.apk`." Similarly other sites also deliver the application archive as expected but with no file extension all, and are instead accessed by a URL such as `http://yadroid.com/?download=260`. Yet others will "double package" the application for delivery, resulting in a Zip (or other) archive that contains a Zip file that is an application.

We performed recursive decompression of archives, and tested each file we eventually obtained to determine whether it was a ZIP file that contained the `AndroidManifest.xml` binary XML file. If so, we classified it as a valid Android application.

Corpus size. We used the above described collection and pruning process to collect 41,057 applications from 194 alternative markets in October 2011. The identification of markets and subsequent downloading of applications is biased by popularity, both by using search results to identify marketplaces, and by the likelihood that popular applications are made easier to locate by each marketplace interface.

3.2 Collecting applications from the official Android market

The official Android market is intended to only be accessed directly from Android devices. Even when "installing" an application using the online interface at `market.android.com`, a signal is pushed directly to a device associated with logged-in user.

Even thought the Android framework is open source, many software components found on Android devices are proprietary, including the official Android market. We created a protocol-compatible tool that facilitates granular access to applications in the official market. We designed the tool iteratively by reverse-engineering the market components found on an Android device, observing network traffic during a market transaction, and by observing server responses after manually adjusting protocol parameters.

The official market protocol requires authentication and a device identifier. To authenticate to the service, we created a new user using an actual smartphone, and extracted the device identifier from the device. The username, password, device identifier, and SDK version are used to establish a market session. This session is similar to the session status that would occur when opening the "Market" application, the official client, on an Android device.

Once a session is established the server is queried for a list of application categories (e.g., Finance, Education, Medical). Much like the official client, server results vary depending on several parameters. By manipulating these parameters, a client can obtain different results to mimic views present in the official client such as "Featured" or "Top Free" in any category. For example, applications can be ordered by any of POPULAR, NEWEST, FEATURED, NONE or the field can be omitted.

In addition to mimicking the capabilities of an official client, we are able to manipulate additional parameters, such as the wireless carrier associated with this client. A physical device is typically associated with a single carrier and the official client simply utilizes the carrier associated with the device. We can enumerate sets of carriers impersonating devices on several networks by altering the Mobile Country Code (MCC) and Mobile Network Code (MNC) as defined in ITU E.212. For example the United States has MCC's 310-316, and MNC 260 specifies T-Mobile. We can iterate 310012, 310410, 310120, and 310260 to impersonate devices from Verizon, AT&T, Sprint and T-Mobile, respectively.

The ability to impersonate devices from various networks is important for coverage as some applications may only be made available to customers on certain networks. Likewise, certain applications only exist for certain types of device hardware, geographic region, and software versions of Android. Any particular combination of market parameters will currently return a maximum of 800 results, so that it is not possible to simply iteratively collect all applications in the entire market.

The actual applications are not downloaded through the existing market session. Market query results contain meta-data about applications. Some of this meta-data is available in the official client such as the title, cost, and review ratings. Other meta-data is not visually displayed, such as the application's AssetID. The AssetID is needed to download applications independent of the existing market session. The AssetID is approximately 20 ASCII digits long and must be precisely specified in order to download the application.

When attempting to download too many applications over a period of time a the server may not permit further downloads, temporarily blocking connections. We observed that HTTP 503 errors precede such blacklisting, and accordingly implemented a back-off procedure.

Corpus size. To create our corpus, we collected free applications from each application category as accessible from the United States on four carriers: Verizon, AT&T, Sprint and T-Mobile. We additionally iterated through known Android and SDK versions, eventually collecting 35,423 applications in October 2011. Because we collected by category, the 35,423 collected applications are biased by popularity in each category. Furthermore, due to the complexity of automating the payment protocol, we only collected free applications.

3.3 Results

We next discuss the prevalence of malware in the marketplaces we have measured. From our collected corpora, we identify known malware attributable to each marketplace. We also offer particular measurements (e.g., on certificate reuse) from the corpus to inform discussion of the protocol provided in Section 4.

To determine a lower bound on malware present in each market, we scanned each file with multiple antivirus products, through the VirusTotal interface [1].[2] VirusTotal is a service that offers file scanning through 42 different antivirus products by vendors such as Symantec, McAfee, Kaspersky, and TrendMicro. Despite the large number of antivirus products being used, malware detection in this manner remains a very conservative lower bound as mobile malware detection is less mature than desktop malware detection. Some reports show that many mobile anti-virus detection rates very low: between 0 and 32% [28]. It is highly unlikely that any so-called "zero-day" malware would ever be detected under this procedure. Therefore, the actual delivery of malware from marketplaces is quite likely a larger problem than we show.

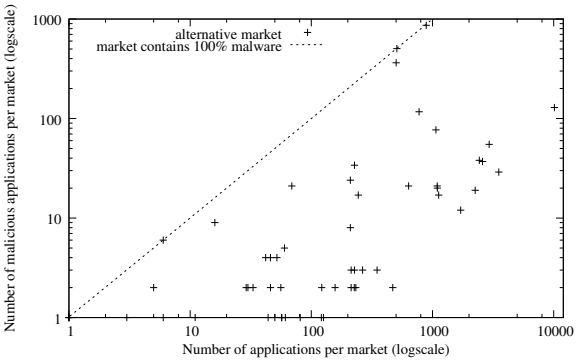

Figure 1: **Alternative Market Malware**: Total applications and detected malware per market. Each point corresponds to one measured market. Points closer to the dashed line deliver higher percentages of malware.

Some alternative markets appear to be completely absent of malware, but a few markets distribute malware almost exclusively. In the scatterplot of Figure 1, each market we crawled corresponds to one point, whose x-coordinate denotes the total number of applications in that market, and whose y-coordinate denotes the number of applications detected as malicious in this market. The dashed line in Figure 1 represents the threshold where *every* application sampled from a market would be detected as malicious (and hence, no point can be above that line). Several points approach this line, demonstrating that our naive sampling identified a number of markets which almost exclusively distribute malicious applications. Particularly preoccupying is the case of the markets in the top right corner of the graph. Not only do these markets have very high percentage of infected applications, but they also provide a large number of applications.

We can further use this data to attempt to exhaustively classify all Android malware as repackaged or some other type of malware. After eliminating "potentially unwanted programs" detected by anti-virus, such as spyware, we can concisely catalog all existing malware as of November 2011. We observed 55 different families of malware, 40 of which (or 73%) (such as those enumerated in [20] and more exhaustively in [36]) employ some type of repackaging or spoofing. We note that many early Android malware families as well as the most recent employ some type of repackaging.

[2]Due to limitations to the VirusTotal interface, applications larger than 20 MB were not scanned.

As another datapoint, in [36], the authors use a combination of package name comparison to applications found in the official market and manual analysis to classify repackaged applications. They similarly find that 86% of unique samples in their Android malware corpus are repackaged. The reported number is different from our observation for two reasons: First, in [36] the described corpus is entirely comprised of malware samples, many of which belong to the same family leading to a non-uniform distribution across malware families. Second, the definition of *repackaged* in [36] is slightly different and does not include the category we term *spoofed* in Section 2.2.

We next look at possible indicators of malware distribution strategies. We first measured the number of applications which provide package names that form valid domains. To do so, we parse each package name with the Perl module `Data::Validate::Domain`. We find that 83% percent of the applications originating from the official marketplace have package names that, when reversed, represent a well-formed domain, following Google's suggestions (see Section 2) for package naming conventions. Interestingly, applications from alternative markets exhibit a slightly higher rate at 86%. This seems to indicate that exotic naming conventions are not indicative of malware. On the contrary, we found that, in markets with the highest percentage of malicious applications. applications tend to comply *more* with the proposed standard naming conventions. In hindsight, this does not come as a surprise: malicious applications designers have incentives to make their applications "blend in" as much as possible.

Figure 2 shows the distribution of application sizes for the official and alternative Android markets. Approximately 40% of all applications are greater than 1 MB. Alternative market applications are generally slightly larger than those in the official market, but the size distribution between the market types is clearly similar.

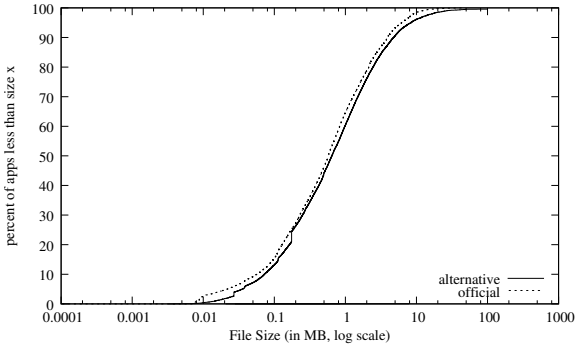

Figure 2: **Application Sizes**: CDT of market applications

Finally, we attempt to characterize which signing strategies are being used in alternative marketeplaces. The lack of a PKI and general lack of proper certificate validation does not encourage adoption of best practices. Indeed the near ubiquitous use of self-signing certificates enables the publisher to adopt a number of different signing strategies. For instance a publisher could use the same certificate to sign every application they publish, or could use a different certificate for each version of each application. Shortly stated, certificates do not provide any guarantee on the application integrity, or origin, and patterns of certificate misuse may be evidence of application repackaging.

We observed heavy re-use of signing certifications in our collection. Indeed, only 52% of certificates from the official market were unique. The distribution of certificates is not uniform, as some certificates are used as many as 693 times, while others are only used once. In alternative markets, the signing strategies vary drastically. Some markets exhibit distributions similar to the official market, while others use a single

signing certificate.

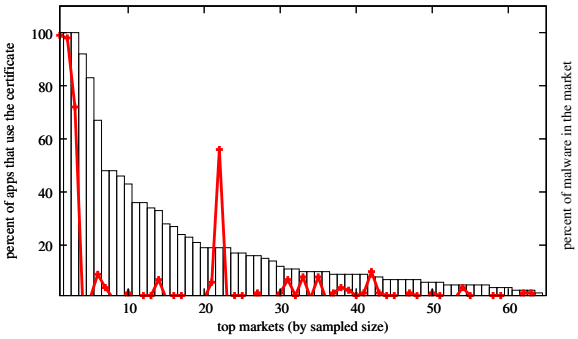

Figure 3: **Certificate Reuse in Alternative Markets**: Percentage of apps which share the same certificate (bars) overlaid with the percentage of malware (line plot). Each point corresponds to one measured market. Markets are ranked by decreasing reuse of certificates. Only the top 64 markets (in number of applications) are presented.

Figure 3 plots, for each market among the 64 markets that distribute the most applications, the highest percentage of applications in the market that are signed using the same signing certificate. We overlay the plot with a line showing the corresponding percentage of malware in the same markets. Strikingly, almost all the alternate marketplaces with high percentage of malware appear to significantly re-use signing certificates. Calculating the Pearson correlation coefficient between the percentage of malware, and the percentage of certificate reuse, across all markets yields $\rho \approx 0.64$. In the seediest markets with close to 100% known malware, *all* applications are signed using the same signing certificate. In this case, the remaining applications to make up the 100% are quite likely malware that is not yet detected by anti-virus.

In sum, these results provide clear evidence that malware in alternative markets is a problem we cannot neglect. As a point of comparison, in the official market, we discovered 119 applications containing malware, or 0.003% of all applications we surveyed. While certainly very low, this number needs to be taken with caution: these are the applications that were detected by anti-viruses as being malicious, and are therefore a strict lower bound on the total amount of malware actually present in the official Android market.

4 Toward Application Verification

Regardless of application vetting policies, it is possible that an application can be repackaged and published into marketplaces that users will frequent. Yet users have no way of knowing if an application claiming a particular origin is in fact created by the assumed author. Here we present a very simple protocol for end-to-end application verification, and discuss an example implementation. The idea behind the protocol is that, while not a panacea against all attacks (see Section 7) it raises the bar that attackers have to clear to be able to carry out spoofing attacks, while being essentially freely deployable with the current Android infrastructure.

In the context of this paper, verification means that the application is *authenticated*, and that, as a result, its *integrity* against repackaging by third-parties is guaranteed.

4.1 Protocol

Prior to publishing, an application must be cryptographically signed. This signing makes use of the private key of a keypair generated by the developer. The existence of a keypair provides developers and users with the primitives required to perform other PKCS actions. In particular, the protocol described below takes advantage of the well-known ability to verify a signature. That is for a keypair: secret signing key ssk and public verification key pvk, that the signing of data d results in signed data sd:

$$sd = sign_{ssk}(d)$$

Furthermore, that signed data can be verified using only the associated public key:

$$verify_{pvk}(sd) = true$$

It is assumed that the ssk is selected uniformly at random from the set of all possible keys, and that without the ssk it is computationally infeasible create an sd' that can be verified.

If the developer makes the pvk widely available, it can be used to locally verify signed applications. In order to deter developer impersonation in repackaged applications, the pvk should *not* be published via the marketplace from which the application is obtained. If this were permitted, unscrupulous persons could simply continue to repackage applications and provide new pvk' keys along with new applications when published. Instead we propose that the author's pvk be stored in a predefined location on the author's web server or use methods similar to Domain Key Identified Mail (DKIM) [25] to provide the pvk via DNS (or both). In both cases the verification is tied closely to the DNS controlled by the publisher. Again, to deter repackaging, this DNS location must not be specified in a hidden manifest, but must be closely coupled to information presented to the user. As mentioned in Section 2, application package namespaces "should use Internet domain ownership as the basis... (in reverse)." Therefore, by reversing the package name, a URL can be constructed to locate the pvk.

If developers honor the direction to name packages appropriately (83% of applications in the official market already conform to this convention), the pvk can be unambiguously located relative to the URL corresponding to the package name. Suppose that, by convention, the pvk is stored in a file `android.cert` at the root of the domain. For instance, an application with the package name `com.facebook.katana` would be signed using an ssk that has an associated pvk available at `http://facebook.com/android.cert`. The use of domain ownership for key publishing permits the use of self-signed certificates making this protocol immediately deployable. The use of CA's and other PKI infrastructures remain (at this point) optional.

Propagating key information via DNS has certain performance benefits compared to storing the public verification key on a webserver, which must serve the key to every mobile device at verification time. Both methods are however susceptible to various attacks which are discussed in Section 5.

By decoupling cryptographic signing from the distribution mechanism, the application can be verified independent of how an application is obtained. Applications obtained via unmoderated file sharing forums, will still bear a package name plausible to the user (e.g., `com.facebook.katana`), and the device will attempt to verify the application with the legitimate Facebook pvk. If the application had been repackaged, the verification will fail. If the verification succeeds, it was signed by the owner of the `facebook.com` domain.

Application verification should take place as part of the installation process. Install-time verification can prevent undesirable applications from ever executing on the mobile device. Figure 4 shows a timing diagram of the entire verification process. As previously described, the publisher's keypair is created orthogonal to

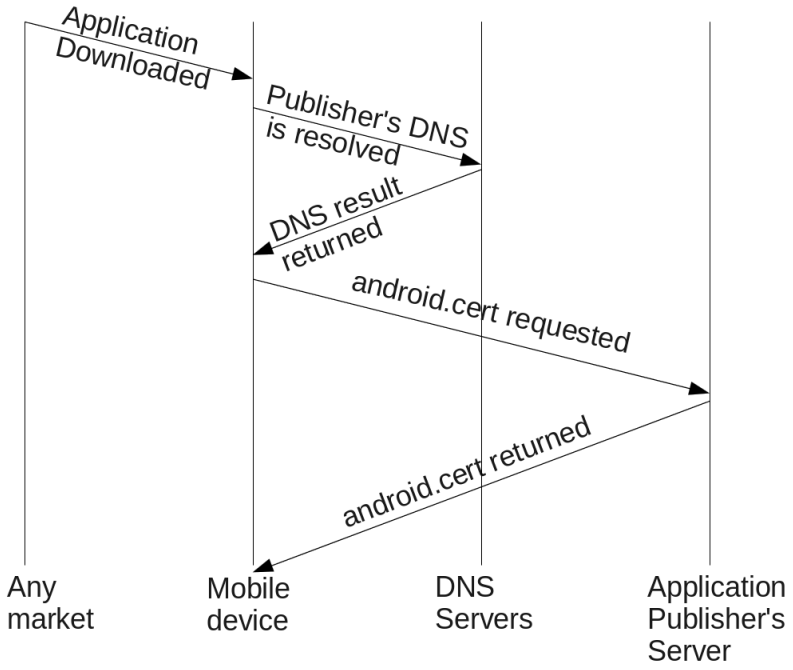

Figure 4: **Verification protocol**: Network Diagram

verification. First a user locates an application in some mobile marketplace and the application is downloaded to the mobile device via whatever mechanism the marketplace supports. Once the entire application is downloaded, the embedded signatures can be checked to be well-formed (locally on the mobile device). Next the package name is extracted from the application and reversed in order to determine the location of the pvk. The pvk is retrieved from the publisher's server (or from DNS). The application is then verified using the pvk. If the verification succeeds the application is installed using the typical platform installation process. If verification fails, the application is not installed. Of course, when applications fail verification the protocol could be modified to permit the user to install the application anyway, or to upload the file to a security team for analysis.

Publishers may need to update to a new, legitimate keypair resulting in a new pvk becoming available. Similarly, if a publisher's ssk is compromised a repackaged application may be installed with an pvk that is thought to be valid at install time, but was later found to be compromised. For these reasons, application verification may also be performed periodically or on-demand. Similar to the end result of failed verification at install time, a failed verification in this case would likely result in the uninstallation of an application.

A verification process such as described here is independent to any vetting process imposed by a market policy. The cryptographic verification simply demonstrates that an application is what the publisher intended to provide to the consumer. It makes no attempt to determine if the behavior of an application is malicious.

With this protocol the end device is able to verify that the application is exactly what the publisher intended for the user. In the physical world, in addition to the trusting integrity of the store, there is some independent binding to the creator of the software. This binding takes many forms such as product packaging, branding, holographic CDs, and other anti-piracy technologies. The protocol we propose, called AppIntegrity, enables similar binding to take place on modern application markets, creating a way to bind a website owner to a particular application.

Publishers currently do not make the pvk available, as a consequence it is not possible to fully test the mitigation capabilities of AppIntegrity. However, our measurements (by domain validity checks and manual malware analysis) and the repackaging classification techniques in [36] suggest that AppIntegrity may see great success in mitigating current malware.

4.2 Implementation

The protocol described in Section 4.1, is realized in proof-of-concept applications designed to run on any computer and as an Android application, which we call AppIntegrity. Android's architecture permits the entirety of an application to be observed by other applications. Accordingly, AppIntegrity registers a handler for the PACKAGE_ADDED[3] intent that performs verification whenever new applications are downloaded. Since most publishers have not made public keys available, a failed verification results in giving the user the choice to uninstall the application.

AppIntegrity takes advantage of several Android features:

1. Application package names are intended to be unique and based upon domain ownership of the developer.

2. Android applications have read access to other applications. While each application can store data in a private area, the application itself may be read by other applications. Thus a verification program has the ability to obtain package name and signature information from other applications.

3. Android applications are written in Java which has extensive cryptographic libraries that can be used to verify signatures.

4. The Android documentation specifies RSA when generating a private key. The use of RSA in key creation results in a SHA1withRSA (see Figure 5) signature, which is compatible with the existing specification for DKIM (DKIM is defined to use RSA-SHA-1 or RSA-SHA-256 for signing and verification). As seen in Table 1 the majority of applications we observed use one of the two DKIM compatible algorithms.

```
$ keytool -printcert -file CERT.RSA
Owner: CN=First Last, OU=Unk, O=Unk, L=City, ST=State, C=US
Issuer: CN=First Last, OU=Unk, O=Unk, L=City, ST=State, C=US
Serial number: 4d895f96
Valid from: Tue Mar 22 22:48:54 EDT 2011 until: Sat Aug 07 22:48:54 EDT 2038
Certificate fingerprints:
 MD5:  07:E4:51:41:E8:80:92:97:F9:6F:AF:BF:57:2F:28:2A
 SHA1: D5:A0:3D:A4:E5:0F:D7:9E:B3:53:95:83:8C:CA:AB:A5:EB:E2:C4:29
 Signature algorithm name: SHA1withRSA
 Version: 3
```

Figure 5: **Android signing key**: keytool output for signing key

Future implementation could take many forms. To fully realize the protocol shown in Figure 4, the Android framework needs minor modification to the install process. Either verification must be built into the package installation process or a new intent needs to be broadcast post-download prior to package install. Meanwhile, the proof-of-concept application may be downloaded from (anonymous URL). Device carriers

[3]Note that this isn't exactly the same as the described protocol, since the application is technically already installed by the time the PACKAGE_ADDED intent is broadcast. This intent is the nearest to the desired functionality that a typical, unprivileged application can achieve.

Algorithm	Official	Alternative
MD5withRSA	9.784%	7.553%
SHA1withDSA	2.662%	2.743%
SHA1withRSA	87.458%	87.157%
SHA256withRSA	0.091%	2.543%
MD2withRSA	0.005%	0.004%

Table 1: Signing Algorithms observed in markets

or manufactures may choose to install a verification application in such a way that the user can't uninstall it, forcing verification to occur.

4.3 Performance Evaluation

Minimal network overhead is crucial as many carriers now have limited data plans. Currently, a typical RSA key found in the official market averages 922 bytes (0.0008 MB). Given the current distribution of application sizes (as shown in Figure 2), the additional network overhead introduced by verification is marginal. Figure 6 is scaled to show the only appreciable overhead introduced by obtaining the certificate. As seen in the Figure 6, less than 4% of applications would exhibit significant overhead relative to downloading the application. The applications in question are simply so small that the additional 922 bytes is significant, however it is extremely likely that the user is already downloading many other [24], larger applications further reducing any concerns of network performance degradation.

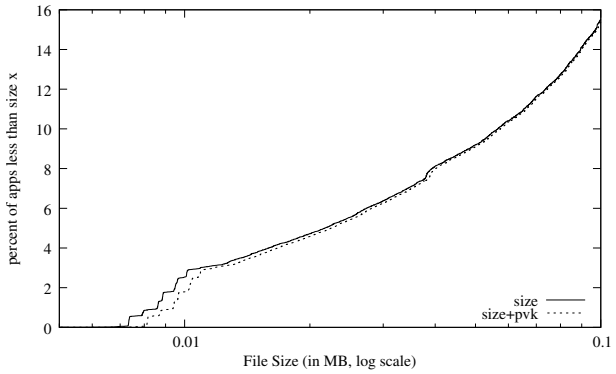

Figure 6: **Size Overhead**: Less than 4% of applications would exhibit appreciable overhead (plot magnified from official market plot in Figure 2).

Similarly, processor use directly affects battery life on mobile devices [11], and as such, excessive resource use could hinder adoption. Since devices already perform cryptographic signature verification the additional verification is not significantly different. Currently a manifest (different that the `AndroidManifest.xml`) is stored in a special `META-INF` directory along with the public keys in the `.apk`. The device currently verifies the signatures stored in the manifest using the embedded public key. With our proposed protocol the same key would be obtained dynamically, but the cryptographic operations would remain the same.

To encourage adoption, public keys could still be included in the `.apk` files, and actually both keys (which may be identical) could be used to verify the application. This additional verification would result in a

linear increase in processing time as each `.apk` component is verified twice. If the protocol is integrated into the Android package installer, there would effectively be no additional overhead over the existing installer.

The lightweight cryptographic verification of AppIntegrity will likely outperform other types of "fingerprint" or "signature"(not cryptographic signature) based security solutions. In particular, anti-virus, symbolic execution, anomaly detection [14], static analysis [29, 30] would all likely require extensive processor and/or memory requirement which are not desirable on a resource-constrained mobile device.

Since little or no modification is required to to the Android framework, there is negligible network and processing overhead, and there is no additional burden to implement a PKI, AppIntegrity can be deployed to Android with little cost.

5 Security Analysis

The primary benefit of AppIntegrity is the ability to verify the integrity of a published application independent of how the application is obtained. Under the current model, an attacker needs only to succeed in getting malware onto a device. Typically this is achieved by publishing a malicious application to a marketplace and allowing users to locate and install the application. AppIntegrity significantly increases the effort required for a successful attack. Under this new model, the attacker must also either obtain the original publisher's secret signing key, be in control of the publisher's web server, or commit a man-in-the-middle (MitM) attack on the publisher's DNS records and/or web server. In all cases the attacker must now conduct two successful attacks, and the secondary attack requires more effort than application repackaging.

Man-in-the-middle attacks that target the mobile device may be more difficult to conduct than such an attack on a traditional computer. Modern smartphones and tablets can communicate over several medium. A successful MitM attack on the client will either need to predict the specific media that will be used, or will need to conduct simultaneous MitM attacks on all nearby WIFI, 2G, 3G and 4G networks.

The official market enforces unique package names, which incidentally lightly deters the republication of repackaged applications back to the market. An application with exactly the same package name may not be published. In order to republish in the official market, some existing malware, such as DroidDream-Light2, uses capitalization differently in package names. For example, `com.gb.CompassLeveler` vs `com.gb.compassleveler`. Since DNS does not preserve case, the AppIntegrity verification would resolve to the legitimate key, and fail.

5.1 Limitations

AppIntegrity would benefit from a few minor design changes to the Android platform: permitting additional privilege to the verification software, enabling actions to perform prior to application install and clearly displaying package name information to users prior to install. The proposed protocol does make several assumptions about the user, and if the user is deceived the effectiveness of AppIntegrity suffers.

Domain name deception. Once a user has located a particular application that they are interested in installing, the installation interface must clearly display the package name (or derived domain) to the user. The user may or may not recognize the application name, package name, developer, etc. Even when the user does recognize the application name and developer, it is up to the user to detect *typosquatting*, the intentional registration of domain misspellings [26]. For example, if the user installs a repackaged Google maps look-alike that has a package name `com.g00gle.maps` the certificate will be retrieved from `maps.g00gle.com` and will cryptographically verify. Without an external validation service for URLs (e.g., PKI, reputation system), such attacks will remain possible.

Domain recognition. Similarly, many users recognize names such as Google, Facebook, etc but the vast majority of applications are created by less recognizable publishers. One may argue that as an application becomes popular, users are more likely to recognize the publisher (and associated domain). However, the problem of unrecognized publishers remains. AppIntegrity provides a foundation that could be used to create additional protocols or services to help solve this problem. For instance, AppIntegrity would provide assurance that a given publisher produced a certain application, and an external vetting service could assist in confirming that this publisher is reputable.

To address both domain name deception and domain recognition, one could reasonably imagine such a service building upon Perspectives [35], with notaries voting on the reputation of a given publisher. Such an architecture is already deployed as a Firefox browser add-on (Convergence, [33]), and the same functionality could probably be implemented on Android devices.

Lack of Privilege. The current AppIntegrity application could be uninstalled by a user or potentially by malware. A previously mentioned, a manufacturer or wireless carrier could install AppIntegrity in a way that makes user uninstallation difficult. However, as with most security properties, *rooting* or *jailbreaking* the device undermines this added security.

Prior Infection. Devices that are already infected with malware that has elevated (root) privileges are subject to other attack classes, such as drawing over the existing user interface. In these situations, AppIntegrity only assists in preventing malware from entering the device, and is subject to all the same issues as typical software.

5.2 Keeping Private Keys Private

As with most PKCS structures, the cryptographic properties provided by the keypair require the private key to remain secret, known only to the owner. Any other party that knows a user's private key can impersonate that user. For these reasons, users typically create their own cryptographic keypair. Contrary to this convention, Amazon's Appstore (one of the commercially-backed Android markets) supplies an account-unique key to the publisher [2]. In this model, Amazon could impersonate any application publisher, and a security breach of the Amazon market would result in all keypairs being compromised. We encourage developers to exercise the option to request the use of a non-assigned key for application signing.[4] As stated in section 4.1, to enable legitimate verification, public keys should not be stored alongside applications in a marketplace.

Similarly, smartphone or tablet users that have "rooted" their device often install entire new operating system images known as "custom ROMs." These ROMs are created and made available by enterprising developers such as the Android Open Source Project (AOSP). The developers of these ROMs may chose to publicly publish associated private keys. Since the private keys are widely available, no identity can be bound to anything signed by the key. Malware, such as jsmshider [6], may take advantage of this cryptography faux-pas.

Under a model that encourages self-signed certificates, such as the current Android model, the burden of securing the private key falls solely on the publisher. Application publishers that do not properly secure their secret signing key risk others using their identity to publish applications. Under the protocol outlined in section 4.1, loss of the private key would allow an attacker to modify an application and have the modified application successfully verify.

[4]Developers may request to use a non-Amazon key by submitting a request through the Amazon AppStore Developer Portal

6 Related Work

Spoofing attacks similar to the Netflix malware were theorized by Felt et al. in [21]. Felt et al.'s work [21] predates the recent Netflix spoofing malware which very closely mimics the Facebook attack described in the paper. Additionally Felt et al. also provide a survey of much of the mobile malware discovered form 2009 to 2011 in [20].

In [34], Vidas et al. observe application repackaging as one type of an "unprivileged attack." The class of unprivileged attacks is one part of a greater taxonomy targeting Android devices. Vidas et al. also observe that malware is often present in alternative ("black") markets and such applications often "offer no additional value to the consumer." These observations are confirmed in our alternative market corpus discussed in Section 3. Burguera et al. also cite the repackaging and distribution in alternative markets as evidence for the need of their primary contribution in [12], which is a system for crowd-based behavioral malware detection. Both Vidas et al. and Burguera et al. observe that applications are signed and the current signing process in no way inhibits repackaging and republication of applications.

Zhou et al. conducted alternative market research focusing on four alternative marketplaces [37], and have found a significantly smaller percentage of malware than we observed. Different from Zhou et al.'s study, we investigate a larger number of application marketplaces, and look at possible indicators of suspicious markets (e.g., extensive reuse of identical signing certificates). In another difference, Zhou et al. present a new tool for re-actively detecting malware on found in markets, where AppIntegrity strives to proactively prevent the installation of software signed by those other than the originator.

In [36], Zhou et al. describe a malware collection consisting entirely of Android samples. The authors provide measurement of the malware collection and describe the "evolution" of malware by studying related samples chronologically. The authors also find a large amount of application repackaging and provide measurements of activation mechanisms, secondary payloads, and permission used by malicious applications.

Chen et al. [13] use application metadata to identify web applications which the authors then provide a means of app isolation. Chen et al. reference the Chrome Web Store which allows "verified apps." The procedure for obtaining the "verified" icon in the store is to pay a $5 fee and the application developer must verify domain ownership via Google's "webmaster tools." The term "verified" is used differently here, as the verification is proven to the market which then assures the consumer. The additional assurance provided by proving domain ownership is likely useful as a means to increase application use and market reputation, but is somewhat orthogonal to the end-to-end integrity provided by AppIntegrity.

Enck et al. describe a lightweight application certification service, Kirin [19]. This service forces applications to pass several rules at install-time, such as the absence of permission combinations the rule creator deemed dangerous. AppIntegrity could possibly be implemented as a feature of Kirin, or independently as described above, in addition to Kirin.

AppIntegrity focuses on ensuring end-to-end integrity for applications, and makes no attempt to analyze the inner workings of an applications or otherwise protect the user from applications that are malicious from origin. For this reason it makes sense to pair AppIntegrity with taint tracking systems such as TaintDroid [17] or PiOS [16] in order to detect privacy leaks. Similarly, Hornyack et. al have retrofitted Android [22] in a way that permits executing existing applications in a safe way.

7 Discussion and Future Work

AppIntegrity does not require any changes to the current developer build process for Android. The application structure and cryptographic signing are used in exactly the same manner as currently employed.

Similarly AppIntegrity is designed to make use of the self-signed keys widely used by Android developers. As shown by the reference implementation provided for Android, AppIntegrity can be immediately adopted without the need of a large PKI system.

If an entity desires the added value of a trusted third party verifying developer entities, a PKI can be applied in addition to the protocol described in Section 4.1. Again Android has features that facilitate this, as applications can be signed by multiple keys, allowing for an application to be signed by a market proprietor in addition to the developer. Additional signing by the market proprietor is akin to physical store reputation, in both cases the market is certifying that the software obtained from the market is legitimate. Of course, a more traditional PKI model could be imposed where a developer key is signed by a third party who also maintains a registry of developer public keys.

We hope that developers elect to publish their public keys as we describe in Section 4.1. In order to encourage adoption, we hope that Google will adjust the Android developer documentation and effectively make public key publication part of the standard developer account setup.

AppIntegrity is still compatible with alternative markets. Applications that are republished via alternative markets can be downloaded and verified by a user who can be confident that the installed software is what the developer intended for delivery. Similarly, AppIntegrity would be compatible with "private markets" given that the devices that have the private markets provisioned have network access to protected domain spaces. Consider a "secure Android" under development by a government entity, as long as devices can access the government network (via VPN for example), certificates can still be retrieved from the appropriate URLs and verification can be performed.

Even though our reference implementation and related discussions largely focus on Android, AppIntegrity can leverage existing application signing for many mobile platforms, and indeed application delivery mechanisms for traditional PCs. The most common mobile consumer platforms: Android, iOS, and Symbian all already use application signing in some way, and can benefit from a verification system like AppIntegrity.

Throughout this text we have framed the use of AppIntegrity on terminal devices, such as smartphones. Additional verifiers could be used to embody trust in other ways. For example, a third party could monitor all the public keys found in applications and resolve and verify these keys with the keys found the applications respective domains. The third party could provide a "verified" seal similar to services for websites available today. Similarly a market might proactively and/or periodically verify applications that are submitted for publication.

AppIntegrity relies upon public keys being readily available and bound to an entity via domain ownership. The availability of these keys could complement other application market functions such as application revocation. Currently when malware is identified in the Android market, both the publisher and the consumer are at the mercy of Google to remotely uninstall applications from infected devices. By extending our presented protocol to verify applications prior to execution, disabling of malware can be performed by either the market proprietor (e.g. Google) or by the domain owner (e.g. the publisher).

8 Conclusion

Application markets are now commonplace for mobile devices. We have shown that not all markets are created equal: quite the opposite, in fact, as some distribute malware almost exclusively. Most of this malware is *repackaged* in some way giving victims something desirable to execute.

By analyzing signing certificate strategies we observed that markets that deliver the highest percentage of malware are also those that reuse signing keys the most, unilaterally across the marketplace in fact.

In order to mitigate the threat of repackaging, we present an end-to-end verification protocol, AppIntegrity, that facilitates cryptographic verification between the software creator and the end consumer. The protocol is realized in reference implementations for PC and Android devices, but is applicable to other mobile frameworks and application markets.

The cost of adoption for AppIntegrity is very low. The minimal network and local resource use is ideal for the constrained environment of mobile devices. Furthermore, the end-to-end protocol can be used with existing official and alternative markets alike.

Relating to Android in particular, AppIntegrity requires no changes to the existing Android development process. Minimal changes to the Android framework could enhance the ability for AppIntegrity protect users, but even when used with the current version of Android, AppIntegrity can provide added safety by rapidly uninstalling unverified applications, and providing building blocks for future protocols and services. By binding public keys based on domain ownership, AppIntegrity has the ability to leverage PKCS without the need for a complicated PKI, further contributing to making AppIntegrity rapidly deployable.

Acknowledgments

The authors thank Mila Gorodetsky for her assistance with certain foreign markets. This work is supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and by the National Science Foundation under IGERT award DGE-0903659. The authors also wish to thank the proprietors of VirusTotal for access to VirusTotal's resources.

References

[1] Virustotal. http://www.virustotal.com.

[2] Amazon appstore frequently asked questions. https://developer.amazon.com/help/faq.html, Oct. 2011.

[3] android-apktool: a tool to reverse engineer Android apk files, 2011. https://code.google.com/p/android-apktool/.

[4] Android developer guide 4.0 r1. http://developer.android.com/guide/topics/manifest/manifest-element.html, Oct. 2011.

[5] android4me: J2ME port of Google's Android, 2011. https://code.google.com/p/android4me/downloads/list.

[6] Security alert: Malware found targeting custom roms. http://blog.mylookout.com/2011/06/security-alert-malware-found-targeting-custom-roms-jsmshider/, June 2011.

[7] ASRAR, I. Could sexy space be the birth of the sms botnet? http://www.symantec.com/connect/blogs/could-sexy-space-be-birth-sms-botnet, July 2009.

[8] ASRAR, I. A touch of mobile threat dj vu. http://www.symantec.com/connect/blogs/touch-mobile-threat-deja-vu, Feb. 2010.

[9] ASRAR, I. Will sms bring you free vouchers? http://www.symantec.com/connect/blogs/will-sms-bring-you-free-vouchers, Apr. 2010.

[10] ASRAR, I. Will your next tv manual ask you to run a scan instead of adjusting the antenna?, Oct. 2011.

[11] BECHER, M., FREILING, F., HOFFMANN, J., HOLZ, T., UELLENBECK, S., AND WOLF, C. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proc. IEEE Symp. on Security and Privacy* (2011), IEEE, pp. 96–111.

[12] BURGUERA, I., ZURUTUZA, U., AND NADJM-TEHRANI, S. Crowdroid: behavior-based malware detection system for android. In *Proc. ACM work. Security and privacy in smartphones and mobile devices* (2011), ACM, pp. 15–26.

[13] CHEN, E., BAU, J., REIS, C., BARTH, A., AND JACKSON, C. App isolation: get the security of multiple browsers with just one. In *Proc. ACM CCS* (2011), ACM, pp. 227–238.

[14] CHENG, J., WONG, S., YANG, H., AND LU, S. Smartsiren: virus detection and alert for smartphones. In *Proc. ACM MobiSys* (2007), pp. 258–271.

[15] CHRISTIN, N., WEIGEND, A., AND CHUANG, J. Content availability, pollution and poisoning in file sharing peer-to-peer networks. In *Proceedings of the 6th ACM conference on Electronic commerce* (2005), ACM, pp. 68–77.

[16] EGELE, M., KRUEGEL, C., KIRDA, E., AND VIGNA, G. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Network and Distributed System Security Symposium* (2011).

[17] ENCK, W., GILBERT, P., CHUN, B., COX, L., JUNG, J., MCDANIEL, P., AND SHETH, A. TaintDroid: an Information-Flow tracking system for realtime privacy monitoring on smartphones. In *OSDI 2010* (Vancouver, BC, Canada).

[18] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of android application security. In *Proc. of the 20th USENIX Security Symposium* (2011).

[19] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On lightweight mobile phone application certification. In *Proc. ACM CCS* (Chicago, IL), pp. 235–245.

[20] FELT, A., FINIFTER, M., CHIN, E., HANNA, S., AND WAGNER, D. A survey of mobile malware in the wild. In *Proc. ACM work. Security and privacy in smartphones and mobile devices* (2011), ACM, pp. 3–14.

[21] FELT, A., AND WAGNER, D. Phishing on mobile devices. In *IEEE Workshop on Web 2.0 Security and Privacy* (2011).

[22] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proc. ACM CCS* (Chicago, IL, 2011), ACM, pp. 639–652.

[23] JOHN, J., YU, F., XIE, Y., ABADI, M., AND KRISHNAMURTHY, A. deSEO: Combating search-result poisoning. In *Proceedings of USENIX Security 2011* (San Francisco, CA, Aug. 2011).

[24] KELLEY, P., CONSOLVO, S., CRANOR, L., JUNG, J., SADEH, N., AND WETHERALL, D. An conundrum of permissions: Installing applications on an android smartphone. In *the Workshop on Usable Security* (2012).

[25] LEIBA, B., AND FENTON, J. Domainkeys identified mail (dkim): Using digital signatures for domain verification. In *Proceedings of the Fourth Conference on Email and Anti-Spam (CEAS)* (2007), Citeseer.

[26] MOORE, T., AND EDELMAN, B. Measuring the perpetrators and funders of typosquatting. *Financial Cryptography and Data Security* (2010), 175–191.

[27] MOORE, T., LEONTIADIS, N., AND CHRISTIN, N. Fashion crimes: trending-term exploitation on the web. In *Proc. ACM CCS* (Chicago, IL, 2011), pp. 455–466.

[28] PILZ, H., AND SCHINDLER, S. Are free android virus scanners any good? AVTEST Report.

[29] SCHMIDT, A., BYE, R., SCHMIDT, H., CLAUSEN, J., KIRAZ, O., YUKSEL, K., CAMTEPE, S., AND ALBAYRAK, S. Static analysis of executables for collaborative malware detection on android. In *Proc. IEEE ICC* (2009), IEEE, pp. 1–5.

[30] SCHMIDT, A., CLAUSEN, J., CAMTEPE, A., AND ALBAYRAK, S. Detecting symbian os malware through static function call analysis. In *Proc. MALWARE* (2009), IEEE, pp. 15–22.

[31] SCHÖNEFELD, M. Reconstructing dalvik applications. *CANSECWEST 2009* (Mar. 2009).

[32] SCHWARTS, M. Google removes malware apps from android market, June 2011.

[33] THOUGHTCRIME LABS. Convergence: An agile, distributed and secure stratefy for replacing certificate authorities. `http://convergence.io`.

[34] VIDAS, T., VOTIPKA, D., AND CHRISTIN, N. All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX Workshop on Offensive technologies* (2011), USENIX Association.

[35] WENDLANDT, D., ANDERSEN, D., AND PERRIG, A. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference* (2008), pp. 321–334.

[36] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Proc. IEEE Symp. on Security and Privacy* (2012).

[37] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS* (2012).