Reducing the Window of Opportunity for Android Malware
Gotta catch 'em all

Axelle Apvrille[1] and Tim Strazzere[2]

[1]Fortinet
[2]Lookout Mobile Security

About Authors

Axelle Apvrille is a Senior Analyst and Research at Fortinet, FortiGuard Labs. She tracks down all kinds of malware for mobile phones, and tries to reverse them. Her work has already been presented at several conferences such as EICAR, VB, BlackHat, ShmooCon etc. Before reverse engineering with Fortinet, Axelle worked for over 12 years on the research and design of security systems. She also enjoyed teaching computer security in several French engineering schools, published in academic journals or conferences, and filed over 10 patents. Last but not least, Axelle is a proud user of OpenSolaris.

Contact details: EMEA AV Team, 120, rue Albert Caquot, 06410 Biot, France, e-mail: aapvrille@fortinet.com

Tim Strazzere is a Senior Security Engineer at Lookout Mobile Security. Along with writing security software, he specializes in reverse engineering and malware analysis. Some interesting past projects include having reversed the Android Market protocol, Dalvik decompilers/fuzzers and memory manipulation on mobile devices.

Contact Details: 1 Front Street, Suite 2700, San Francisco, CA 94111, USA, email: strazz@gmail.com

Keywords

Malware, Android, Static analysis, Crawler, Marketplaces

Abstract

Spotting malicious samples in the wild has always been difficult, and Android malware is no exception. Actually, the fact Android applications are (usually) not directly accessible from market places hardens the task even more. For instance, Google enforces its own communication protocol to browse and download applications from its market. Thus, an efficient market crawler must reverse and implement this protocol, issue appropriate search requests and take necessary steps so as not to be banned.

From end-users' side, having difficulties spotting malicious mobile applications results in most Android malware remaining unnoticed up to 3 months before a security researcher finally stumbles on it. To reduce this window of opportunity, this paper presents a heuristics engine that statically pre-processes and prioritizes samples. The engine uses 39 different flags of different nature such as Java API calls, presence of embedded executables, code size, URLs... Each flag is assigned a different weight, based on statistics we computed from the techniques mobile malware authors most commonly use in their code. The engine outputs a risk score which highlights samples which are the most likely to be malicious.

The engine has been tested over a set of clean applications and malicious ones. The results show a strong difference in the average risk score for both sets and in its distribution, proving its use to spot malware.

1 Introduction

The mobile Android operating system is rising in all areas. It is rising in *market shares* - 52.5% of sales in Q3 2011 according to Gartner, in front of Symbian, iOS and Windows Mobile - and in *number of applications*: (Wikipedia, 2011) now reports 370,000 applications in Google's Android market. Unfortunately, it is also rising *in malware*. (Schmidt et al., 2009) had already predicted this in 2009, and most anti-virus vendors have acknowledged the rise in various blog posts (Pontevès, 2011), technical reports (McAffee Labs, 2011) or conferences (Armstrong & Maslennikov, 2011). In some case, malicious samples have been downloaded massively: for example, Android/DrdDream was downloaded over 200,000 times (Lookout Mobile Security, 2011). In November 2011, there are approximately 2,000 Android malicious unique samples that belong to 80 different families[1]. If, like (Anderson & Wolff, 2010) claimed, the web is dead in favour to users downloading *applications* rather than directly browsing the web, it is likely this trend will only sharpen in the next few years.

Usually, anti-virus vendors find new malware from one of the following sources:

- **Users**: Victims, users, customers, partners or security researchers regularly submit suspicious files they encounter to AV vendors. Those samples are analyzed and if they are found to be malicious and undetected, a new signature is added to the AV engine. 100 of PC malware are

[1]Those statistics are taken from Fortinet internal databases. It should however be noted that figures vary among anti-virus vendors depending on the classification of samples.

submitted daily for analysis through that service, but unfortunately there are only few mobile submissions. One of the possible reasons to this is that malicious files are often hidden on mobile phones. For instance, an end-user typically does not see the Android package (APK) he/she installs, and therefore, doesn't know what to submit for scanning. Other reasons are that end-users are not accustomed to using file browsers on their phones, or the lack of education on mobile malware.

- **AV malware exchange**: AV vendors do daily automated exchanges with each other. This is a large resource for malicious samples and it ensures protection against virulent samples from all participating vendors. However, of course, the exchange only occurs once a vendor has spotted the malware. So, this source does not help find unknown malware in the wild.
- **In the wild**: Security researchers seek the web, forums or social networks for malicious samples, but in the middle of hundreds of thousands of genuine mobile applications, spotting malicious ones is (fortunately) like finding a needle in a haystack. Some automation is needed. Additionally, the advent of application stores harden the process of downloading applications on a desktop for analysis, because they lock up access to a given user account and his/her mobile devices.

It is that third and last category this paper is focusing on, and more specifically on malware for *Android* platforms. We present a heuristics engine, that helps sort out collections of applications (malicious or not). Precisely, we are interested *in mobile malware that are not detected by any vendor yet*. Those samples may belong to already known families, but still be undetected because current signatures (anti-virus detection patterns) are not good enough, or, in other cases, they might consist in entirely new and unknown families. The latter is certainly more attractive to researchers, but, yet, both categories need to be detected to protect the end-user, and consequently, *both* categories are taken into account in this paper.

As we show in the next section, only little research has been conducted on finding mobile malware in the wild. We discuss the limitations of previous work and highlight what we are able to tackle. We also provide a rationale for crawling Android market places for unknown samples which are in the wild (section 3). Basically, our contribution consists in explaining the issues with scanning Google's Android Market (section 4) and how we manage to put a magnifying glass on the haystack and find the needles (section 5). The results of our system is discussed in section 6. This work can certainly be improved - a few options are detailed in the results section - but it opens research on the subject.

2 State of the Art

Spotting malware in the wild, as early as possible before they have had time to cause harm (or too much) is an idea which has already been developed multiple times *for PC* malware, particularly with the use of honeypots.

(Wang et al., 2006) presented an automated web patrol system which browses the web using potentially vulnerable browsers piloted by monkey programs, in hope of identifying exploits in the wild. Then, (Ikinci, Holz, & Freiling, 2008) built a malicious web site crawler and analyzer, named Monkey-Spider. From various sources such as keywords or email spams, their system generates a list of URLs to crawl. They download everything they find on the website and make sure to follow links, even those found in Javascript or PDF documents. Then, they search the dump for malware using common anti-virus products as a first stage, and sandboxes in a second stage. Alternatively, (Ma, Saul, Savage, & Voelker, 2009) proposed to identify malicious web sites using an automated URL classification method. The idea consists in only using the URL (its look, length of hostname, number of dots...) and its host (IP address, whois properties) to determine if it is malicious or not. The content, or context, is not downloaded, and thus results in a lightweight detection system.

However, all those systems show severe limitations when it comes to finding *mobile* malware, because of the specificities of mobile networks:

- Most application stores do not provide direct access to the applications they host. Consequently, a `wget` on the stores only downloads HTML pages - which are irrelevant for mobile malware - and not the mobile application itself. In particular, the Android Market implements its specific protocol to download applications (Strazzere, 2009; Pišljar, 2010). The crawler in (Ikinci et al., 2008) would need to support such protocols to provide any result.
- URLs to be displayed on mobile phones generally have a different format than on PCs. They are typically shorter or shortened (using a URL shortening service), prefixed with ”m.” or using domain name mobi etc. Thus, (Ma et al., 2009) 's work and Monkey-Spider's seeder (Ikinci et al., 2008) would need to be adapted.
- Up to now, there are only few exploits for mobile phones and, actually, no browser vulnerability at all has ever been used by an Android malware. This is because they mostly manage to do their malicious tasks by clever calls to public APIs or social engineering (Apvrille & Zhang, 2010). So, solutions like (Wang et al., 2006) which only look for browser exploits would be bound to miss many malicious samples.
- Mobile phones are less easy to manipulate than a desktop (limited resources etc). Thus, solutions such as (Wang et al., 2006) which browse the web from the *client* itself - the mobile phone in our case - are not very practical for mobile phones. Moreover, monkey programs to automate the browsing on mobile phones (like the monkeyrunner for Android) are not fully mature yet.
- In the case of mobile platforms, the maliciousness of mobile malware cannot be limited to downloading applications, accessing given URLs or connecting to remote servers. The very fact mobile phones operate on a different network (GSM) opens up to other targets such as calling premium phone numbers, sending SMS messages or accessing WAP gateways. (Ikinci et al., 2008)'s sandbox would need to detect malicious behaviours for these. Actually, writing a sandbox for a mobile environment is a project in its own. Currently, for Android devices, we are only aware of DroidBox (`https://code.google.com/p/droidbox`), but it is in alpha stage. It requires manual source code modifications and when we tested it over Android/Geinimi.A!tr it was slow and unable to detect the malware's malicious activities.

So, it seems that research for PCs cannot directly be applied to mobile platforms and requires modifications in depth. Hence, we search for work specifically meant for mobile phones but there is only little prior art in this domain. A few months ago, a Google Summer of Code project consisting of an Android Market crawler was proposed (Logan, Desnos, & Smith, 2011) but later cancelled. Another project, named DroidRanger (Zhou, Wang, Zhou, & Jiang, 2012), seems promising, having found several malicious applications in the Android market and alternative markets, but it isn't published yet.

One of the closest match to mobile malware scanners is (Bläsing, Schmidt, Batyuk, Camtepe, & Albayrak, 2010). In that paper, the authors propose an Android Application Sandbox (AAS) to detect suspicious software. For each sample, first, they perform static analysis: they decompile the code and match 5 different types of patterns: using JNI, reflection, spawning children, using services and IPC, requested permissions. Then, they complement their approach with dynamic analysis consisting of a system call logger at kernel level.

Note (Bläsing et al., 2010) only handles the malware analysis part, not the crawling part of market places: AAS is meant to be installed within the Android Market for instance. In our paper, sections 3 and 4 detail the crawling of market places from independent, external hosts. As for static analysis, the work we present in this paper covers much more malicious patterns (see Section 5) and thus makes detection of suspicious malware more accurate.

(Teufl et al., 2011) proposed an Android Market metadata extractor and analyzer. They downloaded the metadata (i.e information available on the application's page: permission, download counts, ratings, price, description...) for 130,211 applications and then analyzed the metadata for various correlations. Our paper goes a few steps further, as first it downloads the applications from the market - which is more complicated than getting the metadata - and second, the analysis is performed on far more parameters, 39 properties currently to be precise.

The process we propose in this paper (see Figure 3) consists in:

1. Crawling mobile market places for Android applications. Section 4 presents a few hints at how to crawl the Android Market,

2. Statically analyzing samples as we receive them in a heuristics engine(section 5). This analysis computes a risk score based on the features it sees in the application's decompiled code. Static analysis has the advantage of being virtually undetectable, as obviously the malware cannot modify its behaviour during analysis. The only method to bypass static analysis is code obfuscation and we detect some attempts which use encryption. Static analysis is also relatively fast, hence with less risks of creating a bottleneck as mentioned by (Ikinci et al., 2008).

3. If this score is greater than a given threshold, the sample is labeled as suspicious and undergoes further treatment. This treatment is out of the scope of the paper, but for instance, it can be manual analysis, AV scanning or dynamic analysis. The goal of our work here is to prioritize samples that need closer investigation. Given the amount of samples to process, it is important

to start with those which are the most likely to be malicious. Others can be scanned later - if there is time.

3 Rationale

In this section, we explore the reasons and difficulties for finding Android malware in the wild.

Android applications can be downloaded from several sources. The best known application store is Google's Android Market, with 370,000 applications and 7 billion downloads in November 2011 (Wikipedia, 2011), but numerous other sources exist, ranging from perfectly legitimate and manually reviewed stores like Amazon's (Lookout Mobile Security, 2011) to more unofficial markets (e.g blapkmarket).

The total count of Android applications is unknown as the list of market places themselves evolve regularly and because several of them do not provide an accurate headcount of applications they store. Moreover, some applications are listed in multiple places. We know for sure there are 370,000 applications in Google's market (Wikipedia, 2011), and we counted the number of applications in 10 other market places: 199,617 applications. There are still 37 other market places we did not count applications for, not to mention forums and file sharing websites. So, even if our figures include a few duplicates, there are still probably over 600,000 Android applications in the wild.

Given those facts, scanning Android applications is a considerable job, and it is not surprising that many malware stay in the wild undetected before an anti-virus vendor or a security researcher finally spots them and alerts the community. For Android malware, according to our research, the gap between release in the wild of a new *mobile malware* and its detection by one anti-virus vendor is bigger than that: it is approximately of *80 days*! We explain below how Figure 1 was computed.

The date of first detection by an anti-virus vendor is precisely known, and, generally, it does not vary much from one vendor to another. The difficulty resides in finding the day the malware was first released in the wild. Initially, we considered using the begin date of the certificate used to sign the malware, but developers typically re-use their certificates or use public certificates so this date would be earlier than the actual release of the malware. Instead, we chose to use the *timestamp of the package's ZIP*.

We are aware this date is not fully reliable, and only consider it as an approximation of the malware's release date. Indeed, the package's timestamp is not cryptographically signed, so it can obviously be tampered. Also, it is possible the malware author zips the package days before he/she actually releases it etc.

As an additional validation of the date, we only considered cases where:

$$\text{certificate begin date} \leq \text{package's zip date} \leq \text{first detection date}$$

Out of 90 malicious samples, only dates for 4 samples were evicted.

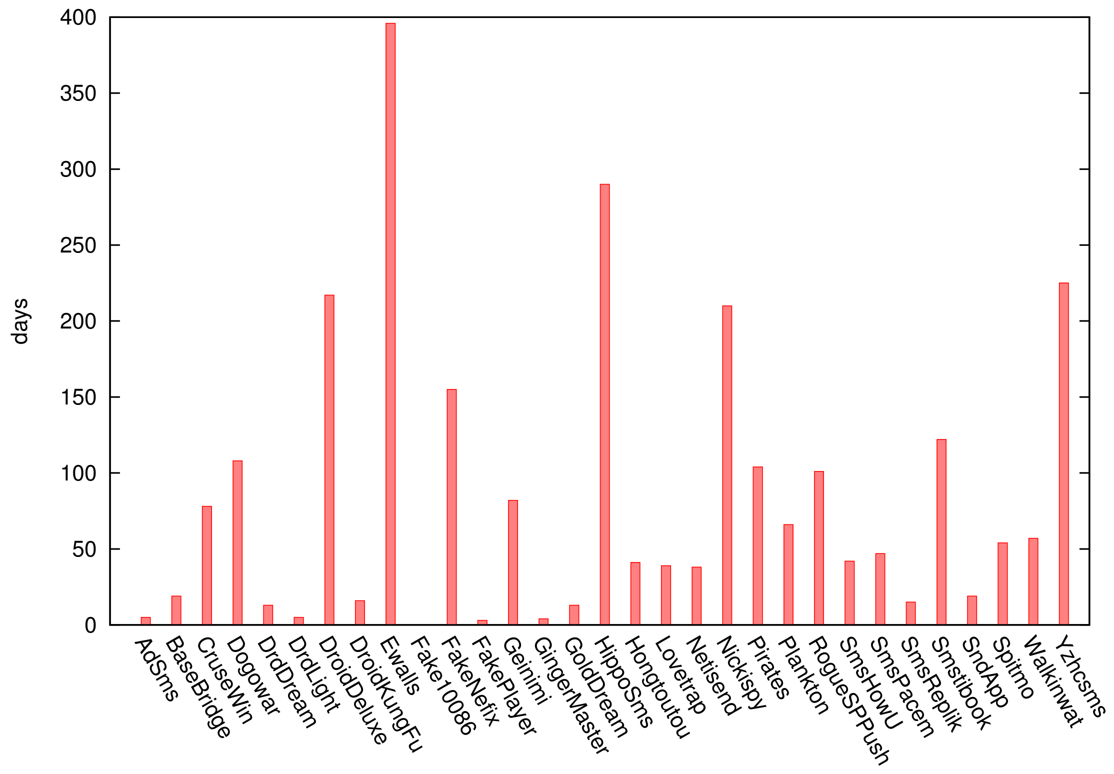


Figure 1: Number of days between first release in the wild of a malware sample and first detection by any vendor

The amount of Android applications to scan combined to the detection delay suggests we are currently unaware of several Android malware in the wild. This is only another incentive for our work.

4 Hints to Crawl the Android Market

Google's Android Market is only accessible via a proprietary protocol implemented on the device. As Google hasn't released any official API for it, reverse engineering the protocol and mimicking a device is the only way to interact with the market. To make things even harder, Google has been revving their market application nearly as often as they release new operating systems. Though since the first market release, the protocol has remained rather constant, keeping the protocol backwards compatible and not breaking older clients.

The market currently uses protocol buffers (Protb, n.d.) for communication both to the client and the server, which are then encoded with a Base64 Websafe character set. This is where most of the complexities of crawling the market occur. In order to properly crawl the market, the protocol must be kept up to date and in sync with what the server expects. The basic request context contains data specific to the device the request is from, these include the software version, product name, SDK version, user language, user country, operator name and a client ID. It also contains the authentication token with is tied to the users Google account. Lastly, the request context will also hold the Android ID which is linked to both the user's account and the device. For a proper request to be created, all these values must match up - or the resulting response can vary drastically. Crawling will only get harder since application developers can restrict who is allowed to see their application by any combination of these value.

Additionally, in order to keep a full listing of applications available in the Android Market, a crawler must perform odd sets of searches. Since the protocol is based off mimicking a device, the same limitations that are imposed on a device are imposed on a crawler. This means searches can only be conducted with 10 results being returned at a time, with a maximum of 800 results for any given search or category.

Within each category there are different views, such as 'Top Free', 'Top Paid', 'Just In Free' and 'Just In Paid'. This means that for one valid request context, a US based user on Verizon running SDK 10 on a Nexus S for example, can see 40 different categories with 4 different views giving a possibility of 128,000 different applications. Though this does not necessarily mean that they are all unique packages. To get those, it would require a minimum of 12,800 search requests (10 results for each search), followed by 128,000 requests to get each specific applications metadata. Then to download the applications, it would require another 128,000 requests. These are only the applications available for browsing by a phone, there are still countless other applications, with more being added almost every minute.

The last thing to complicate matters a bit more is *banning* from the market. On top of keeping track of request contexts, to maintain a good crawler a wide array of accounts must be created and maintained. If too many requests happen to fast, there are many different types of bans that may come into effect. These can range from an IP address ban, to an account being blacklisted (both from searching and downloading) to an Android ID and device ban. Monitoring the health of the crawling accounts is another thing that must be taken into consideration for proper crawling.

When creating a fully functional crawler for the Android Market, one should take into consideration again, that you are mimicking real devices. Unlike crawling a normal web page, this means you need to maintain multiple accounts, multiple device contexts and possibly multiple IP addresses. We found that using a combination of these along with exponential back-off rate limiting helps ensure we don't become banned. Along with this we enlist many health checks to ensure accounts do not appear to be flagged or be returning back bad data. Trying to rationalize your crawlers traffic is easier when you think of it in the context of, how much traffic could one device possibly generate and how fast can it do this? Asking ourselves these questions often, provides a good gut check on how design the rate limiting properly to not get banned.

After maintaining a crawler for some time, the value is easily seen. If we take all the metadata and binaries, we can perform countless types of queries which are not possible through the normal market protocol. An example of these are looking for anything named "Angry Birds" which isn't developed by ”Rovio”[2], or have different permissions. Another example is searching all descriptions and package names for similar characteristics, such a ”DroidDream”, which was one way extra accounts were found that where being used by the DroidDream crew (Lookout Mobile Security, 2011).

5 Heuristics Engine

As various market places are being crawled (see Section 4), Android applications are massively being downloaded. All those applications are not malicious - fortunately for end users - so we need to quickly sort out from the mass those which are the most suspicious. This work is performed by a heuristics engine. This tool can be seen as a quick pre-analyzer that rates applications according to their presumed suspiciousness. Applications which receive the highest score, thus which are the most likely to be malicious, then undergo further analysis, outside the scope of this paper.

For each Android application, the engine's algorithm is fairly simple and illustrated at Figure 3. First, the sample is uncompressed. This is usually nothing more than unzipping, as Android package's format (APK) is a zip file. We also handle cases where the sample is additionally zipped or RARed. Then, the sample's classes.dex, which concentrates the application's Dalvik code, is decompiled using baksmali (Smal, n.d.) to produce a more or less human readable format named Smali. The package's manifest, initially in binary format, is also decoded to plain XML format. Finally, all those elements - smali code, XML manifest, package's resources and assets - are analyzed. The analysis consists in searching for particular risky properties or patterns, and then accordingly incrementing the risk score.

The difficulty and success of the engine actually lies in writing clever property detectors so as to generate few false positives and false negatives. The task proves out to be more difficult than expected, because genuine applications sometimes use unexpected functionalities. For example, as some of the most advanced Android malware typically use encryption algorithms to obfuscate their goal (e.g Android/DroidKungFu), we considered raising an alarm whenever an application uses encryption. A naive detector consists in detecting calls to the Java `KeySpec` API, but this isn't any good, because it also detects many advertisement kits that genuine applications use. Indeed, many advertisement kits encrypt communication with their servers or implement tricks to be ensure the ads are not removed. After analyzing a series of clean applications, we implemented the following pattern:

calls to KeySpec *or* SecretKey *or* Cipher APIs
but *not* from com.google.ads, *nor* mobileads.google.com, *nor* com.android.vending.licensing *nor* openfeint *nor* oauth.signpost.signature *nor* org.apache.james.mime4j *nor* com.google.android.youtube.core

In most property detectors, we had to filter out cases where the functionality was being used within advertisement kits, billing APIs, legitimate authentication, youtube, social gaming networks such as Openfeint etc.

The 39 property detectors we implemented so far fall in one of the 7 categories below:

[2]This was the case for samples of Android/RuFraud recently.

- **Permissions required in the Android manifest**. This is one of the first properties that comes to mind to check what a sample does. We keep a particular eye on Internet, SMS, MMS, calls, geographic location, contacts and package installation permissions. If those permissions are requested, we increment the risk score. However, permissions alone are insufficient to spot malware.

- **API call detectors**. This is the most important category of property detectors. It consists in detecting the use of particular Java methods, classes or constants, spotting them in the decompiled smali code. The API elements it detects are of very different nature. Some of those concern actions or features of the phone: send/receive SMS, call phone numbers, geographic location, get telephony information, listing or installing other packages on the phone. Others concern Java language tweeks: dynamic class loading, reflection or JNI. Finally, a few calls concern the underlying Linux operating system such as the creation of new processes. See Table 5 for details.

- **Command detectors**. The engine also detects use of specific Unix or shell commands. This is similar to detecting API calls, except commands can be located within scripts of raw resources or assets, so those directories need to be scanned too. Currently, we only increment the risk score when the command `pm install` (installation of Android packages) is detected.

- **Presence of executables or zip files in resources or assets**. This is used to detect malware which run exploits on the phone. We said previously exploits weren't used that often yet, but, when an exploit is used the sample is generally malicious.

- **Geographic detectors**. Currently, 40% of mobile malware families seem to originate from Russia, Ukraine, Belorus, Latvia and Lithuania, and 35% from China[3]. The engine consequently slighlty raises the risk score for samples which appear to come from those countries. In particular, it raises the risk score if the signing certificate's country is one of those, or if the malware mentions a few keywords such as `10086` (China Mobile customer service portal), `cmnet` or `cmwap` (China Mobile gateways).

- **URL detectors**. On one side, access to Internet is important to malware authors to report back information, update settings or get further commands, so it seems important to increment the risk score when the sample accesses Internet. On the other side, there are so many genuine reasons to access Internet that we have to make sure not to raise the alarm unnecessarily. We chose to raise the risk score only once if a URL is encountered (i.e if the engine detects 4 URLs, the risk score is only incremented once), and also to skip the extremely frequent URLs such as the Android Market's URL, Google services (mail, documents, calendar...), XML schemas and advertisement companies.
  Note there is a particular case for URLs: if the URL downloads an APK, we raise the risk score more importantly as this can mean the sample is trying to update or install another application.

[3]Statistics from Fortinet's internal databases.

- **Size of code**. An analysis of the size of malicious APKs compared to benign APKs shows that the average size is comparable, but that the distribution is different. In particular, there are more very small malware, with sizes less than 70,000 bytes: 30% of malicious files against 21% of clean files (see Table 5). So, we raise the risk score for samples below 70,000 bytes.
- **Combinations**. We increment the risk score if some specific conditions are met, such as if the sample gets the geographic location and accesses Internet. Indeed, in that case, the sample has the capability to report the end-user's geographic location, which results in a privacy threat.

API detected	**Threat**
`sendTextMessage()`, `sendMultipartTextMessage()`	Sending SMS to short numbers
Constants `FEATURE_ENABLE_MMS`, `EXTRA_STREAM`, `content://mms`	Sending MMS without consent
Constants: `EXTRA_EMAIL`, `EXTRA_SUBJECT`	E.g. Communicating with remote server via emails
`SmsMessage.createFromPdu()`, `getOriginatingAddress()`, `SMS_RECEIVED`, `content://sms`, `sms_body` ...	Forwarding SMS to a spy number, deleting SMS etc.
`Intent.ACTION_CALL`	Calling a premium phone number
Constant `POST` or class `HttpPost`	POSTing private information via HTTP
KeySpec, SecretKey, Cipher classes	Using encryption to obfuscate part of the code
Methods of the TelephonyManager class: `getDeviceId()`, `getSubscriberId()`, `getNetworkOperator()`, `getLine1Number()`, `getSimOperator()`, `getSimSerialNumber()`, `getSimCountryIso()`	IMEI, IMSI are personal information
Methods of PackageInfo class: `signatures()`, `getInstalledPackages()`	Checking malware's integrity, deleting given packages, posting list of packages to remote server etc.
DexClassLoader class	Loading a class in a stealthy manner
Static methods `Class.forName()`, `Method.invoke()`	Reflection. Loading class in a stealthy manner.
Method `Runtime.exec()` or using android.os.Exec class or `createSubprocess()`	E.g. Executing an exploit
`JNIenv`, `jclass`, `jmethodID`, `jfieldID`, `FindClass`	JNI: executing native code

Table 1: Implemented Java API call detectors which are particularly monitored to detect Android malware

The risk score has no particular unit. It is incremented based on the different likelihoods of a given situations for Android malware or clean applications. To compute the weights (risk score increments), we analyzed 97 malicious samples (taken from 41 different families) and 217 clean samples and tested whether each property was found or not. For each situation, we therefore compute a percentage of likelihood.

Then, basically, we are interested in big differences between percentages for Android malware and percentages for clean samples.

If the difference of percentage points is ≥ 50, we assign a weight of 5.
If the difference ≥ 40 and $< 50, weight = 4$.
If the difference ≥ 30 and $< 40, weight = 3$.
If the difference ≥ 20 and $< 30, weight = 2$.
Finally, if the difference is less than 20%, we use the smaller weight, 1.

So, for example, Table 5 shows that 59% of Android malware send SMS messages whereas only 6% of clean samples do. The difference of percentage points is 53, so we increment the risk score by 5 if the situation is encountered.

Situation	Likelihood % for Android malware	Likelihood % for clean files	Weight
Send SMS	59	6	5
Receive SMS	60	10	5
Performs HTTP POST	68	25	4
Combination of SMS and access to Internet	46	6	4
Gets IMEI	63	20	4
Uses HTTP or views a URL	85	50	3
Gets IMSI	36	1	3
Code contains a URL	65	41	2
Uses encryption	34	10	2
Gets phone line number	27	6	2
Gets information concerning the SIM card	32	5	2
Specifically targets China	26	0	2
Lists installed packages	33	5	2
Other situations			1
Size $< 70,000$ bytes	30	21	1
..			

Table 2: Comparing likelihood percentages of a subset of situations computed for 97 malware and 217 clean files

6 Results

This section details the results of the tools presented in this paper.

6.1 Market Scanner Results

The implementation of the market scanner is not public and consequently could not be disclosed in this paper. However, it is successful in entirely scanning Google's Android Market.

Besides its initial goal - retrieving samples from the market place - the market scanner proved out to be quite useful in another area: tracking malware authors.

Indeed, combining heuristics on the applications, common package naming schemes and tracking metadata, developers can be tracked , people pirating their applications can be tracked, and so can malware authors.

Actually, this method was used to track Legacy (aka Android/DroidKungFu) in the Android Market. Originally, we found Legacy samples in third party Chinese markets, which appears to be the place where the authors first released it. Then, since we had historical data in that market (we keep a chronology of metadata using when searching that market), we saw the authors build a user base with an application - then push a malicious update to the market. When we searched for the non-malicious update, we could see that it was being seeded into the Android Market.

6.2 Heuristics Engine Results

On its side, a heuristics engine prototype was implemented as a basic Perl script. Overall, we processed more than 3,000 samples with it. In particular, it was at the origin of the discovery of Riskware/Sheriff and Riskware/ESSecurity.

To test the engine, we had it analyze a set of 947 clean samples[4], and a set of 107 malicious samples[5]. Note we did *not* use the same sets as the ones used to compute the risk score increments at Table 5 so as not to influence results.

The results are displayed at Figure 2. They show a clear difference in the distribution of risk score for both sets: clean samples tend to have most of their risk scores below 15, while malicious samples are the most frequent above 45. For those data sets, there was no risk score above 40 for clean samples and above 55 for malicious ones.

The computed average risk score for clean samples is 8, while it is of 44 for malicious samples.

6.3 Limitations

Though its results are quite promising up to now, there are a few limitations and improvements to plan for the engine.

First, it is important to understand a heuristic engine is by design not perfect: it generates false positives and false negatives.

The case occurred for an application named Prepay Widget, an Android widget to display your

[4]Those clean samples were taken from the web, in particular from the F-Droid open source market place, and were manually double checked to ensure they were genuine.

[5]Malicious samples were downloaded from Mila Parkour's repository on `http://contagiominidump.blogspot.com` and from a malware exchange with NetQin.

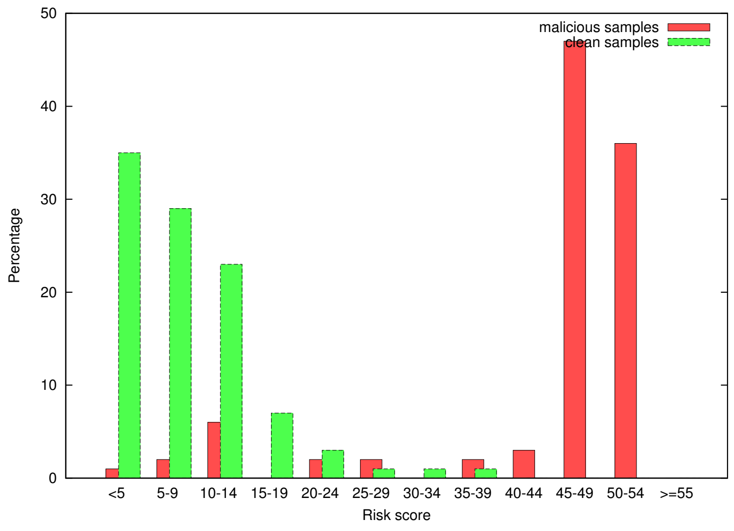


Figure 2: Distribution of risk scores for a set of malicious samples and a set of clean samples

plan's balance, free minutes, traffic etc. The application was sending USSD commands[6] to get plan's information and thus triggering the call property detector. It was also reading incoming SMS as some operators reply to USSD via SMS, and thus caught by the SMS receiver detector. It was signed by a Russian certificate, so caught by the geographical detector. It was testing whether the phone was rooted (to put the dialer in background) and thus triggering the Runtime.exec() detector etc. All those alarms could be explained after a manual check, but they resulted in a high risk score (36) for a non-malicious application. However, with all the borderline techniques it used, we believe the heuristic engine was however right to raise the alarm as this *could* have been malicious.

Reciprocally, Figure 2 show a few malicious applications have risk score below 15, and this will be the case for a few clever applications like (Cannon, 2011). This Proof of Concept provides a remote shell via the installation of an application that does not request any permission. The trick consists in having the application launch a web browser, registering a custom URI and having the remote server send encoded commands through the connection to the web browser. Then, the commands are decoded and executed. The code for the PoC is not published so we could not test the heuristic engine on it, however, from discussion with its author, it appears it would have at least triggered the API call detector to `Runtime.exec()`.

Clearly, the goal of the heuristics engine is not to detect *everything*, but to help detect *most* malicious malware. The results we presented at Figure 2 prove this is statistically true.

Technically speaking, the engine could be enhanced in several ways:

[6]USSD is a GSM protocol to communicate with the operator.

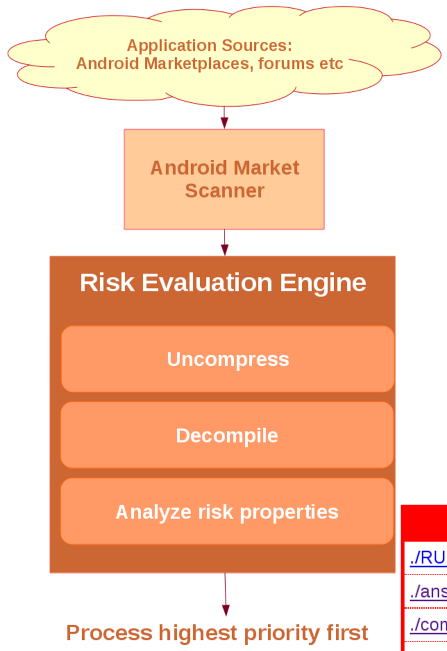
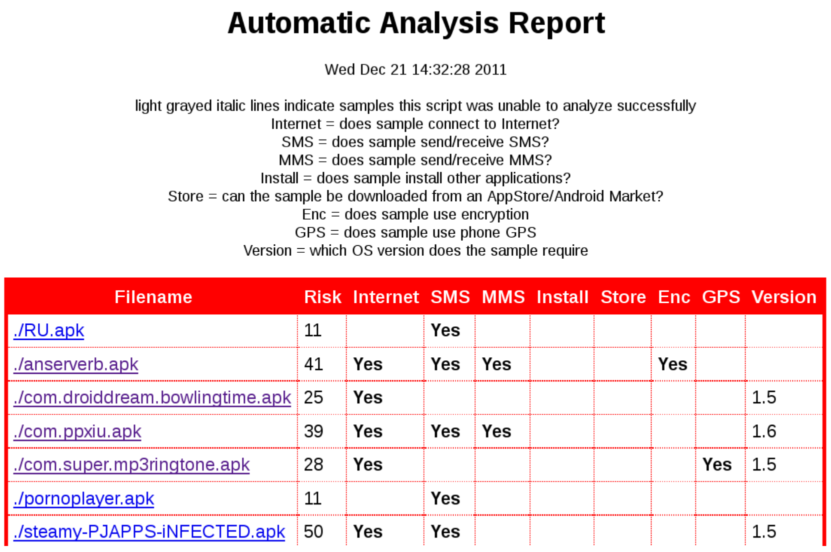


Figure 3: Process to find new mobile malware. The risk evaluation engine is our heuristics engine



Automatic Analysis Report

Wed Dec 21 14:32:28 2011

light grayed italic lines indicate samples this script was unable to analyze successfully
Internet = does sample connect to Internet?
SMS = does sample send/receive SMS?
MMS = does sample send/receive MMS?
Install = does sample install other applications?
Store = can the sample be downloaded from an AppStore/Android Market?
Enc = does sample use encryption
GPS = does sample use phone GPS
Version = which OS version does the sample require

Filename	Risk	Internet	SMS	MMS	Install	Store	Enc	GPS	Version
./RU.apk	11		Yes						
./anserverb.apk	41	Yes	Yes	Yes			Yes		
./com.droiddream.bowlingtime.apk	25	Yes							1.5
./com.ppxiu.apk	39	Yes	Yes	Yes					1.6
./com.super.mp3ringtone.apk	28	Yes						Yes	1.5
./pornoplayer.apk	11		Yes						
./steamy-PJAPPS-INFECTED.apk	50	Yes	Yes						1.5

Figure 4: Sample output report of the heuristic engine (extract)

- Performance. For example, currently each property detector results in a search (grep) on a given directory. This means parsing the analysis directory several times (nearly one for each property - depending on properties). Rather, a single common search could be done, and the results scanned for each property.

- Improving or adding detectors. We plan to improve the executable detector which currently merely detects the presence of an executable or zip file in assets or raw resources. It is therefore triggered by the presence of genuine libraries such as libGoogleAnalytics.jar. The detector could filter out such libraries or scan for given keywords in the executables.
  We also plan to add a public certificate detector to spot applications signed using a debug, test or development certificate. Indeed, the following public certificate has been used several times by malware authors and might be a potential indicator:

  ```
  EMAILADDRESS=android@android.com  CN=Android  OU=Android
  O=Android  L=Mountain View  ST=California  C=US
  Serial number: 936eacbe07f201df
  ```

  We also consider improving the URL detector, as nearly all URL trigger the alarm (see section 5). It would be interesting to update (Ma et al., 2009) for mobile URLs and increment the risk score differently depending on URLs which are found.

- Computing weights. Data mining approaches would be an improvement to our heuristics engine. In our prototype, we arbitrarily decided to assign weights from 1 to 5 depending on difference of percentage points, but a real engine should certainly be tuned from results of data mining research.
- Recursive applications. The heuristics engine detects that a given sample contains another APK in its resources or assets, as indeed, this is an additional risk. However, the engine does not then recursively analyze that APK.
- Tests. We plan to test the engine against larger sets of samples, to fine tune the detectors and risk increments. Testing the engine against clean file sets is particularly time consuming, because each application has to be manually inspected to make sure it is not malicious or infected.

7 Conclusion

In this paper, we have presented ways to help spot Android malware which are in the wild. We explained the catches in implementing a market scanner, and implemented one that entirely scans Google's Android Market. To do so, it uses protocol buffers and performs several searches using different combination of parameters such as the country, language, application category etc. It makes sure to crawl all existing applications, and not only those visible by a given device. It also deals with the risk of getting banned because of too many downloads. As a side-effect, by keeping a history of scanned metadata of markets, the scanner has also successfully been used to track malware authors such as the creators of Legacy / DroidKungFu.

While markets are being crawled and samples stack on a disk, we use a heuristics engine to quickly pre-process samples. The goal is to give a rough idea of which samples are the most likely to be malicious and prioritize them. This heuristics engine is detailed in the paper. It is a static analyzer that checks for 39 different properties such as requested permissions, calls to particular Java methods or classes, constants, assumed geographic data, code size etc. Each property corresponds to a given risk score increment. The value of the increment in itself has been computed out of training data sets. An engine prototype is currently implemented as a Perl script and has been tested against 947 clean samples and 107 malicious ones, for which it provides clearly different risk scores.

The concepts and implementation of the heuristics engine are strongly based on Android malware statistics. The paper therefore also presents a few interesting results such as the fact Android malware sit in the wild 3 months in average before anybody spots them, or that 63% of Android malicious samples retrieve the phone's IMEI and 59% send SMS messages.

The question of scanning mobile markets is relatively new and there hasn't been much research on it yet. So, the tools presented in this paper are quite relative newcomers. They are consequently expected to much improvements in the future, both in performance, tuning of scores and selectivity of malware.

Appendix: Android market places

Amazon AppStore	4,000
Android Blip	> 70,000
Android Pazari	2,052
Appoke	3,300
AppsLib	38,771
F-Droid	502
GetJar	75,000
Hyper Market	792
Indiroid	>700
Soc.io	4,500

http://andappstore.com
http://andiim3.com
http://androides-os.com
http://androidis.ru
http://android-phones.ru/category/files/
http://www.anzhi.com
http://aptoide.com
http://apk.hiapk.com
http://apps.opera.com/
http://blapkmarket.com,
http://indiroid.com
http://mikandi.com
http://myandroid.su/index.php/catprog
http://onlyandroid.mobihand.com
http://open.app.qq.com
http://snappzmarket.com/
http://wandoujia.com/
http://www.androidpit.com
http://www.19sy.com
http://www.1mobile.com
http://www.92apk.com
http://www.androidonline.net
http://www.androidz.com.br
http://www.appchina.com
http://www.appitalism.com
http://www.aproov.com
http://www.downapk.com
http://www.eoemarket.com
http://www.handster.com
http://www.insydemarket.com
http://moyandroid.net
http://www.nduoa.com
http://www.openappmkt.com
http://www.pocketgear.com,
http://slideme.org
http://www.sjapk.com
http://www.starandroid.com
http://www.yingyonghui.com
http://www.zerosj.com/
http://yaam.mobi/

http://forum.xda-developers.com
http://4pda.ru/forum/index.php?showforum=281

References

Anderson, C., & Wolff, M. (2010, August). *The Web is Dead. Long Live the Internet.* (http://www.wired.com/magazine/2010/08/ff_webrip/all/1)

Apvrille, A., & Zhang, J. (2010, May). Four Malware and a Funeral. In *5th Conf. on Network Architectures and Information Systems Security (SAR-SSI).*

Armstrong, T., & Maslennikov, D. (2011). Android malware is on the rise. In *Virus bulletin conference.*

Bläsing, T., Schmidt, A.-D., Batyuk, L., Camtepe, S. A., & Albayrak, S. (2010). An Android Application Sandbox System for Suspicious Software Detection. In *5th international conference on malicious and unwanted software (MALWARE'2010).* Nancy, France, France.

Cannon, T. (2011, December). *No-permission Android App Gives Remote Shell.* (`http://viaforensics.com/security/nopermission-android-app-remote-shell.html`)

Ikinci, A., Holz, T., & Freiling, F. C. (2008). Monkey-spider: Detecting malicious websites with low-interaction honeyclients. In *Sicherheit* (p. 407-421).

Logan, R., Desnos, A., & Smith, R. (2011). *The Android Marketplace Crawler.* (`http://www.honeynet.org/gsoc/ideas`)

Lookout Mobile Security. (2011, August). *Lookout Mobile Threat Report.*

Ma, J., Saul, L. K., Savage, S., & Voelker, G. M. (2009). Beyond blacklists: learning to detect malicious web sites from suspicious urls. In *Proceedings of the 15th acm sigkdd international conference on knowledge discovery and data mining* (pp. 1245–1254). New York, NY, USA: ACM. Available from `http://doi.acm.org/10.1145/1557019.1557153`

McAffee Labs. (2011). *Mc Affee Threats Report: Third Quarter 2011.* (`http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2011.pdf`)

Pišljar, P. (2010, February). *Reversing Android Market Protocol.* (`http://peter.pisljar.si/`)

Pontevès, K. de. (2011, November). *Android Malware Surges in 2011.* (`https://blog.fortinet.com/android-malware-surges-in-2011`)

Protocol Buffers. (n.d.). (http://code.google.com/p/protobuf/)

Schmidt, A.-D., Schmidt, H.-G., Batyuk, L., Clausen, J. H., Camtepe, S. A., & Albayrak, S. (2009). Smartphone Malware Evolution Revisited: Android Next Target? In *4th international conference on malicious and unwanted software (malware)* (pp. 1–7). IEEE.

Smali. (n.d.). (`https://code.google.com/p/smali`)

Strazzere, T. (2009, September). *Downloading market applications without the Vending app.* (`http://strazzere.com/blog/?p=293`)

Teufl, P., Kraxberger, S., Orthacker, C., Lackner, G., Gissing, M., Marsalek, A., et al. (2011). Android Market Analysis with Activation Patterns. In *Proceedings of the International ICST Conference on Security and Privacy in Mobile Information and Communication (MobiSec).*

Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., et al. (2006). Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the network and distributed system security symposium, ndss 2006, san diego, california, usa.* The Internet Society.

Wikipedia. (2011, November). *Android Market.* (`https://en.wikipedia.org/wiki/Android_Market`)

Zhou, Y., Wang, Z., Zhou, W., & Jiang, X. (2012). Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th network and distributed system security symposium (ndss 2012), san diego, ca, february 2012.*