SherlockDroid, an Inspector for Android Marketplaces

Axelle Apvrille
FortiGuard Labs, Fortinet
120 rue Albert Caquot, 06410 Biot, France
aapvrille@fortinet.com

Ludovic Apvrille
Institut Mines-Telecom, Telecom ParisTech, LTCI CNRS
450 route des Chappes, 06410 Biot, France
ludovic.apvrille@telecom-paristech.fr

July 22, 2014

Abstract

With over 1,200,000 Android applications in Google Play alone, and dozens of different marketplaces, Android malware unfortunately have no difficulty to sneak in and silently spread. This puts a high pressure on antivirus teams. To try and spot new malware instances, we have built an infrastructure, named SherlockDroid, whose goal is to filter out the mass of applications and only keep those which are the most likely to be malicious for future inspection by Anti-virus teams.

SherlockDroid consists of marketplace crawlers, code-level property extractors and a classification tool named Alligator which decides whether the sample looks malicious or not, based on some prior learning.

During our tests, we have extracted properties and classified over 480k applications. Since the beginning of July 2014, SherlockDroid has crawled 88,369 applications with the detection of one new malware, Android/Odpa.A!tr.spy, and one new riskware. With previous findings, this increases SherlockDroid and Alligator's "Hall of Shame" to 7 malware and potentially unwanted applications.

1 Introduction

For anti-virus researchers, spotting new malicious samples among the plethora of legitimate applications is a difficult task: there are over 1,200,000 applications in Google Play according to Wikipedia and numerous other marketplaces.

To assist researchers, we have built an automated marketplace crawling and filtering framework, named *SherlockDroid*. It is meant to process a large quantity of applications, filter out applications which are either clean or known malware, and only keep a very small set of suspicious samples. This set is then manually inspected by anti-virus analysts or security researchers so as to find whether they do contain or not novel malicious patterns.

It is important to highlight that SherlockDroid's goal is not to find *all malware*, but some which are new / unknown to the community. Missing known malware is not our concern. Missing unknown malware is not desirable, but probably inevitable with today's knowledge in anti-viral techniques compared to the flow of incoming malware. Conversely, wasting analysis time on clean samples is a big issue. The little time anti-virus analysts have, we want them to spend it on malicious samples. So, SherlockDroid is designed and tuned to minimize False Positives (clean samples marked as suspicious) while False Negatives (malicious samples marked as clean) are less important.

At [AS12], we had presented a very preliminary prototype of such a framework. Then, at [AA13], we presented SherlockDroid's open-source learning and classification engine. After several improvements, tuning and live experiments, we now wish to present the entire SherlockDroid as we haven't done that before.

Project name	Number of applications
DroidScope [YY12]	2 families
pBMDS [XZSZ10]	3 malware
Andromaly [SKE$^+$12]	4 artificial malware
Crowdroid [BZNT11]	2 real malware, 3 artificial
TaintDroid [EGC$^+$10]	30 applications
MADAM [DMSS12]	56 applications
AAS [BSB$^+$10]	150 clean samples, 1 artificial malware
Permission-Based framework [AZ13]	500 applications
PUMA [SSL$^+$12]	1,811 clean applications and 249 malware
DroidRanger [ZWZJ12]	204,040 applications
PlayDrone [VGN14]	1,100,000 applications
SherlockDroid	118,369 applications N.B: During our tests, subtasks of SherlockDroid like DroidLysis property extraction and Alligator classification have independently been run on over 480,000 samples

Table 1: Comparing number of applications researchers used to evaluate their system

Our presentation at Hack.Lu will globally follow the layout of this paper: comparison with other research (section 2), description of SherlockDroid's architecture (section 3), implementation issues for crawlers, static analysis property extraction (sections 4 and 5), classification results and malware which were discovered (sections 6 and 7). In addition, we will include a short demo of the tool in action.

2 Related work

There are several ways to try and detect Android malware. The detection can be done on or in interaction with the mobile device (e.g [BZNT11]), from within the marketplace itself [BSB$^+$10], or on an external host like SherlockDroid.

At table 1, apart from DroidRanger [ZWZJ12] and PlayDrone [VGN14] we discuss later in this section, we show that those studies have not been conducted on large scales. Either, they have not been designed to crawl marketplaces, or their applicability in practice for the anti-virus industry is still to demonstrate.

DroidRanger has been quite successful at spotting malware. By design, it is however limited to detect variants of known malware families or unknown malware that dynamically load untrusted code, via the implementation of only two heuristics. It is blind to other kinds of malware. In comparison, SherlockDroid's force relies on a far larger set of heuristics, 289 currently, which opens up its mind to more different malware kinds.

PlayDrone's goal is quite different from ours. It was implemented to gather statistics on the population of Google Play applications. As a side note, we remark that, despite its claim, PlayDrone *is not the first scalable Google Play crawler*. In 2012, our research work [AS12] with Lookout already described in length their crawler and implementation hints. Our paper clearly states the *entire* Google marketplace was scanned at that time, even keeping track of meta-data and applications versions over time. As a matter of fact, this helped Lookout track malware authors of DroidKungFu [Str11]. This being said, PlayDrone is nonetheless interesting research work. The main differences with our crawler are:

- PlayDrone uses Amazon EC2 cloud services to leverage computation power. We re-used old unused desktops: this is less efficient, but obviously less expensive.
- So as not to get their IP addresses banned during Google accounts creation, PlayDrone proxied their request through third party service providers. In SherlockDroid, we initiate a different Tor connection for each request.
- PlayDrone had other users create Google accounts for them, via Amazon's Mechanical Turk[1]. We automated the

[1] Mechanical Turk is a web service where registered users get paid to carry out simple tasks.

creation of Android IDs using an open source toolkit [Akd].

Also, it does make sense to entirely crawl Google Play for statistical goals, but in our case, for malware detection, this is over-kill. For example, applications which do not connect to Internet, do not make calls and do not send SMS are very unlikely to be be malicious.

3 Architecture overview

3.1 Components of SherlockDroid

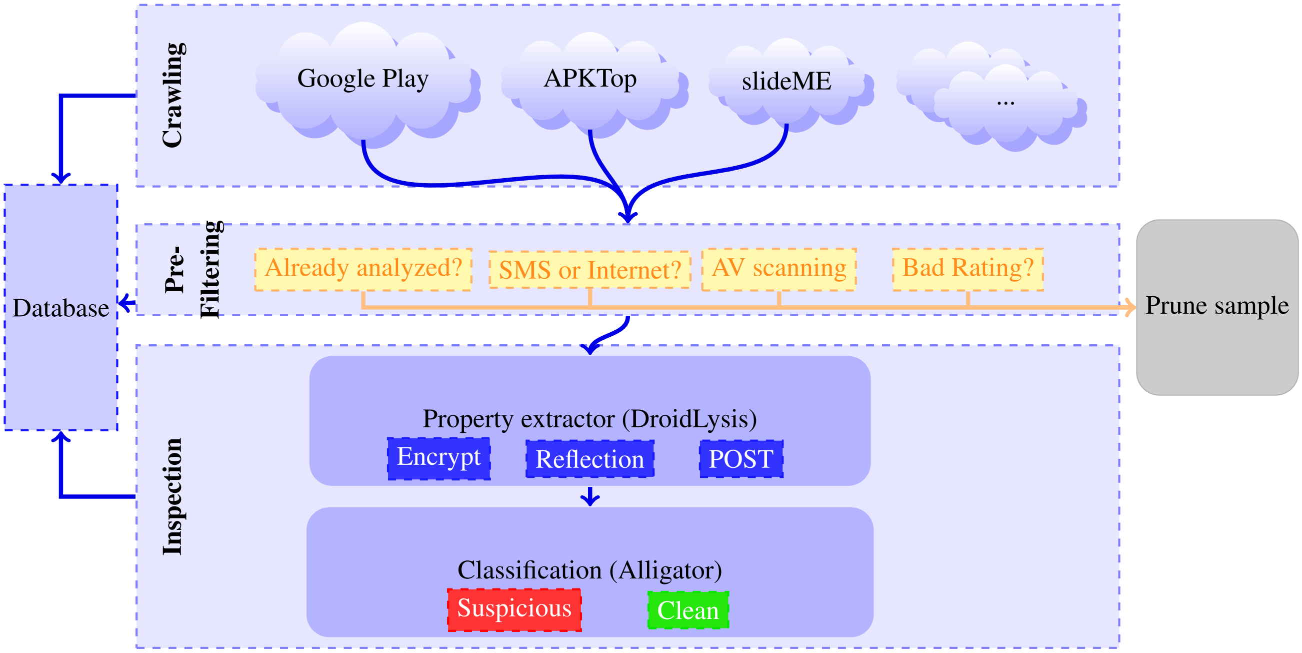


Figure 1: SherlockDroid Architecture

SherlockDroid is the name given to the entire architecture illustrated at Figure 1. It consists of three different steps:

- **Crawling**. Crawlers download samples from various marketplaces. Currently, we have crawlers for Google Play, APKTop, AppsApk, SlideME, and a generic crawler which recursively parses a URL for Android applications. See Section 4 for more details.
- **Prefiltering**. This layer is in charge of pruning samples which are not interesting to analyze for a reason or another. Our implementation prunes samples which are:
  - already detected by an AV engine
  - already analyzed by SherlockDroid
  - have neither INTERNET nor SMS permission

  Some forms of pre-filtering may also be implemented within the crawlers itself. For example, our Google Play crawler only considers applications with a rating less than 2 and download counts smaller than 1,000,000. This is a customizable choice, and of course, other prefilters may be contemplated.

- **Inspection**. When the entire architecture is named SherlockDroid, inspection can be seen as the brains of Sherlock Holmes. It is the place where, based on collected evidence, Sherlock Holmes decides whether a sample is suspicious or not.

  This step expands in 2 sub-steps which are shown at Figure 1:

  - Property extraction. First, Sherlock Holmes inspects the sample and methodically collects as much information or evidence as possible for the case. Currently, information is extracted from the static analysis of 289 different properties. This work is handled by a tool named *DroidLysis* - see Section 5.
  - Classification. Then, the collected information - which can be seen as the sample's *profile* - is sent to an open source learning and classification tool named *Alligator*. Like an inspector's dog, based on prior training (learning), Alligator sniffs the sample and tells the inspector whether the sample is suspicious or not. See section 6.

Samples and current SherlockDroid steps are regulated in a SQL database. For each sample, the database keeps track of various parameters such as its origin, hash and also the current SherlockDroid status: if the sample has been pre-filtered or not, if it is detected by the AV scanner or not, if it has been classified by Alligator yet or not.

3.2 SherlockDroid is not an Anti-Virus scanner

The architecture of SherlockDroid highlights an important point: **SherlockDroid is not an anti-virus scanner**. Indeed:

1. Its goal is not to detect *any* malware, but *new unknown (undetected)* malware. As a matter of fact, all downloaded samples are scanned by an AV engine, and known malware are ignored/trashed by SherlockDroid.
2. SherlockDroid is not able to say for sure that a sample is clean or malicious. It only says a sample *looks clean or suspicious*. In practice, it outputs results such as:

   ```
   com.popularapp.periodcalendar.apk:  regular
   (regular:5345.415814546147,
   malware:3136.2452100231844)
   ```

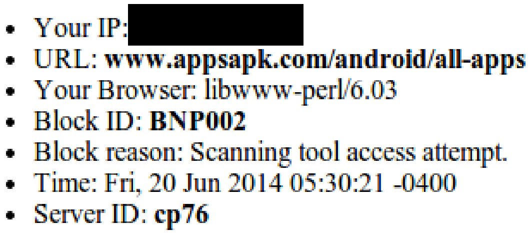
   which means it believes `com.popularapp.periodcalendar.apk` to be clean, its score being higher for resemblance with clean files than with malware. Based on score differences, we are furthermore able to classify the sample as light / medium / strong clean or malware.
3. On an implementation point of view, SherlockDroid does not use any *signature* mechanism, like traditional AV engines do, to detect a given malware family. On the contrary, it relies on several of properties it finds (or not) to build a profile of the sample, and then classifies it. This design is rather comparable to *heuristics*, which often *complement* AV scanners.

4 Crawlers

In theory, the implementation of marketplace crawlers is nothing difficult. In practice, as most marketplaces hide or provide indirect access to Android application packages, it requires some reverse engineering of the downloading protocol.

On Google Play, this protocol is *Protocol Buffers*, often shortened Protobuf and we use [Gir12]'s unofficial Python API to access it. SlideMe, ApkTop and AppsApk provide a proprietary search and download front-end that we simulate in order to download the applications. Our experience shows that this reverse engineering has to be re-done regularly (every 3 or 4 months usually) because marketplaces evolve, applications are no longer found at the same location etc, so crawlers stop to work.

In several cases, marketplaces try to prevent from being crawled. We notice for instance that AppsApk blocks standard download requests which originate from Perl (Figure 2). Fortunately, this protection is easily bypassed by setting a real Android User Agent.



- Your IP: [redacted]
- URL: **www.appsapk.com/android/all-apps**
- Your Browser: libwww-perl/6.03
- Block ID: **BNP002**
- Block reason: Scanning tool access attempt.
- Time: Fri, 20 Jun 2014 05:30:21 -0400
- Server ID: **cp76**

Figure 2: AppsApk marketplace blocking crawlers by User-Agent

The marketplace also banned our IP address. We bypassed the measure by using a different Tor connection for each download.

Google Play too is painful to crawl: it returns at most 500 applications for any search, prevents creation of too many Google accounts from the same IP addresses and bans accounts which show a too high downloading activity. To prevent this, we have created 96 different fake Google accounts and roll on this list for downloads. Actually, the creation of Google accounts itself is cumbersome, in particular the generation of a valid Android ID, so we used [Akd] to automate it. We also used different IP addresses, using a different Tor connection for each download to appear as coming from different location.

Another issue we faced is that marketplaces seldom provide a direct list or a link to *all* applications they have. In the best cases (e.g ApkTop), we are able to request a search for all applications, get pages of response in return and crawl applications on each page one by one. On Google Play, as we have explained previously in [AS12], the list of returned applications is different depending on your search, location and device. Our crawler copes with this limitation by simulating searches on a pre-defined word list of keywords. Like [VGN14], we could have searched from an exhaustive list of dictionary words of different languages, however we will believe that such dictionaries have several redundant (e.g 'operator' and 'operators') and useless words (e.g 'the', 'a', 'an'...) for searches.

Finally, for correct processing, crawlers must also sanitize URLs and application package names. This task is not difficult but mandatory because scripts usually do not like spaces, punctuation or Unicode characters. They must be trimmed or replaced for SherlockDroid to be able to operate correctly.

5 Property extraction (DroidLysis)

In 2012, the early heuristics engine of [AS12] was extracting 39 properties. In 2013, we were extracting over 140 properties [AA13]. In 2014, we are now extracting 289 different properties. A few examples of those properties are listed in Table 2.

As we were regularly increasing the number of extracted properties, we experienced the downsides of this progress: each time a new property was being added, it is preferable to train again the classification tool, i.e re-do the learning phase of Alligator. To do so, we needed updated training sets which included the value for the new property. However, re-extracting all the properties of a training set of, for example, 100k malware is a lengthy procedure which can last several days on regular desktops (see Section 7.2). So, first, we designed a simple work distribution protocol. All results and samples are centralized on a main host. Client workers connect to this host to get a batch of work (e.g 100 samples), extract properties for those samples with DroidLysis and return the result to the main host where data is consolidated. Second, in several cases, we remark that re-extracting all properties for the training sets is overkill. Merely *updating* the sets with a default value for the new property is satisfactory. So, we implemented a property conversion tool where we keep track of changes (added properties). At any time, we are then able to convert properties extracted by a newer version of DroidLysis to an older set of properties understood by Alligator (we remove the added properties). This workaround limits the number of times we have to re-extract properties of our training sets, and the number of time we have to re-train Alligator. We now only do this once in a while, after a bunch of major modifications.

Property type	Example of properties
File properties	type, size, small file indicator, number of classes...
Certificate properties	country, hash algorithm used to sign the certificate, use of development or debug certificates...
Manifest properties	request for given permissions (Internet, logs, contacts...), presence of a SMS or call listener, requested SDK version, number of activities, libraries, permissions, providers...
Dalvik Executable properties	correctness of checksum and hash, magic number, odex indicator
Dalvik code properties	
Attempts to access to data or resources	getAccounts(), getAllBookmarks(), getAllVisitedUrls(), getDeviceId(), getHostAddress(), getLatitude(), getSubscriberId(), loadDex(), loadLibrary(), openNonAsset(), Camera.open()...
Obfuscation	use of encryption, detecting presence of a debugger or an emulator, obvious use of class renaming
Communication	sendTextMessage(), Socket.init()...
Other APIs	getInstalledPackages(), randomUUID(), setJavaScriptEnabled() ...
Command properties	executing native commands, ssh to remote servers, access to system logs
ARM executable properties	execve, kill, presence of known exploit kits (Exploid, mempodroid...), mounting devices, starting shells
Kit properties	presence of numerous adkits (admob, admogo, flurry ...), development SDKs (Adobe AIR, Apache...) , statistics kits (chartboost, mixpanel...), packers

Table 2: Example of properties extracted by DroidLysis

During the testing phase of SherlockDroid, we also noticed several samples were being flagged as suspicious because they were using an advertisement kit with borderline implementation. We reported abuse as much as possible, however in many cases, the context is not strictly speaking malicious. To cope with this issue, we separate properties found in the application's code from those found in third party code. We identify 143 third party kits, and, by default, only consider properties outside those kits. This enables us to tell the difference between an application which has given properties and an application using a third party kit that uses the same properties. For example, a TicTacToe application retrieving GPS location appears different to DroidLysis from a TicTacToe application which uses Admob - which retrieves GPS location too. The first application might be suspicious, whether the second sounds more benign. This mechanism however goes with a drawback currently because third party kits are identified based on their namespace. So, malware trojaning third party kits can evade detection. This happens from time to time: Android/RuSMS.AO hides within Adobe AIR's namespace (com.adobe.air). In such cases, DroidLysis can be asked to scan the entire application, however, currently, this is a manual option.

6 Classification (Alligator)

Alligator is an open-source tool for classification. It is agnostic of the anti-virus world and meant to decide whether a given sample looks more like samples in a given set or another. The sets are called *clusters*, and Alligator can virtually be used to classify anything: fruits / vegetables, male / female, hot / cold etc.

In the case of SherlockDroid, we use Alligator to decide between clean (regular cluster) and malware (malware cluster).

In a first initialization step, Alligator needs to be trained. This is also called the *learning phase*, where we provide examples of typical clean files (learning regular cluster) and examples of malware (learning malware cluster). This phase can grow quite long (see Table 3) and is only meant to be done once in a while.

The main strengths of Alligator are threefold:

1. Multi-algorithm learning. Most classification tools rely on one given distance metric (e.g., Euclidian, Pearson correlation, etc.). Alligator relies on *several* algorithms whose importance for correct identification is automatically computed during the learning phase. Thus, Alligator provides a strong automated help to select clustering algorithms.

2. Favor a cluster over another. During the learning phase, we are able to tell Alligator the importance we give to correct classification in a cluster compared to the other. In the case of SherlockDroid, we tune the learning to minimize False Positives. False Negatives are important too, but only come as a second priority in our case.

3. Lightweight and simplicity. Alligator is a stand-alone Java program. It does not require several prior packages - only Java - and its installation is a matter of unzipping a TAR archive. Its learning is customizable. We compared Alligator to other classification tools such as SVM and Adaboost, and Alligator showed better results in each case. We intend to publish this comparison, but it is out of scope for Hack.Lu.

For more details on Alligator, please refer to [AA13, Apv].

7 Results

7.1 Learning and classification results

The performance of Alligator is evaluated as follows:

1. Alligator is provided with two learning clusters: 11,249 known clean files and 50,000 known malware. Those learning clusters train Alligator. At the end of its training, Alligator parses the regular and malware clusters. For each sample, it pretends not to know from which cluster it comes from and guesses whether it is clean or not. Its correctness is evaluated. We usually get excellent rates (99.9%) at this stage.

2. Then, Alligator is asked to classify two guess clusters it does not know of. Those clusters are made of samples which are *not* in the learning clusters. The samples are actually newer (downloaded at later dates). For those samples, we actually know which ones are malicious and those which are clean. We ask Alligator to classify the samples automatically, and check how well it performed. This test is harder on Alligator, but realistic. We are very happy with the results (see Table 3) : only few False Positives (0.69%) - which is our primary interest. The False Negative rate also remains quite low (4.57%).

	Regular cluster size	Malware cluster size	Time	False Positives	False Negatives
Learning time on samples collected before June 14, 2014	11,249	50,000	46 minutes	0.04%	0.05%
Guess on new samples on samples collected between June 14 and June 23, 2014	1,448	3,062	13 minutes	**0.69%** (10)	**4.57%** (140)

Table 3: Learning and classification results of Alligator

Actually, more than just classifying as clean or malware, Alligator is able to say how clean or malicious a sample looks to it. Does it look slightly malicious ("light malware") or very malicious ("strong malware")? etc. Table 4 shows the detailed classification of our guess clusters. In particular, we note that all False Positives (clean samples wrongly classified as malware) are classified as *light* malware, i.e potentially samples on which Alligator does have doubts.

Samples to classify collected between June 14 and June 23, 2014	Regular			Malware		
	Strong	Medium	Light	Light	Medium	Strong
Clean samples	862 (59.5%)	207 (14.3%)	369 (25.5%)	10 (0.7%)	0	0
Malicious samples	0	2 (0.1%)	138 (4.5%)	475 (15.5%)	0	2447 (79.9%)

Table 4: Detailed classification results of Alligator

Note that Alligator scales to larger volumes. We have had it learn with 80k clusters [AA13], and classify guess clusters of 486,890 samples.

7.2 Performance

SherlockDroid tasks run in different processes: one per crawler, one for pre-filtering, one for property extraction (DroidLysis), one for classification (alligator). Those tasks are scheduled by the OS, and the time they take varies on various factors like size of package, current bandwidth, proportion of Dalvik code to disassemble, size of clusters, available RAM etc. Examples of timing on a standard non-dedicated desktop are shown at Table 5. Those measures show no obvious bottleneck, however the measures should be improved for a firm conclusion.

7.3 Filtering

SherlockDroid is running on a research server of Fortinet's FortiGuard Labs since July 2013. However, due several factors (upgrades of DroidLysis or Alligator, system maintenance etc) it has only been up sporadically a few days in a row during 2 different campaigns: July-August 2013 and June-July 2014 (see Table 6). We intend to run it full time as soon as possible.

For 100 samples...	Time
Download	≈ 13 minutes
Property extraction	≈ 9 minutes
Classification	≈ 14 minutes

Table 5: SherlockDroid approximate processing speed for 100 samples on a non-dedicated Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz with 8G of RAM. No obvious bottleneck.

Campaign date	Nb of samples crawled during the campaign	Nb of suspicious flagged by SherlockDroid
July-August 2013	30,000	346
July 2014	88,369	62
Total	118,369	408

Table 6: Tightening SherlockDroid's filtering

SherlockDroid has scanned 118,369 applications. We notice that its filtering has tightened - a good sign in our case - as recently, it has crawled 88,369 applications and only raised the alarm for 62 suspicious samples. The first of those 62 samples turned out to be Android/Odpa.A!tr.spy. Then, there are three different versions of Riskware/Blued!Android (leaking name, GPS coordinates, clear text password, age and weight). The other samples are still being analyzed at the time this paper is written. To help analyst process the most suspicious samples first, Alligator can display sorted output.

7.4 Spotting malware

So far, SherlockDroid has identified the following:

- It has detected 2 malware: Android/MisoSMS.A!tr.spy (December 2013) and Android/Odpa.A!tr.spy (July 2014). MisoSMS had actually been in Fortinet's sample database for some time, but had surprisingly not been detected by any anti-virus vendor. The Odpa sample was unknown to Fortinet and was believed to be in the early days of its life. On VirusTotal, only 4 / 31 vendors were aware of it.
  See `http://blog.fortinet.com/Clean-for-the-phone--but-not-clean-in-the-code/`.
- It has discovered 5 potentially unwanted applications (PUA): Adware/Geyser!Android and Riskware/SmsControlSpy!Android in August 2013, Riskware/Zdchial!Android and Riskware/SmsCred!Android in November 2013, and Riskware/Blued!Android in July 2014. Potentially Unwanted Applications can be seen as borderline cases which are neither fully clean nor really malicious. Adware/Geyser was sending the victim's GPS coordinates in clear text. Riskware/Zdchial leaks the IMEI and IMSI to a remote server, and Riskware/SmsCred sends login and password credentials in clear text...
  See `http://blog.fortinet.com/Alligator-detects-GPS-leaking-adware` and `http://blog.fortinet.com/Alligator-at-GreHack`.

From those discoveries, we note that SherlockDroid seems particularly successful at spotting spyware. We attribute this to the fact that spyware often raise several boolean properties at extraction (sending SMS, listening to SMS, placing calls, leaking IMEI, IMSI etc) which helps Alligator identify their eccentricity.

7.5 Typical classification errors

The most common cases where SherlockDroid fails to correctly classify a sample usually fall among one of these two categories:

1. Applications sending e-mails or SMS for bug reports. Those applications may even query system logs and multiple system properties to fill out the bug report. Doing so, they set several boolean properties as true, and mislead Alligator in thinking the application is malignant. Manual study of the context reveals the case is not malicious.

2. UI or system tweaking applications or SMS management tools. Those tools require lots of low level tweaks (su, busybox, system commands...) which, once again, trigger false alarms for SherlockDroid.

We are currently contemplating solutions to solve those issues. So far, the analysis of SherlockDroid's errors has always been extremely helpful to improve it. As we remarked in Section 5, this is for instance how we got the idea to identify third party kits and rule them out.

8 Conclusion - Future Work

SherlockDroid is a scalable crawler and detection framework to spot new Android malware. It has crawled 118,369 samples on marketplaces. During our tests, it has extracted properties and classified over 480k known samples. In concrete terms, in a year, it has directly identified 6 new malware and potentially unwanted applications.

In the future, we plan to:

- Run SherlockDroid full time for months and identify potential bottlenecks
- Enhance the crawling of Google Play by searching for similar or recommended applications that we have already downloaded
- Keep current crawlers up-to-date and add new crawlers for other marketplaces
- Collect contextual information to some properties so as not to raise false alarms for benign cases such as applications implementing bug reporting by e-mail
- Scientifically measure and select the best size of learning clusters with regards to classification time and false positive/false negative percentage

Acknowledgements: we wish to thank Ruchna Nigam for her help on SherlockDroid.

Additional information

8.1 Axelle Apvrille

- Location: France
- Employer: Fortinet
- Brief biography:

  Axelle Apvrille's is currently a Senior Antivirus analyst and researcher at Fortinet, where she more specifically looks into mobile malware. She has been very happy to present at Hack.Lu in 2013, and in various other conferences (VB, Hashdays, Hacktivity, Insomni'Hack...).

  Axelle is a member of the pic0wn CTF team. That's the 3-member team which managed to get in the Hall of Fame of local teams at Hack.Lu 2013 [AAB14] ;)

- List of publications and papers: `http://wikisec.free.fr/papers/papers.html`
- Presentation or educational experience: presented a several conferences, I've also been a lecturer in several French engineering schools.

8.2 Ludovic Apvrille

- Location: France
- Employer: Telecom ParisTech
- Brief biography:

  Ludovic Apvrille completed a Ph.D. in 2002, in the Department of Applied Mathematics and Computer Science at ISAE, in collaboration with LAAS-CNRS and Alcatel Space Industries (now, Thalés Alenia Space). After a postdoctoral term at Concordia University (Canada), he joined LabSoc in 2003 as an assistant professor at Telecom ParisTech, in the Communication and Electronics department. He obtained his HDR (Habilitation á Diriger les Recherches) in 2012. His research interests focus on tools and methods for the modeling and verification of embedded systems and Systems-on-Chip. Verification techniques target both safety and security properties. He's the leader of the free and open-source UML/SysML toolkit named TTool, and of Alligator.

  Ludovic is also member of the pic0wn CTF team :)

- List of publications and papers: `http://perso.telecom-paristech.fr/~apvrille/research.html`
- Presentation or educational experience: full time researcher and lecturer :) so plenty of conferences, and courses on operating systems, embedded systems etc.

8.3 Submission information

- Probable presenter: Axelle
- Reason why this material is innovative or significant: finding new malware is difficult. Automated systems often look cool in theory but show no results in practice. Here, with SherlockDroid, we have a system which is operational and has already identified malware in the wild.
- Prepared material: this paper and a demo on SherlockDroid
- Already presented: no

References

[AA13] Ludovic Apvrille and Axelle Apvrille. Pre-filtering Mobile Malware with Heuristic Techniques. In *GreHack*, pages 43–59, November 2013. Grenoble, France.

[AAB14] Ludovic Apvrille, Axelle Apvrille, and Pierre Bogossian. Retour d'expérience sur quelques épreuves de Hack.Lu. *MISC Magazine*, (73):50–86, May-June 2014. in French.

[Akd] Akdeniz. Google Play Crawler JAVA API. `https://github.com/Akdeniz/google-play-crawler`.

[Apv] Ludovic Apvrille. Alligator. `http://perso.telecom-paristech.fr/~apvrille/alligator.html`.

[AS12] Axelle Apvrille and Tim Strazzere. Reducing the Window of Opportunity for Android Malware. Gotta catch'em all. In *Journal in Computer Virology*, volume 8, pages 61–71, 2012.

[AZ13] Zarni Aung and Win Zaw. Permission-Based Android Malware Detection. *International Journal of Scientific and Technology reseach*, 2, March 2013.

[BSB$^+$10] Thomas Bläsing, Aubrey-Derrick Schmidt, Leonid Batyuk, Seyit A. Camtepe, and Sahin Albayrak. An Android Application Sandbox System for Suspicious Software Detection. In *5th International Conference on Malicious and Unwanted Software (MALWARE'2010)*, Nancy, France, France, 2010.

[BZNT11] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[DMSS12] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. MADAM: A Multi-level Anomaly Detector for Android Malware. In *Computer Network Security - 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security, MMM-ACNS*, volume 7531 of *Lecture Notes in Computer Science*, pages 240–253, St. Petersburg, Russia, "October" 2012. Springer.

[EGC$^+$10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[Gir12] Emilien Girault. Reversing Google Play and Micro-Protobuf applications, September 2012. `http://www.segmentationfault.fr/publications/reversing-google-play-and-micro-protobuf-applications/`.

[SKE$^+$12] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "andromaly": a behavioral malware detection framework for android devices. *J. Intell. Inf. Syst.*, 38(1):161–190, February 2012.

[SSL$^+$12] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Alvarez Maranon. Puma: Permission usage to detect malware in android. In Alvaro Herrero, Vaclav Snasel, Ajith Abraham, Ivan Zelinka, Bruno Baruque, Hector Quintian-Pardo, Jose Luis Calvo-Rolle, Javier Sedano, and Emilio Corchado, editors, *CISIS/ICEUTE/SOCO Special Sessions*, volume 189 of *Advances in Intelligent Systems and Computing*, pages 289–298. Springer, 2012.

[Str11] Tim Strazzere. Security Alert: Legacy makes another appearance on Android Market, Meet Legacy Native (LeNa), October 2011. `https://blog.lookout.com/blog/2011/10/20/security-alert-legacy-makes-a-another-appearance-on-android-market-meet-legacy-native-lena/`.

[VGN14] Nicolas Viennot, Edward Garcia, and Jason Nieh. A Measurement Study of Google Play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 221–233, New York, NY, USA, 2014. ACM.

[XZSZ10] Liang Xie, Xinwen Zhang, Jean-Pierre Seifert, and Sencun Zhu. pBMDS: a behavior-based malware detection system for cellphone devices. In *Proceedings of the third ACM conference on Wireless network security*, WiSec '10, pages 37–48, New York, NY, USA, 2010. ACM.

[YY12] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *USENIX Security Symposium*, pages 569–584, 2012.

[ZWZJ12] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 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, USA, Feb 2012.