Journal of Computer Virology and Hacking Techniques manuscript No.
(will be inserted by the editor)

SherlockDroid: a Research Assistant to Spot Unknown Malware in Android Marketplaces

Axelle Apvrille · Ludovic Apvrille

Received: date / Accepted: date

Abstract With over 1,400,000 Android applications in Google Play alone, and dozens of different marketplaces, Android malware unfortunately have no difficulty to sneak in and silently spread. Known malware and their variants are nowadays quite well detected by anti-virus scanners. Nevertheless, the fundamentally new and unknown malware remain an issue.

To assist research teams in the discovery of such new malware, we 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.

In our tests, we extracted properties and classified over 480K applications. During two crawling campaigns in July 2014 and October 2014, SherlockDroid crawled over 120K applications with the detection of one new malware, Android/Odpa.A!tr.spy, and two new riskware. With previous findings, this increases SherlockDroid and Alligator's "Hall of Shame" to 8 malware and potentially unwanted applications.

Keywords Android, malware, classification, static analysis, security

1 Introduction

With the plethora of new Android applications (between 20K and 40K monthly according to AppBrain) malware authors can easily sneak in malicious applications.

Some of these are fortunately filtered by Anti-Virus products. Detecting known malware, in particular, is easily done by matching a hash (called a 'signature' in the AV world) on parts of its code. Variants can also be detected using more or less complex 'generic signatures' i.e. patterns that match several samples at a time [17]. Nevertheless, *all* anti-virus scanners face difficulties when it comes to detecting *truly new* malware (new families, 0-days...), i.e. a malicious sample which does not look like anything before.

On PC and Mac desktops, AV vendors tackle this issue by complementing their traditional scanners with heuristic engines, machine learning or sandboxes. However, those mechanisms are *by nature imperfect*. We remind that [13] proved that there is no algorithm capable of perfectly deciding whether a program is malicious or not. In practice, AV vendors only use heuristics and sandboxes for reporting (gathering information on potential new malware) or user warning (raising an alert), but never for detection because of the inherent risk of False Positives. Indeed, False Positives (clean samples detected as malicious) is what AV

Axelle Apvrille
FortiGuard Labs, Fortinet,
120 rue Albert Caquot, 06410 Biot, France
Tel.: +33-(0)4-89-87-05-42
E-mail: aapvrille@fortinet.com

Ludovic Apvrille
Institut Mines-Telecom, Telecom ParisTech, CNRS/LTCI, Biot, France
Tel.: +33-(0)4-93-00-84-06 Fax.: +33-(0)4-93-00-82-00
E-mail: ludovic.apvrille@telecom-paristech.fr

vendors fear the most, because of the immediate surge in customer requests and bad press [22] [30] [25]. On mobile devices, the situation is even more complicated. The device does not have the resources to run a sandbox. Vendors typically ask end-users to send their applications for remote analysis in their "cloud". Once again, this is a convenient mechanism to gather statistics and/or raise alarms on suspicious applications, but *it is not directly used for malware detection* for fear of False Positives. As for heuristic approaches and behavior analysis mechanisms, they are yet in their early days [14] [34] [10] and have not been evaluated over large volumes of malware, as further explained in Section 2.

Thus AV analysts have to find truly new mobile malware themselves, manually inspect suspicious samples and update signature databases as quickly as possible: a task which is even harder than finding a needle in a haystack.

To address this problem, we suggest to assist analysts with a tool that will automatically inspect the haystack for them, and only output a handful of suspicious items it finds. This tool is named *SherlockDroid*. We highlight it is *not* an anti-virus scanner: it aims at finding only *new* malware and will completely ignore already detected malware. SherlockDroid outputs only a few applications: a set small enough for analysts and researchers to manually inspect. Its strength resides in selecting the most suspicious applications in that small set to analyze, so that, at least, time shall not be wasted on clean applications. While it is true that SherlockDroid will miss several new malware in the wild, we argue that, without SherlockDroid, even more malware remain undetected.

SherlockDroid relies on the automated combination of marketplace crawlers, filtering and property extraction tools, and classifiers. It is meant to process a large quantity of applications, filter out applications which are either clean or known malware, and finally keep a very small set of suspicious samples. The contribution of this paper focuses on issues encountered during implementation of marketplace crawlers, property extraction and the tuning of Alligator to minimize False Positives. Another contribution of the paper consists in the evaluation of our approach on a huge dataset (480K clusters).

This paper is organized as follows. First, section 2 compares SherlockDroid with other similar heuristic-based approaches. Then, we describe SherlockDroid's architecture (section 3), and detail implementation issues for crawlers, static analysis property extraction in sections 4 and 5. Finally, we focus on classification results (section 6) and discovered malware (section 7).

2 Related work

2.1 Assisting the discovery of unknown malware

Nearly all research work we are aware of are tested on known malware datasets. They consequently confirm maliciousness of *known malware*, but do not spot *unknown malware like new families or 0-days*. In some cases, it is simply not their goal [21] [2] [15] [33]. In other cases, the frameworks were never tested in a real environment or with too few samples to get the chance to discover any new malware (e.g., [35] [14] [34] [7]). Some research work crafted artificial, i.e. self written malware [8] [31] [10]. Besides the dangerosity of such a technique, detecting such a malware is not the same as detecting unknown malware because the malware authors are also the designers of the defense system, which seriously biases the study. Drebin [6], although it has not discovered any new malware, is at least evaluated against unknown malware, where it obtains detection rates between 50 and 75%. This is far lower than with SherlockDroid: over 98% - see section 7). AppsPlayground [26] does not detect new unknown malware either, but succeeds at pin-pointing undetected privacy exposure.

DroidRanger [37] *is the only previous public work which discovered two new malicious families*. It is however limited by design to detecting (i) variants of known malware families or (ii) unknown malware that dynamically load untrusted code, via the implementation of only two heuristics. SherlockDroid relies on a far larger set of heuristics (see Section 5), and leads to the discovery of two new malware and six potentially unwanted applications.

2.2 Scalability to Android marketplaces

Few studies have been experimented on large scale. Several systems have not been designed to crawl marketplaces at all (e.g., [34] [31] [35] [8] [27] [14] [15]) or their applicability in practice for the anti-virus industry is still to demonstrate. Six systems only have been tested over 100K applications: AndRadar [21], Andrubis [2], Drebin [6], DroidRanger [37], PlayDrone [33] and SherlockDroid.

We discussed Drebin and DroidRanger previously. The other three have different goals from SherlockDroid. AndRadar focuses on tracking distribution of known malware among various marketplaces. Andrubis is an Android sandbox, i.e. researchers use it to run an application and get security-related logs and traces. PlayDrone gathers statistics on Google Play applications. It is not the first Google Play store crawler (see [3] in 2012), but the first statistical study on crawled applications. Its crawlers are nonetheless interesting to compare to SherlockDroid's:

- PlayDrone uses Amazon EC2 cloud services to leverage computation power. We re-used old unused desktops, which is less efficient, but 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 creation of Android IDs using an open source toolkit [1].

Although it makes sense to entirely crawl Google Play for statistical goals, but in our case, for malware detection, this is over-kill. For instance, applications which do not connect to Internet, do not make calls and do not send SMS are less likely to be be malicious. According to [18], Trojan-SMS represent 57.08% of mobile malware, and at least Adware, Trojan-Bankers and Trojan-Downloaders (9.85%) require Internet.

2.3 Machine learning

Machine learning and classification are a common solution to classifying Android applications as clean or malicious (see Table 1).

Mast [11] relies on correlations between 208 features. MADAM [14] uses the *k-NearestNeighbor* algorithm for classification. This algorithm is similar to Alligator's proximity algorithm. Andromaly [31] has been tested with several classification algorithms, including *k-Means*. It selects the most accurate classification algorithm for a series of input data (clean, malware). Similarly, the PUMA [28] approach compares the accuracy of several classification algorithms (e.g., randomforest, SVM, decision trees, etc.) in order to detect malware from application permissions. They conclude that decision trees is the best classification algorithms for their features (best True Positive Ratio: 0.92). Yet, SherlockDroid is the only one that combines classification algorithms. The training phase computes a weight for each supported algorithm, e.g., *k-NearestNeighbor*, *correlations*, *SVM*.

Other contributions, such as RobotDroid [36], customize classification algorithms (e.g., SVM) to improve results. In our case, we have large datasets, but they are strongly unbalanced (few clean, many malware). Thus, Alligator has specific options to handle that issue, and is very efficient to correctly handle large and unbalanced datasets, with an increased computation cost since - in the general case, it has to execute several classification algorithms, and not only one. Classification accuracy and computation time are further discussed in section 7 ("Results").

Project name	**Classification algorithms**
MAST	Correlation-based
RobotDroid	SVM
Drebin	SVM
pBMDS	Hidden Markov Model
Andromaly	k-Means, Logistic Regression, Histograms, Decision Tree, Bayesian Networks, Nave Bayes
Crowdroid	k-Means
PUMA	SimpleLogistic, NaiveBayes, BayesNet, SMO, IBK, J48, RandomTree, RandomForest
Permission-based framework [7]	k-Means
SherlockDroid	Combination of any: standard deviation, several variants of k-NNs, correlations, probabilities, ϵ-clusters, SVM

Table 1: Classification approaches of similar approaches vs. SherlockDroid

3 Overview of SherlockDroid

3.1 Main components of SherlockDroid

SherlockDroid is the name given to the entire platform illustrated at Figure 1. The 3-step methodology associated to this platform is as follows:

A/ **Crawling**. Crawlers download samples from various marketplaces. Currently, we have crawlers for the Play Store, APKTop, AppsApk, SlideME, Nduoa[2] and a generic crawler which recursively parses a URL for Android applications. See Section 4 for more details.

B/ **Analysis**. First, a pre-filtering stage prunes samples which are not likely to be interesting to analyzed are pruned. Those are:

- Already detected by an AV engine
- Already analyzed by SherlockDroid

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

[2] Respectively http://www.papktop.com/, http://www.appsapk.com, http://slideme.org/ and http://www.nduoa.com.

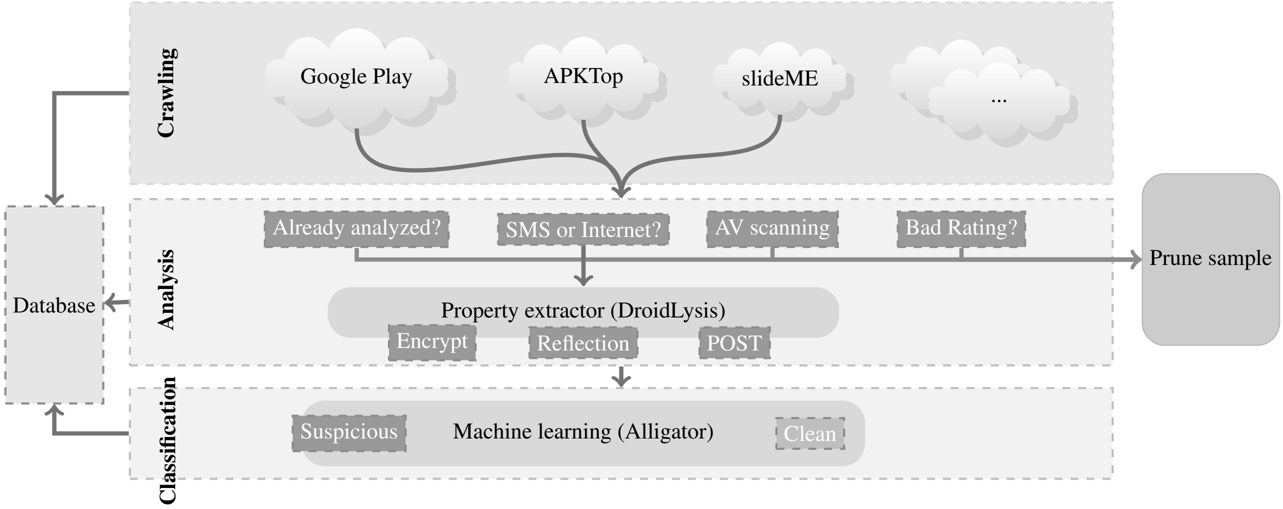

Fig. 1: SherlockDroid Architecture

– Neither INTERNET nor SMS permission[3]

Some forms of pre-filtering may also be implemented within the crawlers itself. For example, our Google Play crawler selects unpopular applications, hoping that end-users complain for a good reason (e.g., "not working" may indicate the application is fake - a mobile trojan).

Then, we extract information from the sample. We perform static analysis of 289 different properties. This work is handled by a tool named *DroidLysis* - see Section 5.

C/ **Classification**. The collected information is sent to an open source learning and classification tool named *Alligator*. It classifies the sample as 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 parameters such as origin, hash, date, status.

3.2 SherlockDroid is not an Anti-Virus scanner

The architecture of SherlockDroid highlights an important point: **SherlockDroid is not an AV scanner**. Indeed:

1. Its goal is not to detect *any* malware, but *new unknown (undetected)* malware, where this comprises unknown variants and, as much as possible, strongly different malware. As a matter of fact, all downloaded samples are scanned by an AV engine, and known malware are ignored 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 not*. In practice, it outputs a score of resemblance to the cluster of clean samples and a score of resemblance to the cluster of malicious samples. When the score is higher with the cluster of clean samples, the sample is believed to be clean. Based on score differences, we are furthermore able to classify the sample as light / medium / strong clean or malware.
3. From an implementation point of view, SherlockDroid does not use any *signature* mechanism to detect malware, like common AV engines do on PC or mobile devices. On the contrary, it extracts properties from the sample, and then classifies it. This design is rather comparable to *heuristics*, which often *complement* AV products [17].

4 Crawlers

An Android marketplace crawler is a program that systematically browses a marketplace to download applications it contains. As most marketplaces hide or provide indirect access to Android application packages, implementing a crawler starts with the reverse engineering of the download protocol. Depending on cases, this is more or less difficult. We faced the following issues:

(i) **Blocked user agents**. In an attempt to block automated downloads, some marketplaces restrict possible browser user agents.

[3] Note this is purely a customizable implementation choice. We might change it in the future if we notice malware commonly bypass those permissions.

(ii) **Download limit per IP address**. Some marketplaces (e.g., Google Play, AppsApk) are known to ban IP addresses [33] when they suspect "illicit activity". We bypass the measure by using a different Tor connection for each download. However, downloads from Tor are apparently banned on some marketplaces (e.g., SlideMe).

(iii) **Download limit per account**. Google's Play store returns at most 500 applications per search, and bans accounts with suspicious activity. To prevent this, we have created multiple fake Google accounts and iterate the list for downloads. Actually, the creation of Google accounts itself is cumbersome, in particular the generation of a valid Android ID, so we used [1] to automate it.

(iV) **Restricted application searches**. On Google Play, the list of returned applications is different depending on your search, location and device [3]. We cope with this limitation by simulating searches on a pre-defined list of keywords. Like [33], we could have searched from an exhaustive list of dictionary words of different languages, but this would mean wasting time on searching for redundant words (e.g., 'operator' and 'operators') or useless words (e.g., 'the', 'a', 'an'...).

5 Property extraction (DroidLysis)

5.1 Property definition and categories

In SherlockDroid, a *property* is a characteristic of an Android application, whether clean or malicious. Taken alone, a single property is always insufficient to determine its maliciousness, but their combination yields information.

DroidLysis properties fall in 4 different categories:

1. **File properties** (54/289). They correspond to characteristics found in 4 important files of applications: the package file itself (e.g., its size), the manifest (e.g., permissions requested, number of services ...), the certificate (issuer, algorithm, date...) and the Dalvik Executable (e.g., magic, correct hash).
2. **Dalvik code properties** (70/289). They are extracted from the Dalvik executable inside the Android package. For example, there are properties to detect certain APIs (e.g., `sendTextMessage`), actions (e.g., ACTION_CALL), intents (e.g., EXTRA_SUBJECT), constants (e.g., `POST`), Dalvik opcodes (e.g., nop, junk bytecode constructions such as [23]). We also detect implementation techniques such as JNI (e.g., use of jmethodID, jfieldID...), encryption (e.g., use of classes KeySpec, SecretKey...) or reflection (e.g., use of `Class.forName()`, `Method.invoke()`...). We detect some forms of obfuscation by analyzing the names of classes and methods (e.g., presence of classes and methods named a, b, c...).
3. **Assets, libraries and resources properties** (22/289). In these files, we pay particular attention to native code like identifying deliberate or hidden ARM executables in those paths and parsing those executables for potential exploits or risky system calls (*su*, *mount*, *execve*, *chmod*...). We are not aware of any other research work detecting such attempts. We also look for JavaScript, URLs, phone numbers that might be mentioned in configuration files, layouts or resources.
4. **Third party kits properties** (143/289). Android applications embed third party code for numerous reasons like advertisement, statistics or error reporting. DroidLysis detects the presence of those kits for reasons that we detail further in Section 5.4.

5.2 Property extraction

In DroidLysis, all properties are extracted statically. We use static analysis because it is particularly fast and values are easy to extract from Android application's file format.

Reciprocally, we do not use dynamic analysis because it slows down performances (time to install, launch and run the application) and because history of malware on PC shows that sooner or later malware authors implement techniques to behave differently when run in emulators, hypervisors or sandboxes.

As much as possible and for ease of manipulation, we choose to extract properties as strings. For Dalvik properties, this is possible because the Dalvik bytecode references each constant, field, method, class as a string. So, we can spot pieces of code which call a given method or perform a given action. Hence, we are able to extract properties on the application's features and behavior without actually having to run it.

Note that detecting API calls is more reliable than relying on the presence of specific permissions, because a permission may be requested and not used, or not requested and bypassed. It is true however that we might detect code which is never called: unless we build call graphs, static analysis fails on this point. Dynamic analysis has other drawbacks: it misses calls when code coverage is incomplete.

Third-party kit properties are extracted from the namespaces present in the Smali output. We discuss this more in detail in Section 5.4.

Resource properties are also extracted from strings as many of them are scripts or XML files, and thus human readable. When they include binary executables, we run the Unix `strings` command on them to spot particular calls (e.g., `pm install`) or system properties (e.g., `ro.kernel.qemu`). This has the advantage of letting us inspect executables without having to disassemble them.

The property extraction source code is unfortunately not public, to prevent malware authors from easily building evasion techniques around them[20].

5.3 Property relevance

We select interesting properties to extract manually, based on our experience of reverse engineering known mobile malware. We identify the mechanisms that enable them to conduct their malicious deeds, and extract corresponding properties.

As an illustration, we list below three examples of how malicious mechanisms were mapped to properties:

- Mobile spyware commonly intercept SMS messages and forward them to another recipient. To do so, they typically register themselves as high priority receivers (to receive SMS before other applications) and then delete the message so that the victim does not get any notification. We track this technique by (i) checking receivers in the manifest, and (ii) spotting calls to `abortBroadcast()` - the method use to delete a message from a queue.
- Mobile spyware are also interested in personal assets. We try to detect each time they access something which might be sensitive. For example, we track calls to retrieve the IMSI (`getSubscriberId()`), calls which list recently accessed URLs (`getAllVisitedUrls()`) etc.
- In several cases, e.g., Android/DrdDream, mobile trojans' graal is root access on the device. This can be achieved by different means, like using a root exploit (rage in the cage, mempodroid ...) or attempting to issue the Unix command 'su'. As some exploits are based on changing the wifi state of the phone to invoke a root shell, we detect CHANGE_WIFI_STATE permission. We also detect attempts to re-mount the system partition in read-write mode. As for the 'su' command, we detect attempts via calls to APIs like `Runtime->exec()`, `ProcessBuilder->start()` or `createSubprocess()`.

This empirical methodology has its obvious limits, but it has the advantage of generating properties which are well tuned to SherlockDroid's goal. Additionally, each property having a practical meaning, it helps us tune and debug incorrect classifications.

5.4 Handling advertisement kits

The amount of advertisement kits (68 different kits identified by [9]) and their high usage is an issue to property extraction for two reasons.

First, the same code will be parsed multiple times where a single extraction would suffice in theory. For example, a given version of AdMob will be found in numerous samples, and it is a waste of time to extract its properties at each occurrence.

Second, advertisement kits typically raise flags on properties retrieving geographical location, IMEI, country etc [24]. While any abuse should be reported, whether it is found in an adkit or not, properties found in adkits blur the application's real intent. The problem is the same for other third party kits such as statistics kits, licensing kits, gaming platforms or development kits (reporting crashes for example).

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, analyze them manually, and then, in future instances, 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.

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. For example, we have encountered this in Android/RuSMS.AO[16]. The malware hides within Adobe AIR's namespace (com.adobe.air). In such a case, we can ask DroidLysis to scan the entire application, but this is a manual option.

6 Classification (Alligator)

6.1 Purpose of classification

Classification relies on extracted properties in order to decide whether unknown samples are more likely to be clean or suspicious samples. Ideally, we expect the classification to lower as much as possible the False Positive Rate. Also, a score on clean / suspicious is preferable over a simple boolean, because it conveys the degress of suspiciousness and helps prioritize which samples should be inspected first.

6.2 Classifying with Alligator

Alligator is an open-source tool for classification [4]. Its algorithms and specificities have been published in [5]. 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, clean / malware 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 may be long (see Table 3) and is only meant to be done once in a while.

SVM [12] and Adaboost [29] are among the most well-known classifiers, especially for the classification of malware and images. Yet, we have selected the Alligator classification engine for the following reasons:

1. **Classification performance**. Most classification tools rely on one given distance metric (e.g., Euclidean, Pearson correlation, etc.), e.g., SVM. Alligator relies on *several* classification algorithms whose importance for correct identification is automatically computed during the learning phase. Adaboost also combines multiple classification algorithms (or the same algorithm with different parameters), but Alligator associates the weight to classification algorithms considering all classification algorithms at the same time, while Adaboost relies on an iterative approach to compute the weights of classification algorithms. Just like for Adaboost, Alligator provides an efficient automated help to select classification algorithms.

   We have compared Alligator with Adaboost and SVM for different kinds of clusters, e.g., for classifying clean / malware applications, and images (e.g., male/female identification, make-up/non make-up, etc.). For clean / malware classification, comparisons with SVM are provided in the result section (section 7). Several comparisons with images are provided in [19]. Basically, Alligator demonstrates an improvement of 9% when compared to SVM for the make up/non-make up categories, and for publicly available sets of images (FCB database).
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. Its learning is highly customizable, but still quite easy: the user just has to select the classification algorithms - and their parameters - he/she would like to experiment with.

7 Results

7.1 Performance

We have tested the performance of each step of SherlockDroid (crawling, pre-filtering, property extraction and classification) taken individually (see Table 2). Each performance test was run alone on a Xen virtual machine with an Intel(R) Xeon(R) CPU E5-2403 at 1.80GHz and 2GB RAM (this is not a very performant host nowadays). The results show that the slowest task in our case is crawling, but our bandwidth impacts crawling.

SherlockDroid step	Nb of samples processed	Total Time (seconds)	Rate (sample per second)
Crawlers	2,605	22,925	0.11
Pre-filtering	3,576	4,145	0.86
Prop. extraction	24, 126	4,380	5.51
Classification	4,574	34	134.53

Table 2: Individual performance measurements of each component of SherlockDroid

Alligator classifies 134 samples per second (see Table 2). The performance of Alligator's training and classification is depicted at Figure 2. The results depend on the number of samples in the learning clusters. As expected, the learning phase is much longer than the classification phase.

The training time is always better with Alligator than with SVM[4], but the classification time is lower for SVM. With clusters of 50K samples, Alligator training phase is almost 4 times faster than the one of SVM (Alligator: 1200 sec., SVM: 4400 sec.), but that gap tends to reduce when clusters get bigger. On the opposite, SVM is faster than Alligator during the classification stage, because SVM can prepare a classification model during the learning phase (the vector values), which is not possible for several algorithms of Alligator, e.g., for k-NN. Yet, the classification time remains reasonable, and is really low with regards to the learning time, even for very large learning clusters. For example, the classification of guess clusters takes around 6 minutes with 480 K learning clusters, and with 50 K clusters, it takes 7.5 ms per guess sample (see the curve at Figure 2).

As we however discuss in the next subsection, Alligator performs much better in terms of classification efficiency, in particular for the False Positive Rate.

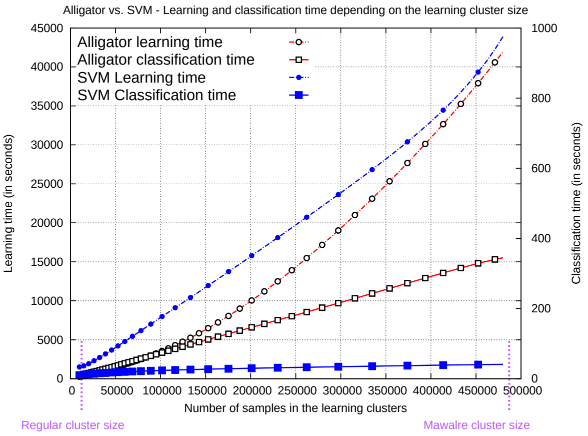

Fig. 2: Learning and classification time for Alligator and SVM.

7.2 False Positive and False Negative Rates

7.2.1 Test bed

The accuracy of the classification stage (performed with Alligator) is evaluated as follows:

- Alligator is provided with two learning clusters: 12,368 known clean files and 486,890 known malware, gathered before end of June 2014. Clean applications have been gathered either after manual analysis or from what we have assumed to be trusted sources, e.g., open-source applications for which we can check the code, or top developers/companies. The known malware cluster consists in Android samples detected by Fortinet and shared to or from other AV vendors.
  At the end of its training, Alligator parses both clusters (clean and malware). For each sample, it pretends not to know from which cluster it comes from and guesses whether it is clean or not. Then, knowing the cluster the sample belongs to, it evaluates the achieved correctness.
  We obtained excellent recognition rates (99.9%). We also evaluate the learning time during this step.
- Then, Alligator is asked to classify two guess clusters it does not know of at training phase, i.e. those guess clusters are made of samples which are *not* in the learning clusters.
  The guess clusters are actually made of samples which were downloaded at much later dates, i.e. after September 2014.
  We choose such guess clusters so as not to bias results. We know which samples are clean and which ones are malicious, but Alligator does not. There are 1,512 clean samples and 3,062 malicious ones.
  We thus ask Alligator to classify the samples automatically, and check how well it performed in terms of recognition rate.
- We compare the training and classification results with SVM in terms of recognition rate.

7.2.2 Classification results

Figure 3 depicts the classification performance in terms of overall classification efficiency, and of False Positive and False Negative Rates, and for both Alligator and SVM. A low False Positive Rate is our prime interest. Indeed, as we stated before, AV labs do not want to bother with the manual analysis of clean samples. On the contrary, missing

[4] We compared with *jlibsvm* [32]

malware is not so important, since so many are already not identified...

The curve shows that Alligator, whatever cluster size, always performs better than SVM for the False Positive Rate. SVM's FP rate is heavily impacted by cluster size: 65% of FP with 480K clusters. Alligator is far less impacted: 1% with 60K clusters and 1.78% with 480K.

Moreover, the average recognition rate is almost always better with Alligator. For 50k samples, precise results are given in Table 3.

Actually, more than just classifying as clean or suspicious, Alligator is able to say how clean or suspicious a sample looks to it. Does it look slightly suspicious ("light malware"), or very suspicious ("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. samples on which Alligator has doubts.

7.3 SherlockDroid in Operations

SherlockDroid is running on a research server of Fortinet's FortiGuard Labs since July 2013. However, due to several external factors (upgrades of DroidLysis or Alligator, system maintenance etc) it has only been up sporadically a few days in a row during 3 different campaigns: July-August 2013 , July 2014 and part of October 2014. We intend to run it full time as soon as possible.

Since July 2013, we have processed over 120,000 applications. However, as many of these applications were downloaded for various tests, there are many duplicates in that count and there are only 47,917 different applications in the database.

Details from the SherlockDroid database (see Table 5) also show that 6,530 samples were pruned because they did not ask for either the SMS or the Internet permission, and that 423 were found to be known malware or Potentially Unwanted Applications (PUA). So, a total amount of 83,119 were filtered out (duplicates, no SMS/internet, known malware) during the pre-filtering stage.

We tried to extract properties from the remaining 40,964 samples. This extraction failed in 330 cases because the sample was empty or damaged. The other 40,634 samples were classified by Alligator, and 65 applications were flagged as suspicious. We confirmed 5 of these to be malicious: 1 being Android/Odpa.A!tr.spy, another one being Riskware/Flexion!Android and 3 being different versions of Riskware/Blued!Android. The different findings of SherlockDroid are detailed in the next subsection.

7.4 Spotting unknown malware

SherlockDroid detected two new unknown malware: Android/MisoSMS.A!tr.spy (December 2013) and Android/Odpa.A!tr.spy (July 2014). See `http://blog.fortinet.com/Clean-for-the-phone-but-not-clean-in-the-code/`. It has also discovered 6 potentially unwanted applications: Adware/Geyser and riskware SmsControlSpy, Zdchial, SmsCred, Blued and Flexion. PUA can be seen as borderline cases which are neither fully clean nor really malicious. Geyser was sending the victim's GPS coordinates in clear text. Zdchial leaks the IMEI and IMSI to a remote server, and SmsCred sends login and password credentials in clear text. See `http://blog.fortinet.com/Alligator-detects-GPS-leaking-ad\-ware` 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 badly filtered samples

The most common cases where SherlockDroid fails to correctly classify (False Positive) 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, such as extracting the call stack or other contextual information for each property.

Other classification failures are due to implementation errors or missing properties (e.g., a missing property detecting a genuine third party kit). So far, the analysis of SherlockDroid's errors has always been extremely helpful

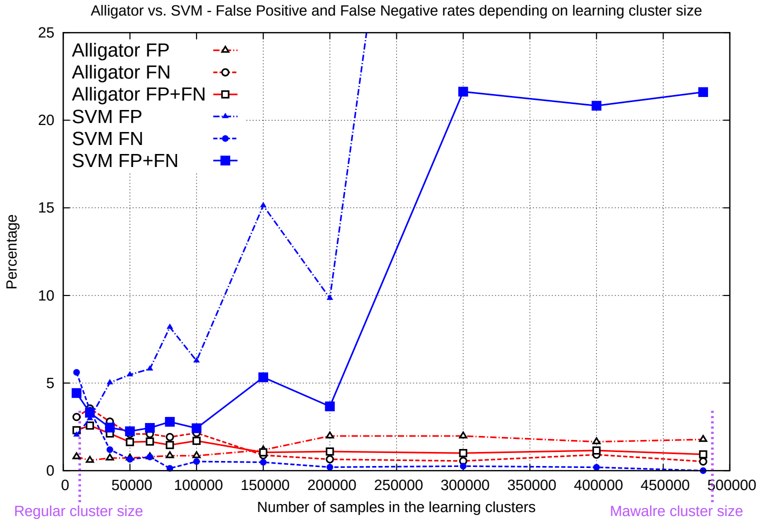

Fig. 3: FP and FN Rates for Alligator and SVM

	Regular cluster size	Malware cluster size	Time	False Positives	False Negatives
For Alligator:					
Learning on samples collected before end of June, 2014	12,368	50,000	20 min	0%	1.55%
Guess on new samples on samples collected in September 2014	1,512	3,062	34 sec.	**0.72**	2.09
For SVM:					
Learning on samples collected before end of June, 2014	12,368	50,000	1 h 15 min	-	-
Guess on new samples on samples collected in September 2014	1,512	3,062	21sec.	**5.48**	0.65

Table 3: Learning and classification results of Alligator

Samples	Regular			Malware		
	Strong	Medium	Light	Light	Medium	Strong
Clean	0 %	83 %	16.3 %	0.7 %	0 %	0 %
Malicious	0 %	0.4 %	1.7 %	83.9 %	0 %	14 %

Table 4: Detailed classification results of Alligator for samples collected after June 14, 2014

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.

Crawling: Number of samples crawled	124,083
Pre-filtering	
Duplicates	76,166
No SMS, no Internet	6,530
Known malware	423
Total number of samples pruned	83,119
Property extraction and classification	
Damaged samples	330
Suspects	65
Confirmed new malware or PUA (since July 2014 - note others were identified with SherlockDroid before that date and are not listed here)	5

Table 5: Details of samples processed by SherlockDroid during the July 2014 campaign

8 Conclusion - Future Work

Spotting really new Android malware in the wild is particularly difficult given the size and number of marketplaces. We therefore designed an automated framework SherlockDroid to assist researchers in finding them with marketplace crawlers, property extractors and a classification engine.

We found two new malware and six potentially unwanted applications by crawling over 120,000 applications within five different marketplaces. Though, to be fair, those samples are not among the most malicious or advanced malware, all of them do expose a more or less reprehensible behavior and were unknown to the Anti Virus community before SherlockDroid spotted them. Compared to prior research work, this is SherlockDroid's main achievement: we are not aware of any other system spotting unknown malware in the wild, apart from DroidRanger which detected two new families. Numerous research projects have only been tested on a few *known malware* or *artificial malware*. Working on known malware is easier because we know what we are looking for. As for artificial malware, they are created to test the system and hence raise ethical issues (how to ensure the malware does not spread), do not necessarily reflect malware in the wild and, in addition, are implemented by the same people who design the detection framework, hence introducing a serious bias.

We plan several improvements for SherlockDroid. In particular, we need to add other marketplace crawlers for a better coverage of Android applications. As for property extraction, we contemplate the use of contextual information in correlation of each property. Contextual information could be data like the call stack of the property. This would help us differentiate benign from malicious cases. For example, if we consider the "send email" property, it is quite different in terms of analysis if that email is sent to report bugs, or if it sneaks out information to a C&C. We would also like to work on differences between PUA and malware. This is difficult because there is no obvious property to tell the difference, and perhaps Alligator could help by introducing multi-class classification: a cluster for clean samples, a cluster for PUA and a cluster for malware.

Finally, we look forward to running SherlockDroid full time on operational servers and thus investigate bottlenecks and performance issues.

Acknowledgements

We wish to thank Ruchna Nigam, for her help on SherlockDroid.

References

1. Akdeniz: Google play crawler java api. https://github.com/Akdeniz/google-play-crawler
2. et al., M.L.: Andrubis - 1,000,000 apps later: A view on current android malware behaviors. In: Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS) (2014)
3. Apvrille, A., Strazzere, T.: Reducing the Window of Opportunity for Android Malware. Gotta catch'em all. In: Journal in Computer Virology, vol. 8, pp. 61–71 (2012)
4. Apvrille, L.: Alligator: AnaLyzing maLware wIth partitioninG and probAbiliTy-based algORithms (2014). http://http://perso.telecom-paristech.fr/~apvrille/alligator.html
5. Apvrille, L., Apvrille, A.: Pre-filtering Mobile Malware with Heuristic Techniques. In: GreHack, pp. 43–59 (2013). Grenoble, France
6. Arp, Daniel and Spreitzenbarth, Michael and Habner, Malte and Gascon, Hugo and Rieck, Konrad: Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket. In: Proceedings of the 17th Network and Distributed System Security Symposium (NDSS) (2014)
7. Aung, Z., Zaw, W.: "permission-based android malware detection". International Journal of Scientific and Technology reseach **2** (2013)
8. Bläsing, T., Schmidt, A.D., Batyuk, L., Camtepe, S.A., Albayrak, S.: An Android Application Sandbox System for Suspicious Software Detection. In: 5th International Conference on Malicious and Unwanted Software (MALWARE'2010). Nancy, France, (2010)
9. Book, T., Pridgen, A., Wallach, D.S.: Longitudinal analysis of android ad library permissions. CoRR **abs/1303.0857** (2013)
10. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: 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, pp. 15–26. ACM, New York, NY, USA (2011)
11. Chakradeo, S., Reaves, B., Traynor, P., Enck, W.: MAST: Triage for Market-scale Mobile Malware Analysis. In: Proc. 6th WiSec (2013)
12. Chang, C.C., Lin, C.J.: LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology **2**, 27:1–27:27 (2011). Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm
13. Cohen, F.: Computer viruses - theory and experiments. Computer and Security **6**, 22–35 (1987)
14. Dini, G., Martinelli, F., Saracino, A., Sgandurra, D.: 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, *Lecture Notes in Computer Science*, vol. 7531, pp. 240–253. Springer, St. Petersburg, Russia (2012)

15. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: 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, pp. 1–6. USENIX Association, Berkeley, CA, USA (2010). URL http://dl.acm.org/citation.cfm?id=1924943.1924971
16. Fortiguard Center: Android/RuSMS.AO (2013). Fortiguard Encyclopedia, http://www.fortiguard.com/encyclopedia/virus/#id=5897642
17. Harley, D., Lee, A.: Heuristic Analysis - Detecting Unknown Viruses (2007)
18. INTERPOL, Lab, K.: 60% of android attacks use financial malware. http://www.kaspersky.com/about/news/virus/2014/sixty-per-cent-of-An\-droid-attacks-use-financial-malware
19. Kose, N., Apvrille, L., Dugelay, J.L.: Facial Makeup Detection Technique Based on Texture and Shape Analysis . In: 11th IEEE International Conference on Automatic Face and Gesture Recognition (FG 2015) (2015)
20. Lindorfer, M., Kolbitsch, C., Milani Comparetti, P.: Detecting environment-sensitive malware. In: Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection, RAID'11, pp. 338–357. Springer-Verlag, Berlin, Heidelberg (2011). DOI 10.1007/978-3-642-23644-0_18. URL http://dx.doi.org/10.1007/978-3-642-23644-0_18
21. Lindorfer, M.e.a.: AndRadar: fast discovery of android applications in alternative markets. In: Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) (2014)
22. Mills, E.: Users upset after CA anti-virus detects Windows system file as virus (2009). http://www.cnet.com/news/users-upset-after-ca-anti-virus-detects\--windows-system-file-as-virus/
23. Patrick Schulz: Dalvik Bytecode Obfuscation on Android (2012). http://www.dexlabs.org/blog/bytecode-obfuscation
24. de Pontevès, K., Apvrille, A.: Analysis of android in-app advertisement kits. In: The 23rd Virus Bulletin International Conference, pp. 157–162 (2013)
25. Popa, B.: AVG Anti-Virus Breaks Down Windows XP Due To False Positive (2013). http://news.softpedia.com/news/AVG-Anti-Virus-Breaks-Down-Windows-XP-Due-\-to-False-Positive-337395.shtml
26. Rastogi, V., Chen, Y., Enck, W.: Appsplayground: Automatic security analysis of smartphone applications. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy, CODASPY '13, pp. 209–220. ACM, New York, NY, USA (2013)
27. Reina, A., Fattori, A., Cavallaro, L.: A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: Proceedings of the 6th European Workshop on System Security (EUROSEC 2013). Prague, Czech Republic (2013)
28. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P.G., Maranon, G.A.: Puma: Permission usage to detect malware in android. In: A. Herrero, V. Snasel, A. Abraham, I. Zelinka, B. Baruque, H. Quintian-Pardo, J.L. Calvo-Rolle, J. Sedano, E. Corchado (eds.) CISIS/ICEUTE/SOCO Special Sessions, *Advances in Intelligent Systems and Computing*, vol. 189, pp. 289–298. Springer (2012). URL http://dblp.uni-trier.de/db/conf/softcomp/soco2012s.html#SanzSLUBA12
29. Schapire, R.E., Singer, Y.: Improved boosting algorithms using confidence-rated predictions. In: Machine Learning, pp. 80–91 (1999)
30. Seltzer, L.: Lessons of the McAfee False Positive Fiasco (2010). http://securitywatch.pcmag.com/malware/283982-lessons-of-the-mcafee-false-posi\-tive-fiasco
31. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.: "andromaly": a behavioral malware detection framework for android devices. J. Intell. Inf. Syst. **38**(1), 161–190 (2012). DOI 10.1007/s10844-010-0148-x. URL http://dx.doi.org/10.1007/s10844-010-0148-x
32. Soergel, D.: Efficient training of support vector machines in java (2014). https://github.com/davidsoergel/jlibsvm
33. Viennot, N., Garcia, E., Nieh, J.: A measurement study of google play. In: The 2014 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '14, pp. 221–233. ACM, New York, NY, USA (2014)
34. Xie, L., Zhang, X., Seifert, J.P., Zhu, S.: pBMDS: a behavior-based malware detection system for cellphone devices. In: Proceedings of the third ACM conference on Wireless network security, WiSec '10, pp. 37–48. ACM, New York, NY, USA (2010)
35. Yan, L.K., Yin, H.: Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In: USENIX Security Symposium, pp. 569–584 (2012)
36. Zhao, M., Zhang, T., Ge, F., Yuan, Z.: Robotdroid: A lightweight malware detection framework on smartphones. Journal of Networks **7**(4) (2012). URL http://ojs.academypublisher.com/index.php/jnw/article/view/jnw0704715722
37. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: 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 (2012)