Pre-filtering Mobile Malware with Heuristic Techniques

Ludovic Apvrille[1] and Axelle Apvrille[2]

[1] Institut Mines-Telecom, Telecom ParisTech, LTCI CNRS,
Campus SophiaTech, 450 route des Chappes, 06410 Biot, France
ludovic.apvrille@telecom-paristech.fr

[2] Fortinet, Fortiguard Labs
120, rue Albert Caquot, 06410 Biot, France
aapvrille@fortinet.com

Abstract. With huge amounts of new Android applications released every day, in dozens of different marketplaces, Android malware unfortunately have no difficulty to sneak in and silently spread, and put a high pressure on antivirus teams. To try and spot them more easily, 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 is made of marketplace crawlers, code-level property extractors and a data mining software which decides whether the sample looks malicious or not. This data mining part is named *Alligator*, and is the main focus of the paper. Alligator classifies samples using clustering techniques. It first relies on a learning phase that determines the intermediate scores to apply to clustering algorithms of Alligator. Second, an operational phase classifies new samples using previously selected algorithms and scores. Alligator has been trained over an extensive set of both genuine Android applications and known malware. Then, it was tested for proactiveness, over new and more recent applications. The results are very encouraging and demonstrate the efficiency of this first heuristics engine for efficiently pre-filtering Android malware.

Keywords: Malware, Mobile phone, Android, Heuristics, Cluster, Filter;

1 Introduction

Once scarce - perhaps even spooky - malicious Android applications have now become a sad reality. There are over 200,000 malicious Android samples in June 2013, and approximately *1,000 new samples are reported every single day* [2]. The difficulty lies in spotting them in the middle of the enormous bunch of legitimate applications: 800,000 applications for Google Play alone [13]. The experience just sounds like finding a needle in a haystack...

Currently, Android malware are only sporadically found in marketplaces when researchers/engineers manually seek given applications, or by customer

input and malware exchange sharing with other anti-virus vendors (see left part of Figure 1). This means that most malware remain undetected in marketplaces for a long time. A possibility to automate the process is first to scan all applications by an anti-virus engine so as to quickly detect known malware and their variants. But then, there is still a huge volume of remaining samples (clean or undetected malware) that are impossible to manually analyze by anti-virus teams.
A way to break down that huge volume into a *much* smaller subset is to rely on automated tools to pre-filter Android applications in marketplaces. In practice, of course, the subset will probably also contain a few clean samples and a few variants of known samples which were undetected by the AV scanner. Those samples are finally inspected by AV analysts or researchers (see right part of Figure 1) and threats can be raised if necessary (hot bulletins, advisories, conference papers, ...).
Since the subset is manually inspected, it is consequently very important to keep its size down. A tight filter results in ignoring some malware. A loose filter results in having far too many samples to analyze, which actually also equals to ignoring plenty of malware (those AV teams don't have time to analyze). As the very idea of filtering is about setting priorities on samples, we prefer the lesser of two evils: **tight filtering**.

At EICAR 2012, an initial pre-filterering framework was presented [4], to start tackling this issue. It consisted of a Google Play crawler and a heuristics engine based on various properties found in the sample (presence of code sending an SMS, use of rooting exploits etc). Each property was given an empirical weight, and at the end an empirical threshold was set to filter in or out. Although empirical decisions are suitable for early prototypes, they quickly reach their limits on full scale systems. This paper enhances the framework with *data mining*. Actually, our contribution is twofold.

1. First, the design and implementation of **clustering techniques specifically adapted to populations of mobile malware**. We rely on state of the art clustering algorithms and adapt them to the filtering framework. Notably, we also design an original learning phase which automatically learns the best configuration. Contrary to usual learning phases, ours combines parameters for several clustering algorithms, not a single one (sections 3 and 4).
2. Second, the evaluation of our overall approach over **large sets of Android applications**.

The paper is organized as follows. First, we explain the overall architecture of our pre-filtering system (section 2), christened SherlockDroid after its ability to inspect clues. SherlockDroid can be seen as an improved and more mature version of the heuristics engine published in [4]. Then, we explain how we designed and integrated a free software clustering system for Android sample pre-filtering. This clustering tool is named *Alligator* [5]. It has been specifically tuned for SherlockDroid, though it could be re-used for other purposes. We quickly present the clustering algorithms it supports (section 3), and then explain its learning

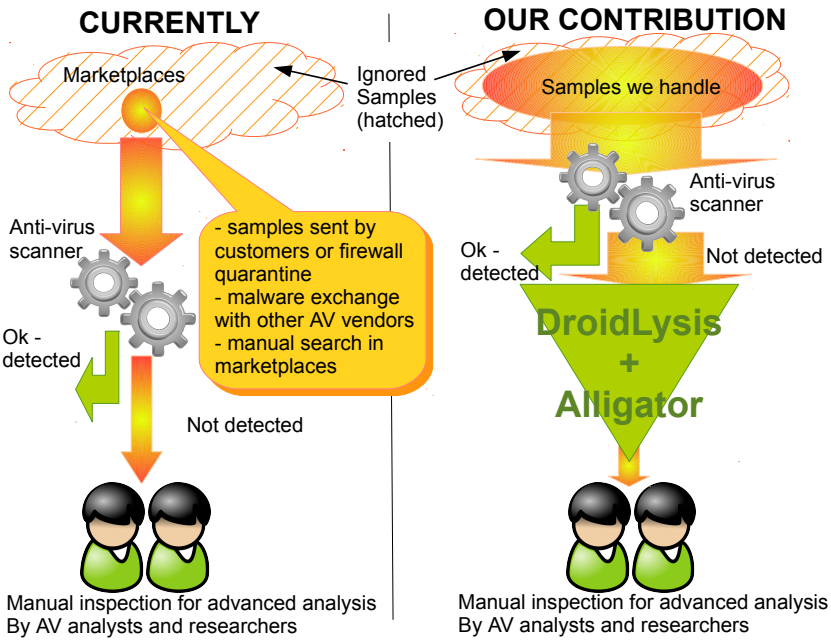

Fig. 1. Pre-filtering we need

phase (section 4), which has the specificity of *combining* multiple clustering algorithms. Then, we present Alligator's results over a wide-set of recent samples (section 5). We explain the differences with other related work in Section 6 and finally conclude the paper.

2 Architecture of SherlockDroid

Our pre-filtering engine, *SherlockDroid*, consists in three main elements. They are depicted at Figure 2.

– **Downloading** (left part of the figure). Samples are downloaded from different Android marketplaces. In particular, we have implemented crawlers for Google Play, ApkTop, SlideME and several less known repositories. Most marketplaces require the development of their own dedicated crawler, with sometimes tricky implementations (e.g. *Google Play* crawler as discussed in [4]). In some cases, we manage to use generic recursive crawlers for some marketplaces. Once a sample is downloaded, it is scanned by an anti-virus engine, to detect known malware. Undetected samples are stored in a database and queued for analysis with DroidLysis.

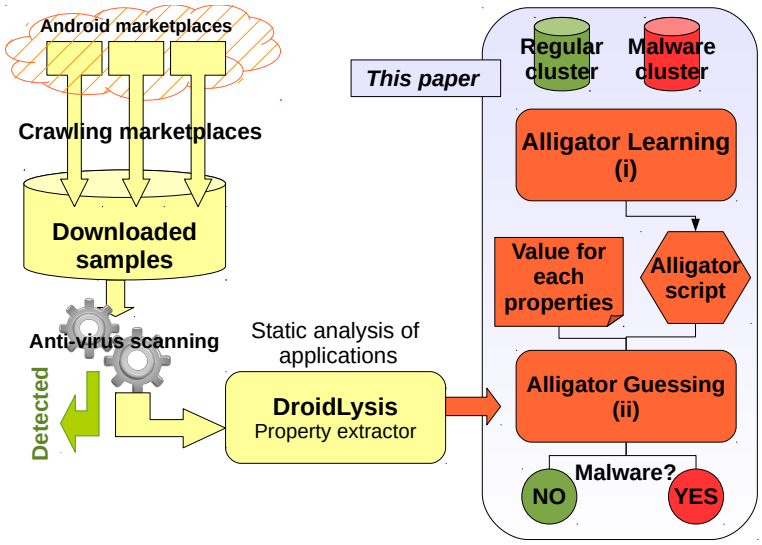

Fig. 2. SherlockDroid's Architecture. This paper mostly focuses on Alligator (right box): its design, its implementation, and its evaluation

- **DroidLysis** (middle part of the figure) is a stand-alone, quick, static analysis property extractor script. It has been presented in [4] but since then we have enhanced it to extract over 140 properties of Android applications. Finding malware is a cat and mice game, so the exact list of which properties are extracted and how cannot be publicly disclosed. Furthermore, other researchers wanting to use this system might want to implement other properties customized to their own needs. However, for the purpose of illustration, some properties have been described at [4] and since then, we have enhanced DroidLysis with more properties such as whether the app uses sockets or not, uses the keyguard functionality, busybox, JSON objects, bookmarks, cookies, aborts broadcasts of SMS, lists packages, contacts etc. Each property is either present or not present. A sample therefore corresponds to a set of boolean values (one for each extracted property), and a cluster is a file containing several sets of boolean values. This file is fed as input to the next module, *Alligator*.
- **Alligator** [5] (right part of the figure) is meant to decide which samples are the most likely to be malicious, based on clustering data mining approaches. It implements two phases: (i) a *learning phase* and (ii) a *guessing phase*.
  In the learning phase, Alligator automatically selects the best combination of weights to apply to clustering algorithms, in order to better identify malware and to reduce false positives. This combination is computed from applications already classified (regular, malware). Selected algorithms and weights are described in a file, written in an ad hoc Alligator script language.
  The guessing phase applies the script generated during the learning phase

onto the list of samples to partition (called *guess cluster*). Alligator runs each clustering algorithms indicated in the script, with the appropriate weights, and outputs for each sample two scores: a score of resemblance to regular samples, and a score of resemblance to malware samples. The higher the malicious is, the more malicious the sample is. If the malicious score is higher than the regular score, the sample is declared as suspicious. It ends up for manual analysis (see Figure 1).

The rest of the paper is dedicated to Alligator, and the overall results of SherlockDroid.

3 Clustering techniques of Alligator

Alligator is a lightweight, highly focused clustering tool, implemented in a few thousands of Java code lines. It is piloted by easily understandable scripts. Its comparison with other clustering tools is provided in section 6.

3.1 Classifying guess samples

Alligator takes as assumption that two main clusters R and M have been already settled with "known" elements. In our case, cluster R holds regular samples (legitimate) and M holds malicious ones. The main purpose of Alligator is to classify unclassified samples, i.e. samples currently part of a cluster named the "guess" cluster G, into either R or M. Each element - also called *samples* - of R, M and G have been given a value for each property p of a set of properties $\mathbb{P}$. The decision to put a sample $g \in G$ in R or M thus depends on the value of each $p \in \mathbb{P}$ for g, and also depends on the various values of samples of R and M for the same set $\mathbb{P}$ of properties.
Finally, a clustering algorithm takes as input a sample g of G, two clusters R and M and returns the score of g for each cluster R and M.

3.2 Clustering algorithms

We now quickly review the clustering algorithms implemented in Alligator for the guessing phase. Some of them are quite well known (e.g., deviation, correlation), therefore, our focus is rather to discuss the main reasons why we think they could be interesting in the scope of malware identification (see also section 3.3).

- **Standard deviation**. Well known metric for computing the distance between a given sample $g \in G$ and the average of cluster R and M for each property. The smallest this distance is with a given cluster center, the better is the chance to be part of that cluster.

– **Probability difference**. The score is based on the percentage of "typical" values that are respected in R and M. "Typical" is defined by a given probability difference *diff*: $proba_1 > diff + proba_2$. For example, let's assume that the probability of property #6: "send_SMS" to be equal to "1" is 0.8 in cluster M and equal to 0.2 in cluster R, and the probability difference $diff = 0.5$: We say that "1" is a typical value of cluster M for property "send_SMS" because $0.8 > 0.2 + 0.5$. Thus, if the value of property 6 of g is $= 1$, then g "respects" one typical value of of M.
Finally, the more typical values g respects for a cluster, the more chance g has to be part of it.

– **Probability factor**. Same as "Probabiblity difference" except typical values are computed using a multiplicative difference: *fact*: $proba_1 > fact * proba_2$.

– **Proximity**, aka k-Nearest Neighbours (k-NN). Based on the usual distance relation, the algorithm takes as input the number of n of nearest neighbours of g. Then, the score of g for R (resp. M) corresponds to the percentage of elements of R (resp. M) that are in this set of closest neighbours. Thus, contrary to the standard deviation that targets the distance with the centers of the two R and M clusters, the proximity cares only with the closest elements of each clusters R and M.

– **Proximity with limited properties**. Same as "Proximity", but we only consider the smallest distances. Thus, two samples which are very close - if not equal - on most properties, but very far with only one property, are classified as very distant with "proximity", but very close with "proximity with limited properties".

– **Correlation**. Alligator first identifies correlations between different properties within a given cluster. For instance, Alligator could show that properties "send_SMS" and "Internet access" are correlated for malware, but not for regular. Then, Alligator computes for each g a score according to correlations g matches within M and R.

– **Epsilon clusters**. In epsilon clusters, samples are grouped according to a minimal distance ϵ. A sample s is added to a given epsilon cluster if and only if there exists one sample in this cluster so that the distance between that sample and s is less than ϵ (see Figure 3). Epsilon clusters are useful to create sub-groups of R and M for each g of the guess cluster, and then figure out the proportion of elements of R and M in that group. Thus, the more elements of R (resp. M) in the sub-group of g, the more chance has g to be an element of R (resp. M).

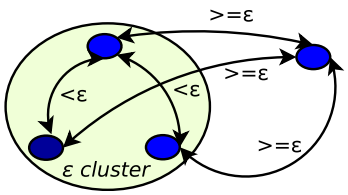

Fig. 3. Notion of *epsilon cluster*. The epsilon cluster of this example contains three elements. Another element is not in that cluster since its distance with any element of the cluster is greater or equal to epsilon

3.3 Discussion on clustering algorithms

Clustering algorithms of Alligator provide different techniques for classifying samples. Algorithms have been selected for their variety in distances: sometimes, it is based on the *center* of clusters (e.g., deviation, probability), or on the *neighbourhood* (e.g., proximity, epsilon cluster). Algorithms also differ in their criteria for computing distances (probabilities, ϵ-path, etc.). But the selection of which algorithms are really relevant for efficiently classifying a set of samples is however not an easy task. By "selection", we mean which importance (or *weight*) should we give to each clustering algorithm. A zero-weight means that the algorithm is useless, a weight greater than zero means that the algorithm is relevant. Two approaches can be used to determine weights:

- Review contributions on malware classification and results on clustering techniques, and empirically define weights for each algorithm.
- Define an automatic weight computation for each algorithm. This is the option we have taken: we have defined and implemented a learning phase in Alligator. This phase is more thoroughly explained in next section.

4 The learning phase of Alligator

As we said previously, selecting clustering algorithms is not so easy in practice. So, to overcome this difficulty, Alligator offers an easy-to-use *learning phase* where a human-readable learning script helps selecting the best combination of algorithms to use.

4.1 Learning phase: general approach

The learning phase works as follows.

1. Already classified samples are put either in cluster R or M. Other samples are put in G.
2. A set S of algorithms - and their respective parameters - is selected. For example, "probability difference with $diff = 0.8$", "proximity of 25 samples limited to 50 properties", etc.

3. The Alligator learning phase determines - using *learning algorithms* - the best *weight* to give to each set of algorithm/parameters so as to maximize the samples of R and M that are classified in their corresponding cluster when applying algorithms/parameters of S. Said differently, the scoring for a sample of R should be higher for cluster R than for cluster M, and the opposite for samples of M.

Once the learning phase has determined all weights of algorithms, then, Alligator can be used in its operational phase (i.e., the guessing mode) in order to classify samples of G.

4.2 Goal of learning algorithms

In Alligator learning, a weight is cut into two parts:

- A user-defined float expression *expr* that takes as argument a float and returns a new float. Usual operations ($+$, $-$, $*$, $/$) and parenthesis can be used to define an expression *expr*.
- A multiplier m to be applied on the result given by the user-defined expression.

Thus, $weight(x) = expr(x) * m$.
Alligator's learning phase intends to automatically determine m. Since one weight is defined for each algorithm and for each cluster (regular, malware), the range of possible m values is large. The learning script can be used to restrict all possible intervals of m for each clustering algorithm, and for each cluster R and M. For example, the following subscript configures the possible values of m for the score on standard deviation: $expr = x^2$ is provided for each cluster, and a range for $m = [0 \dots 1000]$ is provided also for each cluster R and M, with a step of 5. This range means that all values in interval 0..1000 with a step of 5 shall be considered by Alligator in learning mode for that specific algorithm

```
setComplexWeight regular 0-1000,5 x*x
setComplexWeight malware 0-1000,5 x*x
compute deviation
```

The goal of learning algorithms is to determine the best sets of m for each algorithm, for each cluster. "Best" means that we try to optimize the percentage of elements of R and M that are correctly classified with the set of multipliers m. For example, in the case of the previous script, Alligator determined that $m = 10$ for regular, and $m = 350$ for malware. In that case, the learning phase would generate the following script, to be used in guessing mode:

```
setComplexWeight regular 10 x*x
setComplexWeight malware 350 x*x
compute deviation
```

4.3 Learning algorithms

We now explain the basics of learning algorithms meant to determine the multipliers m.

- **Random** tests random multipliers within their possible ranges, and for a given time.

- **BruteForce** parses all multipliers combinations. This algorithm is to be used when the set of all combinations is of reasonable size. We have implemented this algorithm in two different ways. In the first one, a tree of all combinations is first built: each leaf represents one possible combination. Then, each leaf is considered one after the other. The second implementation considers various combinations while the tree is being built and destroyed ("on-the-fly" approach).

- **OneOtherAverage**. Of all multipliers which have a range specified, the algorithm starts working on the first one, and sets all other ranges to their average value. When the algorithm has found the best value for the first variation, it memorizes the value. Then, it works on the second range. The first range is set to the best value, all others are set to average. And so on. This algorithm has the drawback of not taking into account the correlation between multipliers, that is, for a given algorithm, only the multiplier of R or M is explored at a time.

- **TwoOtherAverage**. Same as *oneOtherAverage* except the algorithm works simultaneously on two multiplier ranges at a time. The others are set to the average value, or to the best value that was previously identified. Since the combinations are much larger than in *oneOtherAverage*, this algorithm takes much longer to complete. But it can explore at the same time the best multipliers for R and M for a given clustering algorithm.

- **Combination of learning algorithms**. *random* and *oneOtherAverage* (or *twoOtherAverage*) can easily be combined. For example *random* is first used for a given duration to define a first set of best values for each multiplier. Then, *oneOtherAverage* is run using as starting values these best values, instead of using average values. Other combinations are obviously possible. In particular, *OneOtherAverage* can be iterated several times until identified multipliers do not improve anymore the classification results.

4.4 Minimizing false positive and/or negative

In our case, we are particularly concerned that Alligator does not flag clean files as suspicious, as then, anti-virus analysts would manually lose their time inspecting them. Flagging a malicious file as clean is obviously not desirable, but not as critical. As said in the introduction, the main idea of our contribution

is indeed to create a small subsets of applications to be manually analyzed. In other words, false positives are more important to us than false negatives. For Alligator, this corresponds to assigning a higher weight to correct identification within R than M. Thus, weights selected by learning algorithms can be computed by also giving more importance to the right identification of clean samples or malware. Two parameters can be used for that purpose:

- **A relative factor f of correct identification between regular and malware**.
  For example, suppose we obtained correct identification percentages of 99% for R and 94% for M, and in another learning, 98% and 97%. If the factor $f = 1$, regular and malware identification has the same importance, and so, the second learning is selected because the average correct identification is 97.5%. On the contrary, if $f = 10$, the first learning is selected since its average identification is $\frac{10*99+94}{11} = 98.54$ whereas the average identification of the second learning is only equal to 97.64.
- **A minimum correct identification rate**: one rate for regular, one rate for malware. Between two sets of weights, with one respecting the minimum correct identification rates, and another one not respecting them but respecting the relative factor, the first one is prior.

4.5 Discussion

The learning phase is of utmost importance for correctly setting the weight of each clustering algorithm and of each cluster R and M. The execution of this phase takes a reasonable time when ranges and steps are well chosen. Large ranges may first be selected with large steps, and then, manual selection of smaller ranges and steps around the found values is a way to more quickly converge to good weight values. Other optimization techniques (e.g., Pareto approaches) could be used to select weights. However, current approach is quite simple and offers good results, both in term of learning time and of percentage of correct classification. The next section focuses more particularly on learning results.

5 Results

The efficiency of SherlockDroid - including Alligator learning and guessing phases - is evaluated below with recent Android applications. Efficiency is evaluated in terms of quickness to produce results, and in terms of relevance of the results. Both points are discussed hereafter.

5.1 Composition of test clusters

Alligator has been trained over real-life clusters of **83,119** malicious Android samples and **8,505** clean ones (see Table 1). Those samples were downloaded

before June 14, 2013. The guess clusters, with samples different from the learning phase, have 19,185 malicious samples and 1,104 regular ones. They were downloaded end of June 2013.

Type of cluster	Malware samples	Regular samples
Learning clusters	82,985	8,299
Guess clusters	19,171	1,103
Total of samples tested	102,156	9,402

Table 1. Number of samples in our test clusters

Gathering *clean* samples is a difficult problem, because it is difficult to be absolutely sure a sample is not malicious. We cannot trust an application to be genuine because it appears on Google Play (as a matter of fact, several malware are encountered on that marketplace, and others). We need to inspect code manually, and check there is no malicious functionality, which unfortunately is a lengthy process. As a compromise, we also populated the set with open source applications (e.g. Mozilla, Savannah Gnu, F-Droid), as their source code can be inspected by the community, thus with reduced risks of being malicious. We also added signed packaged included in the Android operating system (e.g *Gmail.apk*, *GoogleServicesFramework.apk* etc.) and standard applications that ship with the ROM of mobile phones[3]. So, although it would be preferable to have clusters of approximately the same size, this was not feasible in practice. This is however a point we will focus on in our future work.

5.2 Results of the learning phase

We tested Alligator's learning over our different learning algorithms (*brute-force*, *oneOtherAverage*, *twoOtherAverage*, *random* and *combinations / iterations*) with several parameters for each classification algorithms, e.g.:

- Proximity: closest 50, 10, 2 and 1 neighbour(s).
- Correlations: 0.80, 0.75, 0.70, 0.60
- Probability difference: 0.5, 0.2, 0.1
- Probability factor: 10, 5, 2, 1.5, 1.2
- Epsilon clusters: ϵ-path of 10^{-5} to $10^{-}1$

Then, the learning phase outputs a score to use for each (algorithm, parameter) and for each type (clean, malware). For example, the score to apply to $(correlation, 0.80)$ is 414 for clean and 918 for malware.
In terms of computation time, with the "*randomonemultipass* 600 20" option (i.e., random for 600 seconds, and then 20 pass "oneOtherAverage"), the learning phase took around *14 hours* on an average non dedicated host.

[3] Note there has however been isolated cases of infected firmware apps - recall CarrierIQ [11]

5.3 Results of the guessing phase

In a second step, we gathered other samples, different from the ones we had used for training, to be fair for results. The new malicious samples are more recent, collected between June 15th and June 24th 2013. We removed any malicious sample which was detected by a generic signature existing before June 15, 2013. So, the set of new malicious samples contains "new unknown" malware at the time of Alligator's learning. Using the best scripts generated by Alligator during the learning phase, **we tested Alligator over those *new* sets of malware and clean files**.

The results of Alligator guesses are displayed at Table 3.

		Regular		Malware	
Learning	Number of failed/recognized	9 / 8,290		67 / 82,918	
	Failure/success rates in %	0.11%	99.89%	0.08%	99.92%
Guessing	Number of failed/recognized	2 / 1,101		375 / 18,796	
	Failure/success rates in %	0.18%	99.81%	1.96%	98.04%

Table 3. Failure rates for Alligator's learning and guessing phases. For instance, we have a very low rate of FP - which is our main target -, and a 98.04% pro-activity.

On one hand, the failure rate for clean samples is really excellent: this was our main target to reduce that rate since FP may induce extra work to AV analysts. On the other hand, the failure rate for malware samples is also very low, which demonstrates an interesting capability in terms of pro-activity. Alligator was able to flag 98.04% of malware, malware which were unknown (and undetected) to an anti-virus scanner.

5.4 Discussion

We believe the results of Alligator are really excellent, and list below our arguments.

- **Our premier goal of reducing the sheer amount of samples to inspect is taking shape**. Using the current configuration, our pre-filtering framework saves analysts from many unnecessary analysis. If 10,000 clean samples and 1,000 undetected malware go through our pre-filtering system, only $10,000 * 0.18\% + 1,000 * 98.04\% = 1,002$ out of 11,000 remain to be analyzed manually in the end, with only 20 malware remaining undetected (and so, 980 malware are detected).
- **Alligator shows a high proactivity rate 98.04%**. Pro-activity is the capability of identifying new families of malware. Recall section 1: this is

a second goal to our pre-filtering system, as it helps researchers look into new trends in cyber-criminal worlds. As an element of comparison, VB100 tests show pro-activity rates between 70% and 90% for anti-virus products [16]. Although our pre-filtering system is not an anti-virus engine, the figures indicate Alligator is performing well in this area, even on large clusters.

– **False positives are unacceptable for anti-virus scanners (complaints, bad press...)**. However, they are an unavoidable burden for any system relying on heuristics (such as DroidLysis). We have already said this previously but it is important to repeat Alligator is not an anti-virus scanner: it acts after samples have been scanned. Of course, future work will however be made to minimize as much as possible failure rates for clean samples, even if we already consider our 0.18% rate as very low.

6 Related Work

Data mining's goal is to organize large and complex sets of data. In the anti-virus industry, it has often been used to classify PC samples [17] [18] [8], or to combine weak heuristics into stronger rules [12]. However, its use over *mobile* samples is far more recent, perhaps because the growth of mobile malware is quite new altogether. We are aware of 4 prototypes which use data mining in the mobile world: MADAM [10], [15], AAS [6] and Crowdroid [7].

MADAM, unlike SherlockDroid, is a behaviour-based detection engine running directly on an Android phone. It relies on the *k-NearestNeighbour* algorithm for data mining, similar to the proximity of Alligator. MADAM seems promising for a run time detection, but it obviously requires to manually install the application and use it, which prevents from automating the process.

[6] is an Android Application Sandbox to be deployed inside the marketplace itself and to pre-scan applications. Their paper explains how to collect information, notably how to hijack and log system calls, but does not discuss how the final decision - suspicious or not - is to be made. Moreover, AAS has only been tested against 150 clean applications and a single self-written fork bomb.

Crowdroid [7] is an interesting effort to dynamically monitor Android applications. A client application is deployed on the smartphones and sends pre-processed system calls data to a remote server for data mining. Their approach is significantly different from ours on several aspects. First, they send data for analysis over the Internet which opens concerns for privacy even if data is "non-personal". The use of Internet may also cost to the end-user, depending on his mobile subscription. Second, their system relies on dynamic analysis, whereas we have chosen to rely on static analysis only. Without entering the debate, we have chosen the static approach for scalability and efficiency reasons: scalability as we need our system to process thousands of applications. Efficiency because malware commonly expose a different behaviour when run inside emulators or sandboxes [1]. With regards to scalability, it should be noted that Crowdroid has only yet been tested on artificial malware (i.e tailored for the tests) and on a handful of real malware.

Also related to our research, we could cite pBMDS [19] and Andromaly [14] which are behaviour-based detection engines, like MADAM. But they do not use data mining.

One of the closest work to ours is DroidRanger [20]. Like SherlockDroid, DroidRanger is one of the few systems which scale and has been tested on a significant amount of real life applications (recall for instance that AAS has been tested over 150 clean samples and 1 artificial malware, Crowdroid over 2 real malware and a few artificial ones etc). DroidRanger consists in filtering applications statically against pre-computed footprints of known malware. Then, in a second stage, it looks for signs of attempting to dynamically load untrusted code, using two different heuristics. So, basically, the tasks of DroidRanger corresponds to the DroidLysis block in our architecture, except that DroidRanger is limited to either detecting variants of *known malware* families or unknown malware that dynamically load untrusted code, via the implementation of two heuristics. To be fair, DroidRanger is quite successful at spotting malware in that particular subset, however, by design, it is blind to other kind of malware: malware that do not load untrusted code and are not from a known malware family cannot be spotted. Because of those limitations, DroidRanger does not need data mining. In the future, if they extend their system to consider a wider set of possibilities (Alligator processes results from 146 different heuristics) they will necessarily face issues as those addressed by Alligator in this paper.

Finally, from a data mining point of view, several generic data-mining approaches and tools already exist, e.g. [9]. They support many clustering techniques, but unfortunately suffer from several drawbacks with regards to the way we need to use them. First, they mostly target the identification of clusters: in our case, we want to classify samples into two known base clusters (R and M). Second, the classification usually relies on *one* given distance metric (e.g., Euclidian, Pearson correlation, etc.) where Alligator relies on *several* algorithms whose importance for correct identification is automatically computed in a learning phase. Thus, Alligator provides strong automated help to select clustering algorithms. Third, like e.g. [9], Alligator can be piloted by readable, simple scripts and text-based databases, and runnable on any kind of desktop host. Last but not least, Alligator has been implemented with an integrated understanding and minimization of false positives and false negatives. In the anti-virus industry, this is particularly important, as as explained in section 4.4, Alligator has an explicit criteria to favor lower false positives compared to false negatives.

7 Conclusion

With hundreds of new Android applications every day on marketplaces, mobile malware have the opportunity to be silently released, cause havoc and only be noticed several days (or months) later. This puts much pressure on anti-virus teams to rush and classify samples as quickly as possible. To do so, the paper proposes a pre-filtering system which automates analysis of applications published in Android markets. *SherlockDroid* efficiently combines an Android

application crawler, an Android application property extractor (*DroidLysis*) and a data-mining toolkit (*Alligator*), the latter being an important contribution of the paper.

Tests have been conducted over large sets of Android applications. Our pre-filtering system significantly reduces the work load for analysts: 99.8% of clean applications are filtered out. Alligator also seems very promising for the detection of new unknown malware, with a pro-activity rate measured at more than 98%. This gives researchers far better chances of identifying new families of Android malware, and thus warning the community. Alligator already identified unknown malware, e.g. a GPS-leaking adkit [3].

We will however improve our research on several angles. First, technically, we intend to explore new clustering and learning scripts, for instance, automatically refining searches or tightening the filter. We also intend to introduce *weights* on properties so that algorithms such as deviation do not consider each property with equal importance. Second, on a wide scale point of view, we need to find ways to collect more clean malware for training, so that the regular cluster has a comparable size to the malware cluster. Besides theory, we have also already started to test the entire SherlockDroid architecture over several dozens of marketplaces and intend to see in practice how many suspicious samples get flagged and how many new malicious families are discovered that way.

References

1. Axelle Apvrille. An OpenBTS GSM replication jail for mobile malware. In *21st Virus Bulletin International Conference*, pages 108–116, Barcelona, Spain, October 2011.
2. Axelle Apvrille. 1,000 malicious Android samples per day, May 2013. http://blog.fortinet.com/1-000-malicious-Android-samples-per-day.
3. Axelle Apvrille. Alligator detects GPS-leaking adware, August 2013. http://blog.fortinet.com/Alligator-detects-GPS-leaking-adware/.
4. 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.
5. Ludovic Apvrille. Alligator: AnaLyzing maLware wIth partitionin G and probAbiliTy-based algORithms, 2013. http://perso.telecom-paristech.fr/∼apvrille/alligator.html.
6. 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.
7. 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.
8. Jianyong Dai, Ratan Guha, and Joohan Lee. Feature set selection in data mining techniques for unknown virus detection: a comparison study. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research:*

Cyber Security and Information Intelligence Challenges and Strategies, CSIIRW '09, pages 56:1–56:4, New York, NY, USA, 2009. ACM.
9. J. L. de Hoon, S. Imoto, J. Nolan, and S. Miyano. Open Source Clustering Software. Feb 2004.
10. 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.
11. Trevor Eckhart. What is Carrier IQ?, 2011. http://androidsecuritytest.com/features/logs-and-services/loggers/carrieriq/.
12. Igor Muttik. Malware mining. In *21st Virus Bulletin International Conference*, pages 46–51, Barcelona, Spain, October 2011.
13. Google Play Store: 800,000 apps and overtake Apple AppStore!, February 2013. http://www.rssphone.com/google-play-store-800000-apps-and-overtake-apple-appstore.
14. 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.
15. Peter Teufl, Stefan Kraxberger, Clemens Orthacker, Günther Lackner, Michael Gissing, Alexander Marsalek, Johannes Leibetseder, and Oliver Prevenhueber. Android Market Analysis with Activation Patterns. In *Proceedings of the International ICST Conference on Security and Privacy in Mobile Information and Communication (MobiSec)*, 2011.
16. Virus Bulletin, Fighting malware and spam, VB100 Comparative review on Windows Server 2003 R2, October 2012.
17. Jau-Hwang WANG, Peter S. DENG, Yi-Shen FAN, Li-Jing JAW, and Yu-Ching LIU. Virus detection using data mining techniques. In *IEEE International Conference on Data Mining*, 2003.
18. Tzu-Yen Wang, Chin-Hsiung Wu, and Chu-Cheng Hsieh. A virus prevention model based on static analysis and data mining methods. In *Proceedings of the 2008 IEEE 8th International Conference on Computer and Information Technology Workshops*, CITWORKSHOPS '08, pages 288–293, Washington, DC, USA, 2008. IEEE Computer Society.
19. 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.
20. 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.