Identifying Unknown Android Malware with Feature Extractions and Classification Techniques

Ludovic Apvrille
Institut Mines-Telecom
Telecom ParisTech, CNRS/LTCI
Sophia Antipolis, France
Email: ludovic.apvrille@telecom-paristech.fr

Axelle Apvrille
Fortinet
FortiGuard Labs
Biot, France
Email: aapvrille@fortinet.com

***Abstract*—Android malware unfortunately have little difficulty to sneak in marketplaces. While known malware and their variants are nowadays quite well detected by anti-virus scanners, new unknown malware, which are fundamentally different from others (e.g. "0-day"), remain an issue.**

To discover such new malware, the SherlockDroid framework filters masses of applications and only keeps the most likely to be malicious for future inspection by anti-virus teams. Apart from crawling applications from marketplaces, SherlockDroid extracts code-level features, and then classifies unknown applications with Alligator. Alligator is a classification tool that efficiently and automatically combines several classification algorithms.

To demonstrate the efficiency of our approach, we have extracted properties and classified over 600,000 applications during two crawling campaigns in July 2014 and October 2014, with the detection of one new malware, Android/Odpa.A!tr.spy, and two new riskware. With other findings, this increases SherlockDroid's "Hall of Shame" to 9 totally unknown malware and potentially unwanted applications.

I. Introduction

With the plethora of monthly new Android applications (between 20,000 and 40,000 according to *AppBrain*), malware authors can easily sneak in malicious applications. Fortunately, several of these are detected by anti-virus products, which rely on malware 'signatures', or patterns that match several samples at a time [1]. Nevertheless, *all* anti-virus scanners face difficulties when it comes to detecting *truly new* malware (new families, 0-day...).

AV vendors tackle this issue by complementing their traditional scanners with heuristic engines, machine learning or sandboxes, that are used 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 (FP). Indeed, FPs (clean samples detected as malicious) is what AV vendors fear the most, because of the immediate surge in customer requests and bad press [2] [3]. Heuristic approaches and behavior analysis mechanisms are yet in their early days [4] [5] [6] and haven't been evaluated over large volumes of malware, as further explained in section II. So, in practice, there is currently no automated solution to identify those truly new and unknown Android malware.

To remedy this situation, our contribution consists in proposing a framework, named *SherlockDroid*, that automatically inspects Android marketplaces, and only outputs a handful of suspicious items. SherlockDroid is customized to be highly selective, so that it outputs only the topmost suspicious applications. Those samples then undergo manual analysis, so no need to output thousands of suspects: anti-virus analysts would not be manned to process such input. Additionally, it is important that SherlockDroid outputs as few FPs as possible so that analysts do not waste their precious time. With a tight filtering, it is true that SherlockDroid is bound to miss several new interesting malware. While this is not desirable, we argue that without SherlockDroid, even more malware remain undetected. At least, SherlockDroid improves the situation.

SherlockDroid relies on the automated combination of marketplace crawlers, filtering and feature extraction tools, and classifiers. It is meant to process large quantities 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 the feature extraction (named DroidLysis) and the specific classification engine (Alligator). Another contribution of the paper is the evaluation of our approach on a huge dataset (over 600,000 applications).

This paper is organized as follows. First, section II compares SherlockDroid with other similar heuristic-based approaches. Then, we describe SherlockDroid's architecture (section III), and detail contributions on feature extraction and classification issues in sections IV and V, respectively. Our large scale results and a discussion are provided at section VI. Section VII concludes the paper.

II. Related work

A. Assisting the discovery of unknown malware

Nearly all prior research work is tested on known malware datasets. They consequently confirm maliciousness of *known malware*, but do not spot *unknown malware like new families*. In some cases, it is simply not their main goal [7] [8] [9] [10] [11], but rather they focus on particular aspects, e.g., information leakage [9] [11], or target the identification of variants of known malware [12]. 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. [13] [4] [5] [14]). Some research work crafted artificial, i.e. self written malware [15] [16] [6]. 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 [17], although it hasn't discovered any new malware, is at least evaluated against unknown malware, where it obtains detection rates between 50 and 75%, which is far lower than with SherlockDroid (more than 98% - see section VI). AppsPlayground [18] does not detect new unknown malware either, but succeeds at pin-pointing undetected privacy exposure.

DroidRanger [19] *is the only work which actually led to the discovery of 2 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 only two heuristics. SherlockDroid relies on a far larger set of heuristics (see section IV), and leads to the discovery of 2 new malware and 7 potentially unwanted applications (PUA).

B. Features: extraction and relevance

Features of Android application are extracted either with a static analysis of the application package (e.g., [20]), or by dynamically analyzing the behavior at runtime, e.g., function calls [19]. MAST [20] statically extracts 182 features. MAST relies on *attribute-based* selection, that is, features extracted independently from other features, and on *subset-based* extraction that takes into account dependencies between features.

Several approaches mix both static and dynamic analysis, e.g., DroidRanger [19] and AMDetector [21]. DroidRanger [19] relies on a footprint-based detection engine that extracts features both from the manifest file (permissions) and from the semantics found in the bytecode (e.g., send SMS), and also on a heuristics-based detection engine that monitors applications during their execution, e.g., system calls with root privileges. AMDetector [19] is also based on a hybrid static/dynamic approach, and shows a True Positive rate of 88.14% and a False Positive rate of 1.80%. Yet, SherlockDroid/Alligator, which is only based on static analysis, offers a better accuracy in terms of False Positive / False Negative rates (see section VI). In our case, static analysis is fast and makes it easy to extract valuable information from Android application's file format. Reciprocally, we do not use dynamic analysis because we believe it would slow down performance (time to install the application, launch it, run a few commands etc). Indeed, we want to process thousands of applications and cannot afford more than a few seconds on each without creating a serious bottleneck. Additionally, the history of malware on Windows has shown that, sooner or later, malware authors implement techniques to behave differently when run in emulators or sandboxes (e.g. Win32/FakeAV in 2008 [22]).

C. Machine learning and classification

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

MAST [20] relies on correlations. [23] compares five different sets of features and five different classifiers (so, 25 combinations). MADAM [4] relies on the *k-NearestNeighbour* algorithm for classification, similar to the proximity of Alligator. Andromaly [16] 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, PUMA [24] 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: 92%). Yet, SherlockDroid is probably the only one that computes an efficient combination of classification algorithms, that is, a weight is given by the learning phase to all classification algorithms that alligator supports, e.g., *k-NearestNeighbour*, *correlations*, *SVM*, ..., leading to best recognition rates, as presented in the result section. Similarly, [23] does not show better result when it considers several classifiers individually (83% of best accuracy).

Other contributions, like RobotDroid [25], try to customize classification algorithms (e.g., SVM) in order to classify better limited sets of elements while still guarantee a small error. In our case, we do have large sets of data, but they are strongly unbalanced (few clean, many malware). Thus, Alligator has specific options to handle that issue, and as demonstrated by the results section (section VI), it is very efficient to handle large and unbalanced datasets.

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 [14]	k-Means
Feature Selection in Android [23]	Nave Bayes, k-NN, J48, MLP, RF
Alligator	Combination of any: standard deviation, several variants of k-NNs, correlations, probabilities, ϵ-clusters, SVM

TABLE I
CLASSIFICATION APPROACHES FOR ANDROID APPLICATIONS

III. OVERVIEW OF SHERLOCKDROID

A. Main components of SherlockDroid

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

Crawling - Reaching the crime scene. Crawlers download samples from various marketplaces. Currently, we have crawlers for the Play Store, APKTop, AppsApk, SlideME, Nduoa and a generic crawler which recursively parses a URL for Android applications. Crawlers are not detailed in this paper, but more information can be found in [26].

Analysis - Gathering clues. This layer is in charge of analyzing the sample. First, there is a pre-filtering substage, where samples which are not likely to be interesting to analyze (e.g. known malware) are pruned.

Second, the sample undergoes **feature extraction**. Compared to Sherlock Holmes, feature extraction corresponds to Sherlock Holmes inspecting the sample and methodically collecting as much information or evidence as possible for the case. Currently, information is extracted from the static analysis of 289 different features. This work is handled by a tool named *DroidLysis* - see Section IV.

Classification - Sniffing clues. 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 learning, Alligator sniffs the sample and tells the inspector whether the sample is suspicious or not - see section V.

B. SherlockDroid vs. an Anti-Virus scanner

SherlockDroid is not an anti-virus scanner. Indeed, 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 first scanned by an AV engine, and known malware are ignored by SherlockDroid. Also, SherlockDroid is not able to say for sure that a sample is clean or malicious. It only says a sample *looks clean or not* (result of the classification). Thus, SherlockDroid's design is rather comparable to *heuristics*, which often *complement* AV products [1].

IV. FEATURE EXTRACTION (DROIDLYSIS)

A. Feature definition and categories

SherlockDroid considers *features* of Android applications extracted from a static analysis of the code.

DroidLysis features fall in 4 different categories:

1) **File properties** (54/289). They correspond to characteristics found in important files of Android 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). Extracted from the Dalvik executable inside the Android package. There are properties which detect the use of particular APIs (e.g. `sendTextMessage`), actions (e.g. ACTION_CALL), intents (e.g. EXTRA_SUBJECT), constants (e.g. `POST`), implementation techniques (e.g. JNI), or Dalvik opcodes (e.g. nop, const-string). For example, we detect junk bytecode injection like [27].
3) **Resource properties** (22/289). Extracted from files in the package's resource, library or asset directories. They focus on 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 IV-D.

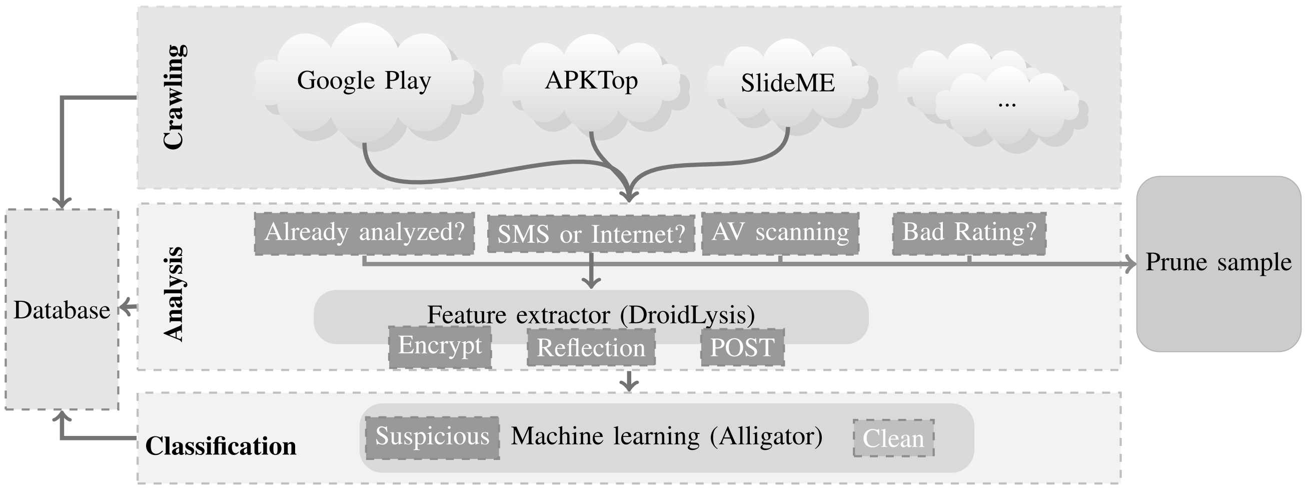

Fig. 1. SherlockDroid Architecture

B. Feature extraction

In DroidLysis, all features are extracted statically.

As much as possible and for ease of manipulation, we have chosen to extract features 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, this is a limit of static analysis. Dynamic analysis on its side has the drawback of not detecting some calls when code coverage is not complete.

Resource features 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 or decompile them.

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

C. Feature relevance

Based on our experience on Android malware reverse engineering, we identify the mechanisms used to conduct malicious deeds, and extract features around it. As an illustration, we now explain the mapping between mechanisms and features in 2 examples.

Mobile spyware are interested in personal assets, e.g. IMSI, visited URLs etc. We track such attempts via calls to `getSubscriberId()` for the IMSI, or `getAllVisitedUrls()` for URLs, and even `getBookmarks()` etc.

In several cases, e.g. Android/DrdDream, mobile trojans' grail is root access on the device. This can be done 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 Wi-Fi 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 features which are well tuned to SherlockDroid's goal, as demonstrated in the result section VI.

D. Handling advertisement kits

We separate features 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 features and an application using a third party kit that uses the same features.

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[28] 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.

V. Classification (Alligator)

A. Purpose of classification

As said before, the classification relies on extracted features in order to decide whether unknown samples are more likely to be clean or suspicious. We target to minimize the FP rate. Also, a score on clean/suspicious is better than a simple clean/suspicious answer, because it allows to assess the degree of suspiciousness, i.e. the priority to give to samples for manual analysis purpose.

B. Classifying with Alligator

Alligator is a free and open-source tool for classification [29]. 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: pictures, applications, 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 II) and is only meant to be done once in a while. Alligator classification algorithms and specificities have been first published in [30].

SVM [31] and Adaboost [32] are among the most well-known classifiers, especially for the classification of malware and pictures. We have developed the Alligator classification engine for the following reasons:

1) **Classification accuracy**. 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. Alligator can thus be seen as a meta-classifier. 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 VI). Several comparisons with images are provided in [33]. Basically, the paper demonstrates a 9% of better classification with Alligator than with SVM for the make up/non-make up categories, and for publicly available sets of images (FCD database) [33]. Similar results have been obtained with several other sets of pictures and application domains.
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) **Forget/Boost too abnormal elements**. During the learning phase, Alligator is able to automatically determine a small set of elements that impact a lot the weight to be applied to classification algorithms. When such a set is identified, Alligator can forget about them to generate the weights, or boost them when they are particular representative.
4) **Lightweight and simplicity**. Alligator is a standalone Java program. It does not require external packages, and its installation is a matter of unzipping an archive. Its learning stage is easy to customize: simply edit a human readable script with a text editor.

VI. Results

A. Performance

The performance of Alligator's learning and classification stages is depicted at Figure 2. The results depend on the number of samples in the learning clusters.

Learning time is faster with Alligator than with SVM[1], but classification time is lower for SVM - it however remains reasonable for Alligator. With clusters of 50 K samples, Alligator learning 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

[1]We have made the comparison with *jlibsvm* [34]

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. The classification of guess clusters takes around 6 min 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 FP rate.

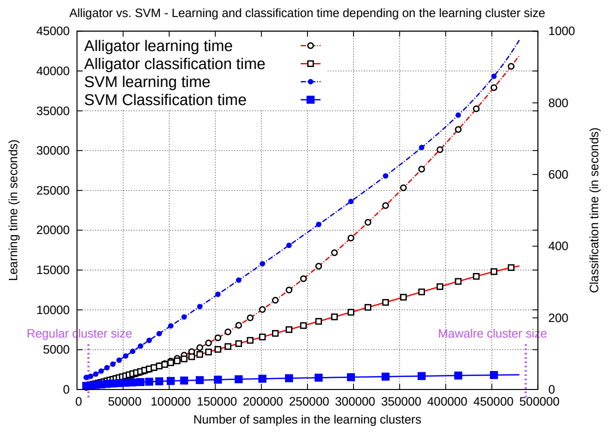

Fig. 2. Learning and classification time for Alligator and SVM.

B. False positive and false negative rates

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. Samples are classified either according to trusted sources or after manual analysis. At the end of its learning, 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 usually get excellent rates (99.9%) at this stage.

Then, Alligator is asked to classify two new clusters 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. There are 1,512 clean samples and 3,062 malicious ones. We ask Alligator to classify the samples automatically, and check how well it performs in terms of recognition rate.

2) Classification results: Figure 3 depicts the results in terms of overall classification efficiency, and of False Positive (FP) and False Negative (FN) rates, and for both Alligator and SVM. A low FP rate is our prime interest: indeed, as stated before, AV labs do not waste time over analysis of clean samples. On the contrary, missing malware is not desirable but less important, since so many are already not identified...

The curve shows that, whatever cluster size, Alligator always performs better than SVM for the FP rate. The fact clean and malware clusters are strongly unbalanced heavily impacts SVM which reaches a FP rate of 65%. Alligator is impacted too but far less: FP is below 1% with 60 K clusters, and 1.78% at 480 K. Moreover, the average FP/FN recognition rate is almost always better with Alligator. For 50 K samples, precise results are given in Table II.

C. Identifying unknown malware

So far, SherlockDroid has extracted features of 615,547 samples: $486,890 + 3,602 + 1,512$ during classification tests, and 124,083 samples actively crawled from Android marketplaces. It has detected 2 new unknown malware: Android/MisoSMS.A!tr.spy (December 2013) and Android/Odpa.A!tr.spy (July 2014). It has also discovered 7 potentially unwanted applications (PUA): Adware/Geyser and riskware SmsControlSpy, Zdchial, SmsCred, Blued, Flexion and SneakFront. 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://www.fortiguard.com/encyclopedia for a precise description of those malware and PUA.

From those discoveries, we note that SherlockDroid seems particularly successful at identifying 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.

D. Typical limits of SherlockDroid

The most common FPs 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 features 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

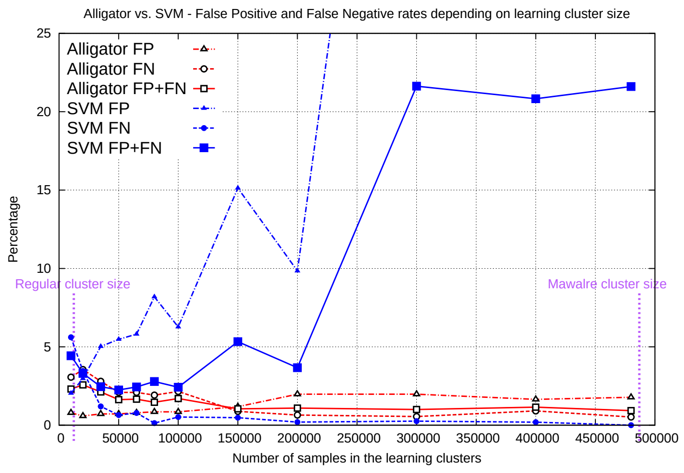

Fig. 3. FP and FN rates for Alligator and SVM

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

TABLE II
LEARNING AND CLASSIFICATION RESULTS OF ALLIGATOR

tweaks (su, busybox, system commands...) which, once again, trigger false alarms.

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 IV, this is for instance how we got the idea to identify third party kits and rule them out.

VII. CONCLUSION - FUTURE WORK

SherlockDroid has been designed to identify new Android malware which aren't known in the anti-virus community yet. It is based on marketplace crawlers, feature extractors and a classification engine specifically adapted to clean/malware identification.

Running SherlockDroid live, we have found 2 new malware and 7 potentially unwanted applications by crawling over 120,000 applications within 5 different marketplaces. With tests, we have extracted features of over 600,000 samples. Compared to prior research work, this is SherlockDroid's main achievement: we are not aware of any other system identifying unknown malware in the wild, apart from DroidRanger which detected 2 new families. Numerous research projects have only been tested on a few *known malware* or *artificial malware*.

We plan several improvements for SherlockDroid. As for feature extraction, we contemplate the use of contextual information in correlation of each feature. Contextual information could be data like the call stack of the feature. This would help us differentiate benign from malicious cases. For example, if we consider the "send email" feature, 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 potentially unwanted applications and malware. This is difficult because there is no obvious feature 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.

REFERENCES

[1] D. Harley and A. Lee, "Heuristic Analysis - Detecting Unknown Viruses," March 2007.

[2] L. Seltzer, "Lessons of the McAfee False Positive Fiasco," April 2010, http://securitywatch.pcmag.com/malware/283982-lessons-of-the-mcafee-false-positive-fiasco.

[3] B. Popa, "AVG Anti-Virus Breaks Down Windows XP Due To False Positive," March 2013, http://news.softpedia.com/news/AVG-Anti-Virus-Breaks-Down-Windows-XP-Due-to-False\--Positive-337395.shtml.

[4] G. Dini, F. Martinelli, A. Saracino, and D. 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*, ser. Lecture Notes in Computer Science, vol. 7531. St. Petersburg, Russia: Springer, "October" 2012, pp. 240–253.

[5] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu, "pBMDS: a behavior-based malware detection system for cellphone devices," in *Proceedings of the third ACM conference on Wireless network security*, ser. WiSec '10. New York, NY, USA: ACM, 2010, pp. 37–48.

[6] I. Burguera, U. Zurutuza, and S. 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*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 15–26.

[7] M. e. a. Lindorfer, "AndRadar: fast discovery of android applications in alternative markets," in *Proceedings of the 11th Conf. on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2014.

[8] M. L. et al., "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.

[9] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. 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*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971

[10] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '14. New York, NY, USA: ACM, 2014, pp. 221–233.

[11] Y. Zhang, M. Yang, Z. Yang, G. Gu, P. Ning, and B. Zang, "Permission use analysis for vetting undesirable behaviors in android apps," *Information Forensics and Security, IEEE Transactions on*, vol. 9, no. 11, pp. 1828–1842, Nov 2014.

[12] T. Shen, Y. Zhongyang, Z. Xin, B. Mao, and H. Huang, "Detect android malware variants using component based topology graph," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, Sept 2014, pp. 406–413.

[13] L.-K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *USENIX Security Symposium*, 2012, pp. 569–584.

[14] Z. Aung and W. Zaw, ""permission-based android malware detection"," *International Journal of Scientific and Technology reseach*, vol. 2, Mar. 2013.

[15] T. Bläsing, A.-D. Schmidt, L. Batyuk, S. A. Camtepe, and S. Albayrak, "An Android Application Sandbox System for Suspicious Software Detection," in *5th International Conference on Malicious and Unwanted Software (MALWARE'2010)*, Nancy, France,, 2010.

[16] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "Andromaly: a behavioral malware detection framework for android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, Feb. 2012.

[17] 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)*, February 2014.

[18] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic security analysis of smartphone applications," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 209–220.

[19] Y. Zhou, Z. Wang, W. Zhou, and X. 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.

[20] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "MAST: Triage for Market-scale Mobile Malware Analysis," in *Proc. 6th WiSec*, 2013.

[21] S. Zhao, X. Li, G. Xu, L. Zhang, and Z. Feng, "Attack tree based android malware detection with hybrid analysis," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, Sept 2014, pp. 380–387.

[22] J. Chandraiah, "Fake anti-virus: The journey from Trojan to a persistent threat," May 2012, https://sophosnews.files.wordpress.com/2012/05/fakeav-image-adjust1.pdf.

[23] M. Mas'ud, S. Sahib, M. Abdollah, S. Selamat, and R. Yusof, "Analysis of features selection and machine learning classifier in android malware detection," in *Information Science and Applications (ICISA), 2014 Int. Conf. on*, May 2014, pp. 1–5.

[24] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. A. Maranon, "Puma: Permission usage to detect malware in android." ser. Advances in Intelligent Systems and Computing, A. e. a. Herrero, Ed., vol. 189. Springer, 2012, pp. 289–298.

[25] M. Zhao, T. Zhang, F. Ge, and Z. Yuan, "Robotdroid: A lightweight malware detection framework on smartphones," *Journal of Networks*, vol. 7, no. 4, 2012. [Online]. Available: http://ojs.academypublisher.com/index.php/jnw/article/view/jnw0704715722

[26] A. Apvrille and L. Apvrille, "Sherlockdroid, an inspector for android marketplaces," in *Hack.lu*, Luxembourg, Oct. 2014.

[27] Patrick Schulz, "Dalvik Bytecode Obfuscation on Android," July 2012, http://www.dexlabs.org/blog/bytecode-obfuscation.

[28] Fortiguard Center, "Android/RuSMS.AO," 2013, fortiguard Encyclopedia, http://www.fortiguard.com/encyclopedia/virus/#id=5897642.

[29] L. Apvrille, "Alligator: AnaLyzing maLware wIth partitioninG and probAbiliTy-based algORithms," 2014, http://http://perso.telecom-paristech.fr/~apvrille/alligator.html.

[30] L. Apvrille and A. Apvrille, "Pre-filtering Mobile Malware with Heuristic Techniques," in *GreHack*, November 2013, pp. 43–59, grenoble, France.

[31] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[32] R. E. Schapire and Y. Singer, "Improved boosting algorithms using confidence-rated predictions," in *Machine Learning*, 1999, pp. 80–91.

[33] N. Kose, L. Apvrille, and J.-L. Dugelay, "Facial Makeup Detection Technique Based on Texture and Shape Analysis ," in *11th IEEE International Conference on Automatic Face and Gesture Recognition (FG 2015)*, May 2015, ljubljana, Slovenia.

[34] D. Soergel, "Efficient training of Support Vector Machines in Java," 2014, https://github.com/davidsoergel/jlibsvm.