A Data-Driven Characterization of Modern Android Spyware

FABIO PIERAZZI, King's College London, UK
GHITA MEZZOUR, International University of Rabat, FIL, TICLab, Morocco
QIAN HAN, Dartmouth College, USA
MICHELE COLAJANNI, University of Modena and Reggio Emilia, Italy
V.S. SUBRAHMANIAN, Dartmouth College, USA

According to Nokia's 2017 Threat Intelligence Report, 68.5% of malware targets the Android platform—Windows is second with 28%, followed by iOS and other platforms with 3.5%. The Android spyware family UaPush was responsible for the most infections, and several of the top 20 most common Android malware were spyware. Simply put, modern spyware steals the basic information needed to fuel more deadly attacks such as ransomware and banking fraud. Not surprisingly, some forms of spyware are also classified as banking trojans (e.g., AceCard). We present a data-driven characterization of the principal factors that distinguish modern Android spyware (July 2016 - July 2017) both from goodware and other Android malware, using both traditional and deep ML. First, we propose an Ensemble Late Fusion (ELF) architecture that combines the results of multiple classifiers' predicted probabilities to generate a final prediction. We show that ELF outperforms several of the best known traditional and deep learning classifiers. Second, we automatically identify key features that distinguish spyware both from goodware and from other malware. Finally, we provide a detailed analysis of the factors distinguishing five important families of Android spyware: UaPush, Pincer, HeHe, USBCleaver, and AceCard (the last is a hybrid spyware-banking trojan).

CCS Concepts: • **Security and privacy → Mobile platform security**; **Software and application security**; • **Computing methodologies → Supervised learning by classification**; • **Social and professional topics → Malware / spyware crime**;

Additional Key Words and Phrases: Machine Learning, Malware, Android, Spyware, Characterization

ACM Reference format:
Fabio Pierazzi, Ghita Mezzour, Qian Han, Michele Colajanni, and V.S. Subrahmanian. 0. A Data-Driven Characterization of Modern Android Spyware. *ACM Trans. Manag. Inform. Syst.* 1, 1, Article 111 (0), 36 pages.
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

According to a Kaspersky Labs report [6], the percentage of mobile malware that is spyware went up from 8.44% in the last quarter of 2016 to 10.27% in the first quarter of 2017, which represents an over 20% increase. According to McAfee's 2017 Mobile Threat report [5], spyware on mobile devices went up 40% in 2016. Several of the top 20 Android malware listed in Nokia's 2017 Threat Intelligence Report are spyware [19]. These recent statistics suggest that spyware is becoming an increasingly important problem. For instance, a McAfee report [5] describes an app designed to increase follower counts on Instagram that leads unsuspecting users to a phishing website.

2158-656X/0/0-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

Spyware is software that illegally gathers information and sends it to an attacker's Command and Control server. Unlike ransomware (which explicitly locks a user's screen and prevents access to files through encryption), SMS fraud (where users may notice charges), or banking trojans (where users may notice money disappearing from their bank account), spyware is stealthy and can siphon off data from calendars, emails, SMSs, contact lists, social media accounts, and more, without the user becoming aware of it. Spyware also constitutes an initial "reconnaissance" phase for more complex attacks. For instance, spyware is often associated with more dangerous threats such as cyber espionage targeting top executives (e.g., Exaspy [37]), whaling attacks (in which large amounts of money are siphoned off from companies [4]), and private data theft for blackmail and extortion purposes (e.g., compromising photos [41]).

The goal of this paper is to characterize the factors that distinguish Android spyware from both goodware and other Android malware. To achieve this, we use VirusTotal to build a dataset of a recent Android samples (July 1 2016 to July 1 2017 time frame). We collect 5,000 spyware, 5,000 goodware, and 5,000 other malware (non-spyware) samples, and then rely on the *Koodous* [3] online service to perform lightweight static and dynamic analysis of these apps.[1] We extract easy to understand features [15] from the resulting logs. We then test several well known classifiers, both traditional and deep, on two problems: distinguishing between spyware and goodware, and between spyware and other malware. We combine these classifiers' output probabilities in an Ensemble Late Fusion (ELF) architecture which achieves the best results—for distinguishing between spyware vs. goodware, it achieves 0.982 F-Score, 0.982 AUC and 0.98% (i.e., <1%) false positive rate. To characterize spyware, we identify the most significant features in separating spyware from the other two classes by examining the feature importances of the best classifiers, histograms of feature values, and graphical representation of classifier rules.

We then perform a detailed analysis of the distinguishing factors of 5 major Android malware families with respect to goodware, other malware, as well as other families of spyware. These 5 families are: UaPush, Pincer, HeHe, USBCleaver, and AceCard. The behavior of these 5 families varies substantially from each other as well as from goodware, other malware (i.e. malware that is not spyware), and even other types of spyware. For instance, AceCard is different from other spyware because it reads and writes many XML, JAR, and APKs, suggesting many unpacking stages. On the other hand HeHe does not show evidence of unpacking, but requests many dangerous permissions such as to mount/unmount filesystems, to change configuration files, and to access user location. UaPush on the other hand differs by performing suspicious encryption operations, possibly linked to unpacking or for stealthily sending sensitive data to the attacker. USBCleaver compromises Windows machines which are connected infected phones and requests mainly permissions to write to external storage. Pincer starts several background processes on the phone, possibly in the hope that detection/removal of one of them by a user would allow it to keep functioning because of the continued presence of the others. To the best of our knowledge, our study of the relationships of these 5 Android spyware families to other spyware families, to other malware families, and to goodware, is the first of its kind, and shows that spyware operates via a multiplicity of diverse techniques.

The closely related research area of *information leakage detection* primarily relies on *taint data flow* [26] and *network traffic* analysis [18]. These techniques detect the flow of both legitimate and inappropriate traffic and require assessment of an app's privacy policy in order to differentiate between the two. In contrast to the information leakage literature, our work focuses on Android spyware and our approach directly learns conditions that predict whether a given sample is spyware or goodware, as well as whether it is spyware or some other category of malware, and does not require assessment of apps policies or user expectations.

Unlike much work that separates Android malware from goodware, (e.g., [7, 10, 53]), our goal is instead to characterize the features that distinguish spyware from either goodware or other forms of malware. Wang et

[1] Approximately 20% of these samples were discarded because they were either corrupted and/or because they crashed during analysis. This is common in past work on dynamic analysis, e.g. [15, 20, 59, 62].

al. [77] present an interesting prior study of spyware using machine learning; however, Want et al. [77] analyze desktop spyware from over twelve years ago, and only uses SVM. Conversely, our paper uses recent Android apps from July 2016 to July 2017, uses a host of traditional and deep classifiers combined as an Ensemble Late Fusion to understand the most accurate classifiers which can support a characterization.

In summary, we make the following contributions:

(1) We consider the task of separating spyware vs. goodware and spyware vs. other malware, and compare the performance of several traditional and deep learning methods. We show for the first time that ensemble learning with different classifiers leads to better 10-fold performance for identifying spyware. To the best of our knowledge, past ML-based malware detection studies have not studied spyware in depth—they mostly consider the general problem of distinguishing goodware vs. malware, and rely on just one individual classifier (e.g., SVM in [10], RF and variants in [53, 72]). Instead, we propose an *ensemble late fusion* (ELF) architecture which outperforms all individual classifiers.
(2) We identify the features that are key to separating modern Android spyware (between July 2016 and July 2017) from goodware and from other malware. We also provide an in-depth characterization of 5 major spyware families, and show how each of them exhibits unique behaviors even when compared to other spyware. Existing work on malware characterization has not focused on spyware — in addition to malware vs. goodware classification, they look at related problems such as predicting malware spread [40] or distribution of goodware and malware in different Android markets [49].

The remainder of the paper is structured as follows. Section 2 provides a brief overview of Android apps and presents five major Android spyware families. Section 3 describes our methodology, including dataset collection, feature extraction and classifiers. Section 4 presents our results on predicting spyware vs. goodware and spyware vs. other malware. Section 5 presents our novel characterization of spyware with respect to goodware and other malware, and an in-depth data-driven characterization of 5 major spyware families. Section 6 discusses related research. Section 7 concludes with limitations, summary of main findings and future work.

2 BACKGROUND

We now provide a quick overview of Android apps (Section 2.1) and on major spyware families (Section 2.2).

2.1 Android Apps

Android apps consist of compressed files called Android Package Kits (APKs). For an APK to be installed on an Android device, it has to contain the program code, developer certificates, and a Manifest file. The Manifest is a metadata file that includes the list of *permissions* required and the list of *components* of the app. Android apps can only access certain resources if they explicitly request the corresponding permissions in the Manifest file. Users need to explicitly approve these permissions either during the installation phase or the runtime phase.[2] In addition to the list of permissions, the Manifest also contains information about the following *components* within an app's structure:

- **Activities**, which correspond to app screens and are used to manage user interface.
- **Services**, which are components that run as background processes.
- **Intents**, which are messaging objects used to request actions from other components.
- **Content providers**, which allow for data exchange across apps.
- **Broadcast Receivers**, which are used to register broadcast messages from the system and other apps.

[2] Unfortunately, requiring users to approve these permissions is rarely effective because users tend to grant permissions without carefully understanding the consequences.

It is important to note that malware that gains root privileges on the device does not have to include all the above information in the Manifest [21]. However, the prevalence of such malware is still relatively low [34].

2.2 Major Spyware Families

This section provides a brief overview of 5 major spyware families (according to Symantec [75]). Their main behaviors are summarized in Table 1.

(1) AceCard. Discovered first in 2014, AceCard had a resurgence of activity from 2015. According to a 2016 Kaspersky report [42], AceCard overlays its own interfaces on top of Web forms associated with popular applications such as Facebook, Twitter, Instagram, Viber, Paypal, and over 30 banks. When a user opens one of these apps on his Android device, AceCard immediately captures confidential information typed into the forms. According to an Oct 2016 McAfee report [14], AceCard even tricks users into revealing their mother's maiden name and other sensitive data, so as to defeat two-factor authentication systems. It even asks unsuspecting users to take a picture with an identity document such as a passport or driving license.
(2) HeHe. HeHe seeks to identify a compromised user's private information including banking credentials. When the infected phone receives either an SMS or a phone call from a list of phone numbers of interest to the attacker (e.g., bank's or credit card company's phone numbers), HeHe deletes the message or rejects the call [23], while simultaneously forwarding this information to the attacker's command and control server. Thus, the attacker can siphon off money from the victim's bank account without the victim receiving any warning messages from the bank.
(3) Pincer. First discovered in 2013, Pincer automatically sends information such as the phone number, International Mobile Equipment Identity (IMEI), carrier, operating system information, SMS traffic, and more to a command and control server [28]. It is believed that Pincer likely acts as a front for subsequent banking trojan activity.
(4) UaPush. One of the most prevalent malware families in recent years, UaPush steals personal information including IMEI, bookmarks, and call history, amongst others [74]. In addition to these activities, some variants may send out premium SMS messages.
(5) USBCleaver. Unlike the preceding malware families, USBCleaver hops to a Windows device when the Android phone is connected via USB to steal information from the connected computer. According to an F-Secure report [27], it provides the attacker with the capability to collect passwords (e.g., in different browsers, in WiFi settings).

3 METHODOLOGY

3.1 Overview

The main goal of this paper is to identify the key factors which distinguish between (i) spyware and goodware and (ii) spyware and other malware. We use VirusTotal to create a recent representative dataset of spyware, goodware, and other (non-spyware) malware (Section 3.2) from which we extract a set of easy-to-interpret static and dynamic features (Section 3.3). To perform a data-driven characterization, we first evaluate detection performance of both traditional and deep ML approaches in distinguishing spyware from goodware and other malware (Section 4). Since our focus is on characterization, we operate in absence of concept drift. We use traditional 10-fold cross validation to evaluate classifier performance using F_1-Score, Precision, Recall, AUROC, False Negative Rate (FNR), and False Positive Rate (FPR). To leverage the best of each classification algorithm, we develop the ELF Ensemble Late Fusion architecture that uses a weighted sum of the probability of being spyware generated by each classifier. Once the weights are learned from the training data, ELF generates better results than all the traditional and deep ML classifiers considered. We use the best performing classifiers to identify the most

Table 1. Behaviors of the main spyware families studied in this paper, according to publicly available AV reports.

	Steals information from device	Creates backdoor	Steals info from USB-connected PCs	Tries to get root access	Requests lots of permissions	Spreads via Bluetooth	Fake login screens	Game repackaging	Fake system update	Sends/Intercepts SMS	Blocks incoming SMS/calls	Hides its icon	Downloads other packages	Fake ads in system bar	Likely steals calls history
AceCard	✓			✓	✓		✓		✓	✓					
HeHe	✓			✓	✓				✓		✓	✓	✓		
UaPush	✓				✓					✓				✓	✓
Pincer	✓	✓								✓	✓				
USBCleaver			✓										✓		

important factors distinguishing spyware from goodware and other malware. We show the associated histograms of values of such features for both classes as well as associated decision trees (Section 5 and Appendix).

3.2 Dataset Collection

We download 5000 spyware, 5000 goodware and 5000 other malware samples from *VirusTotal* [1], a service that scans suspicious files and URLs submitted by users to be tested against multiple commercial AV systems. All samples in our dataset were first submitted to VirusTotal between July 1 2016 and July 1 2017. This choice is also motivated by [55] which empirically determines that VirusTotal's AV detections become stable one year after initial submission.

We use *Symantec threat descriptions* [75] to identify spyware families. In particular, we consider an Android malware family to be a spyware family if the Symantec indicates that the malware family steals information such as location, browsing history, credentials, contacts or photos. In total, we obtain 54 spyware family names. In the Appendix, Table 13 reports the spyware family names with sample count.[3] It is worth noting that some spyware families may also have some overlapping behaviors typical of other malware categories (e.g., sending premium SMSs, banking trojans). We use the VirusTotal Intelligence API to download Android APKs belonging to our list of spyware families that were detected as malicious by at least 10 antivirus engines [55].

We also use the VirusTotal Intelligence API to collect malware samples that are not spyware; we refer to this category as *other-malware*.[4] Finally, we collect *goodware* by querying the VirusTotal Intelligence API for samples first found in the wild between July 2016 and July 2017 which were not labeled as malware by any of the 63 VirusTotal antivirus engines (i.e., detection rate was 0%).

[3]We also tried to download the following families in [75] form VirusTotal [1], but no samples were present (for the period July 2016 - July 2017): Spywaller, Sockrat, Bossefiv, Alienspy, Accstealer, Gomal, Fitikser, Ballonpop, Repane, Dupvert, Sberick, Simhosy, ZertSecurity, Teelog, Yatoot.

[4] In the query submitted to VirusTotal, we look for Android APKs that have been detected as malicious by at least 10 antivirus engines, and that do not belong to our list of spyware families and do not contain "spyware" keywords. As in related work [10], we also decide to exclude adware from this other-malware category because they usually represent applications that annoy users with an excessive number of ads, as opposed to compromise user devices.

Table 2. Static features.

Group	Feature	Description
Author	author	Derived from SHA256 hash of certificate used to sign APK
Permissions	n_std_sw_perm	Number of standard software permissions required to install the app
	n_std_sw_ _perm_dangerous	Number of standard software permissions marked as dangerous in Android documentation
	n_hw_perm	Number of standard hardware permissions required to install the app
	n_custom_perm	Number of custom permissions (i.e., non-standard) defined by the app developer
	$\texttt{sw\_permission}_i$	1 if $\texttt{sw\_permission}_i$ is required by the app, 0 otherwise (i ∈ 1,..,N)
	$\texttt{hw\_permission}_j$	1 if $\texttt{hw\_permission}_j$ is required by the app, 0 otherwise (j ∈ 1,..,K)
Structure	filesize	Application size in bytes
	n_activities	Number of activities
	n_intents	Number of filtered intents
	n_providers	Number of content providers
	n_services	Number of services
	n_receivers	Number of broadcast receivers

Table 3. Dynamic features.

Group	Feature	Description
RW	read [[<basepath>] [<filename>]]	*N*-gram counts about read operations on the Android filesystem. The basepath is just the name of the first folder after the root
	write [[<basepath>] [<filename>]]	*N*-gram counts about write operations on the Android filesystem. The basepath is just the name of the first folder after the root
System	servicestart [<servicename>]	*N*-gram counts about started background processes (i.e., services)
	load [[<classpath>][<classname>]]	*N*-gram counts about dex Android classes loaded during execution
	crypto [[<algorithm>] [<encryptionkey>]]	*N*-gram counts about crypto operations performed during execution
Network	sendnet [[<protocol>] [<port>]]	*N*-gram counts about outgoing network activity
	recvnet [[<protocol>] [<port>]]	*N*-gram counts about incoming network activity
SMS	sendsms [[<phonenumber>] [<message>]]	*N*-gram counts about sms sent by the application

Static and dynamic analysis of samples was performed by an online Android analysis service called *Koodous* [3], which mainly relies on *Androguard* for static analysis, and *Cuckoo* and *Droidbox* sandboxes for dynamic analysis. Koodous has successfully performed static and dynamic analysis of 3,698 (out of 5,000) spyware samples, 4,481 (out of 5,000) other malware samples, and 3,669 (out of 5,000) goodware samples. The lower number of actual logs is due to the fact that some applications were corrupted APKs that did not work and/or crashed during dynamic analysis. We note that having apps not working for dynamic analysis is common to past work on (Android) malware—for instance, in DroidScribe [20], Park et al. [62], EC2 [15] and Onwuzurike et al. [59]. The output JSON logs generated by Koodous are then used to extract a set of static and dynamic features for the input to our classifiers.

3.3 Feature Extraction

We extracted *static* and *dynamic* features from the logs obtained through the Koodous service [3].

3.3.1 Static features. Table 2 shows the static features we consider which are designed to be easy-to-interpret. We now describe the main categories of static features in detail.

Author. We use the hash of the certificate used to sign the app in order to identify the author of an app. The underlying assumption here is that if the same certificate is used to sign multiple apps, then these apps have the same author. Thus, a developer who uses multiple certificates to sign his apps will appear as multiple authors.

Permissions. We use the number of permissions that the app requests. We distinguish between regular software permissions and software permissions marked as dangerous in Android documentation.[5] Dangerous permissions grant access to sensitive data such as call logs and contact lists. We also include the number of hardware permissions requested and the number of non-standard permissions requested. Non-standard permissions correspond to custom permissions requested by a developer. In addition to the number of permissions, we use binary variables that capture whether each standard permission is requested (see Table 2).

Structure. In order to capture the app structure, we have features associated with different types of components in the Manifest: `n_activities`, `n_intents`, `n_services`, `n_providers` and `n_receivers`. We choose to use the number of components rather than the names of these components because malware developers can easily change component names in order to evade detection (as component names are Java class names).

3.3.2 Dynamic features. Koodous [3] performs dynamic analysis by running samples in the *Droidbox* [2, 44] and *Cuckoo* [58] sandboxes which log app behavior. Dynamic analysis may reveal information which may be harder to detect through static analysis (e.g., runtime unpacking of malicious routines [54]). Since Droidbox and Cuckoo produce very similar logs, without loss of generality we consider the logs generated by Droidbox as it produces more information: logs of incoming and outgoing network operations, SMS and phone calls, in addition to file read and write and other system operations (e.g., crypto functions). Moreover, Droidbox provides logs of cryptographic operations performed using the Android API as well as newly started background services and dynamically loaded classes at runtime.

To model dynamic features, we leverage an n-gram representation commonly used in malware analysis and classification (e.g., [12, 24, 31, 67]). Using n-grams, we *count* the following operations captured by the sandbox: *RW* (read and write operations), *System* (e.g., start of a new background process), *Network* (Internet requests), *SMS* (texts sent by the device). The considered n-grams are reported in Table 3. For each dynamic log operation, we extract several possible sets of words in order to limit the impact of simple obfuscation strategies adopted by the attacker (e.g., changing the filename of a written file or folder). We consider a *bag of words* approach in which we extract n-gram counts. We experimentally verified that considering n-grams with $n > 3$ does not yield any performance improvement (because the level of detail increases too much and the features become overly specific, as discussed in [20, 24]). We therefore consider unigrams, bigrams and trigrams. Each unigram/bigram/trigram is a feature whose value is represented by the number of occurrences of that unigram/bigram/trigram in the logs [67]. The idea is that similar malware execute similar sequences of operations.

3.3.3 Summary of Features. The dataset used in this paper contains 190 static features (author: 1, structure: 6, and permissions: 183) and 24,465 dynamic features in total. We rely on 10-fold Cross Validation (CV) to consider different training settings with our classifiers and features. In particular, we use a standard 10-fold CV in which all algorithms are presented with all 190 static and all 24,465 dynamic features. Our results are presented as is, despite the fact that certain features may have only one value in all apps in the training data—which means that those features would not be considered significant during the classification process.

[5] https://developer.android.com/guide/topics/permissions/overview#normal-dangerous

3.4 Supervised Learning Algorithms and Ensemble Late Fusion (ELF)

3.4.1 Traditional Classifiers. We consider six widely used traditional classifiers [32]: Random Forest (RF), Decision Tree (DT), Supper Vector Machine (SVM), K-Nearest Neighbors (KNN), Naïve Bayes (NB), Logistic Regression (LR). Traditional classification algorithms take as their input a set of *(feature vector, class)* pairs. For instance, for each Android app, we extract a feature as described in Section 3.3, and the class is either 1 ("spyware") or 0 ("goodware" or "other-malware", depending on the experiment we are considering). Traditional classification algorithms try to find different equational forms that separate one class from the others. For instance, linear Support Vector Machines (SVM) try to draw a hyperplane that splits the feature vector space into two classes. SVM may use different equational forms: linear SVM uses a straight hyperplane, while kernel SVMs use diverse shapes. Other traditional classifiers (e.g. Decision Tree) use a set of generalized (to higher dimensions) rectangles to state that if a feature vector is within one of these generalized rectangles then it is most likely spyware; and in the other class (either goodware or other malware, depending on the problem we study) when it is not in any such rectangle. Hence, different classifiers try to split the space into a "spyware" part and a second part (either goodware or other-malware). As the assumptions made by some classifiers may be inconsistent with the actual data: some traditional classifiers may perform well, while others may perform poorly. One of the major goals of machine learning is to identify the right classifier for any given data set. We therefore use multiple traditional classifiers to see which one is best at distinguishing spyware vs. goodware, and spyware vs. other malware types.

3.4.2 Deep Learning Classifiers. We additionally consider four representative *deep learning* classifiers which have achieved outstanding results in many ML tasks: Multi-Layer Perceptrons (MLP), Bernoulli Restricted Boltzmann Machines (BRBM), Convolutional Neural Networks (CNN), and the *Wide & Deep* [17] DL architecture recently proposed by Google. We do not consider sequence-based classifiers as the dynamic analysis logs are a collection of events without timestamps, and because our goal is to consider static and dynamic features together (see Section 3.3). We performed extensive hyper-parameter tuning to achieve the best results with deep learning methods (despite the fact that they are less accurate than shallow learning methods, probably due to the limited data available). We include the detailed description of DL architectures used and how hyper-parameter optimization was done in Appendix B.

3.4.3 Ensemble Late Fusion (ELF). Our ELF architecture shown in Figure 1 combines the results of traditional and deep classifiers into a unified prediction. The first part of ELF follows a traditional cycle and is shown in the top part of Figure 1. From a set of Android APKs, we extract a set of static and dynamic features, and then perform a supervised classification step using a set of classifiers. Instead of using the binary prediction generated, ELF, uses the probability returned by each classifier that a particular Android app belongs to the class 1 (spyware) vs. the class 0 (either goodware or other malware). ELF then computes a weighted sum of these probabilities by assigning a weight to each classifier such that the weights add up to one. In order to assign these weights, ELF performs a grid search to identify near-optimal weights without looking at the test set.

To describe ELF more formally, let us consider a binary classification task where an object x_j can have a predicted label $\hat{y}_j = 0$ (goodware or other malware), and $\hat{y}_j = 1$ (spyware). Each classifier outputs a probability of belonging to class 1, $p_j \in [0.0, 1.0]$, where for a given object x_j, if $p_j > 0.5$ then the predicted label $\hat{y}_j = 1$, else $\hat{y}_j = 0$. Since we are considering several supervised classifiers, the i-th classifier outputs a certain probability p_j^i that an object x_j belongs to class 1. The ensemble algorithm computes a Late Fusion Score (LFS) as the weighted sum of the probabilities of all the classifiers:

$$LFS_j = \sum_{i \in \text{classifiers}} w_i p_j^i \tag{1}$$

where w_i is the weight (relevance) of the i-th classifier in the decision, and $\sum_i w_i = 1$. Then, if $LFS_j > 0.5$ an object x_j is assigned label $\hat{y}_j = 1$, otherwise $\hat{y}_j = 0$.

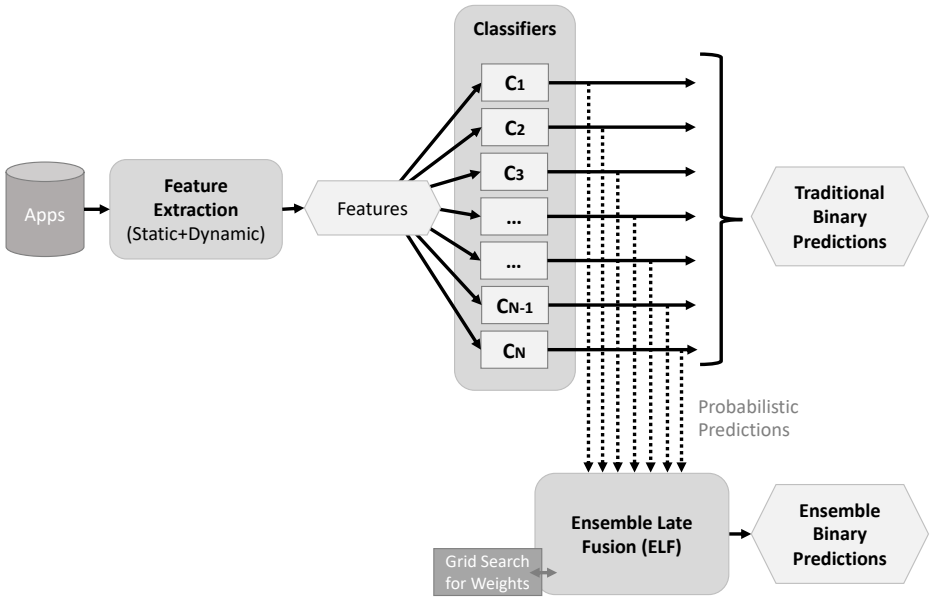


Fig. 1. Scheme of the ELF architecture.

3.4.4 Training ELF weights. We now describe how we identify the best weights for ELF to use. We first split the dataset into two parts: ELF training set and ELF testing set. The identification of the weights relies exclusively on the training set, on which we perform a 10-fold cross-validation with all possible weights combinations of the different classifiers; the performance on the validation are determined according to the weighted score described previously in Equation 1. The *optimal ELF weights* correspond to the ones that obtain the highest performance on the validation set, for all the 10-folds. Note that the testing set is *never* involved in the weight training process. This weight training procedure is repeated in a 10-fold CV fashion for different train/test splits. The final ELF results that we report are the 10-fold average performance obtained by using the best weights found with this procedure.

4 EVALUATION

We evaluate the accuracy of the different classifiers described earlier using 10-fold cross-validation (see Section 3). Identifying the best performing classifier is a necessary step to determine the features that best distinguish spyware from goodware and from other malware.

Metrics. We use the following traditional ML performance metrics: F_1-Score, Precision, Recall, AUC (AUROC), FPR (False Positive Rate) and FNR (False Negative Rate) for a complete view of the performance of every approach. We also report *p-values* obtained via a Student's t-test in order to show statistical significance. Readers will recall that a result is statistically significant when the p-value is less than 0.05 [32]. Moreover, we recall that: the lower the p-value, the higher the statistical significance.

4.1 Spyware vs. Goodware

Table 4 shows the performance of different traditional classifiers in separating spyware from goodware.

Table 4. Performance of classifiers in **spyware vs goodware** (average with 10-fold cross-validation). The best traditional classifier is RF. The best overall classifier is ELF with weights $w_{RF} = 0.711$, $w_{Wide\&Deep} = 0.222$, $w_{BRBM} = 0.045$, and $w_{SVM} = 0.022$. The p-values are computed via a t-test comparing F_1-Scores. We also use standard statistical notation ***, **, * (***<0.01; **<0.05; *<0.1).

Algorithm	F_1	P	R	AUC	FPR	FNR	p-val vs. DT	p-val vs. RF	p-val vs. ELF
RF	0.976	0.990	0.963	0.977	0.98%	3.70%	*** ($2.962 \cdot 10^{-38}$)		*** ($4.309 \cdot 10^{-53}$)
DT	0.971	0.970	0.973	0.971	3.08%	2.70%		*** ($1.302 \cdot 10^{-39}$)	*** ($1.028 \cdot 10^{-122}$)
KNN	0.869	0.868	0.870	0.868	13.36%	13.01%	*** ($4.041 \cdot 10^{-277}$)	*** ($8.897 \cdot 10^{-282}$)	*** ($1.158 \cdot 10^{-287}$)
NB	0.782	0.663	0.954	0.732	48.95%	4.80%	*** (0)	*** (0)	*** (0)
LR	0.704	0.774	0.645	0.727	19.08%	35.45%	*** (0)	*** (0)	*** (0)
SVM	0.423	1.000	0.269	0.635	0.00%	73.09%	*** (0)	*** (0)	*** (0)
MLP	0.674	0.907	0.537	0.741	5.59%	46.30%	*** (0)	*** (0)	*** (0)
BRBM	0.701	0.918	0.568	0.758	5.15%	43.18%	*** (0)	*** (0)	*** (0)
CNN	0.623	0.590	0.783	0.587	62.90%	21.73%	*** ($4.083 \cdot 10^{-95}$)	*** ($3.157 \cdot 10^{-169}$)	*** ($1.568 \cdot 10^{-165}$)
Wide & Deep [17]	0.671	0.883	0.548	0.742	8.21%	45.19%	*** ($6.337 \cdot 10^{-271}$)	*** ($6.993 \cdot 10^{-319}$)	*** ($3.859 \cdot 10^{-318}$)
ELF	0.982	0.988	0.977	0.982	1.23%	2.34%	*** ($9.234 \cdot 10^{-142}$)	*** ($4.309 \cdot 10^{-53}$)	

Traditional and Deep Classifiers Performance. Of the individual algorithms. Random Forest and Decision Trees achieve the best accuracy: both achieve an F-Score and AUC of 0.97, with RF performing slightly better. Linear SVM is the worst, possibly because the feature space is not linearly separable—in contrast, KNN, DT and RF can draw non-linear decision boundaries. Despite extensive hyper-parameter tuning (see Appendix), the deep learning algorithms do not achieve good performance, probably to the limited amount of training data (Section 3.2). Decision Trees achieve the lowest false negative rate (2.70%), which suggests that is largely avoids mislabeling spyware as goodware, and a relatively low false positive rate (3.08%). Random Forest achieves the lowest false positive rate (0.98%) while keeping the false negative rate also relatively low (3.70%). Keeping false positives low is important in order to prevent warnings from getting ignored. *In summary, Decision Tree is the preferred classifier if low false negatives are the priority and Random Forest is the best classifier if low false positives are the priority.*

ELF Performance. We trained ELF on the training data by trying all possible weights w_i in steps of 0.05, so that the the following constraint is preserved: $\sum_i w_i = 1$. Because we have six shallow traditional classifiers and four deep learning classifiers in the ensemble, we want to find a vector of weights $(w_1, \ldots, w_{10})$ such that assigning these weights to each classifier's predictions maximizes F_1-Score and *AUC*. After performing tuning of ELF weights with the process described in Section 3, we identify the following optimal positive weights: $w_{RF} = 0.711$, $w_{Wide\&Deep} = 0.222$, $w_{BRBM} = 0.045$, and $w_{SVM} = 0.022$ (all other weights are 0). These weights represent the relative importance of each decision algorithm in the classification, suggesting that ELF identifies Random Forest as the most important base predictor, Wide & Deep [17] as the second, followed by BRBM and SVM. The last row of Table 4 shows the performance of ELF. We observe that ELF improves all performance metrics, and minimizes the trade-off between FPR and FNR. It is also relevant to observe that while the improvement may seem minor (e.g., F_1-Score of 0.982 for ELF vs. 0.976 for RF), it is actually pretty large as the maximal performance can only go up to 1.

We also used the t-test to compute the p-values in Table 4 comparing the F_1 score of ELF against DT and RF and ELF using 10-fold CV repeated 30 times. As all the p-values are far below 0.01, the finding that ELF is superior to both RF and DT is statistically significant.

Table 5. Performance of classifiers in **spyware vs other-malware** (average with 10-fold cross-validation). The best traditional classifier is RF, which is outperformed by ELF with positive weights $w_{RF} = 0.967$ and $w_{BRBM} = 0.033$. The p-values are computed through the t-test performed against the F_1-Score results and, in addition to the raw p-values, we also use the statistical confidence notation ***, **, * (***<0.01; **<0.05; *<0.1).

Algorithm	F_1	P	R	AUC	FPR	FNR	p-val vs. DT	p-val vs. RF	p-val vs. ELF
RF	0.955	0.963	0.945	0.959	2.97%	5.35%	*** ($7.348 \cdot 10^{-51}$)		*** ($4.534 \cdot 10^{-83}$)
DT	0.949	0.945	0.953	0.954	4.60%	4.65%		*** ($2.798 \cdot 10^{-47}$)	*** ($1.041 \cdot 10^{-122}$)
KNN	0.784	0.778	0.790	0.802	18.61%	21.04%	*** (0)	*** (0)	*** (0)
LR	0.707	0.771	0.655	0.747	16.07%	34.48%	*** (0)	*** (0)	*** (0)
NB	0.416	0.745	0.290	0.604	8.23%	71.04%	*** (0)	*** (0)	*** (0)
SVM	0.445	0.996	0.287	0.643	0.09%	71.28%	*** (0)	*** (0)	*** (0)
MLP	0.120	0.645	0.067	0.518	3.14%	93.35%	*** ($3.618 \cdot 10^{-99}$)	*** (0)	*** (0)
BRBM	0.717	0.662	0.783	0.726	33.15%	21.68%	*** (0)	*** (0)	*** (0)
CNN	0.028	0.138	0.015	0.501	0.79%	98.45%	*** (0)	*** (0)	*** (0)
Wide & Deep [17]	0.074	0.653	0.040	0.418	1.92%	95.99%	*** (0)	*** (0)	*** (0)
ELF	0.960	0.966	0.954	0.963	2.74%	4.59%	*** ($4.468 \cdot 10^{-128}$)	*** ($4.534 \cdot 10^{-83}$)	

4.2 Spyware vs. Other Malware

Performance of Traditional and Deep Classifiers. Table 5 reports the performance of different traditional classifiers in separating spyware from other malware using 10-fold cross-validation. RF achieves the best performance: 0.955 F-Score and 0.959 AUC, with 2.97% false positive rate (FPR) and 5.35% false negative rate (FNR). Here false negatives mean that spyware is mislabeled as other malware which could potentially delay analysts seeking to develop signatures and patches.

Though our ability to separate spyware from other forms of malware is quite high, the results in distinguishing spyware from other malware (shown in Table 5) are slightly lower than those for distinguishing spyware vs goodware (Table 4). This is because spyware is more similar to other malware than to goodware, and hence the classifier finds it this task more challenging. For instance, some banking trojans (e.g., AceCard) may behave like spyware in order to more effectively carry out banking fraud.

Performance of ELF. We now consider the performance of ELF. As in the previous subsection, we use 10-fold cross-validation on the training data to find the weights w_i of the ensemble while requiring that $\sum_i w_i = 1$. The weights that maximize F_1-Score and *AUC* are: $w_{RF} = 0.967$ and $w_{BRBM} = 0.033$ (all other classifiers weights are 0). Note that these weights are different from the case in which we tried to separate spyware from goodware. We report the performance of ELF in the last row of Table 5. We observe that by combining the algorithms, ELF improves all performance metrics, and minimizes the trade-off between FPR and FNR. Again, while the improvement may seem minor, performance is already very high. As before, Table 4 shows the result of performing a t-test comparing ELF's F1-score and AUC to those of the best performing individual classifiers. As all the p-values are substantially below 0.01, ELF's superior performance is not due to chance, and has statistical significance.

5 DISTINGUISHING CHARACTERISTICS OF SPYWARE

The principal goal of this paper is to identify features that best separate spyware from goodware and from other malware. We do this via the *Mean Decreased Impurity* (MDI) metric. MDI is a traditional feature selection method used by Decision Trees in the Random Forest algorithm [32] which progressively splits individual features in order to separate the data more effectively. Minimal impurity is achieved when all the objects belong to a single

Table 6. Features distinguishing spyware from goodware and from other malware.

Category	Static features	Dynamic features
Spyware	-SEND_SMS -RECEIVE_SMS -READ_PHONE_STATE -READ_SMS -RECEIVE_BOOT_COMPLETED -WRITE_SMS -GET_TASKS -CALL_PHONE -CHANGE_NETWORK_STATE -more dangerous perms. -more std permissions -specific authors -PROCESS_OUTGOING_CALL -ACCESS_WIFI_STATE -WRITE_CONTACTS -ACCESS_FINE_LOCATION -ACCESS_COARSE_LOCATION -RECORD_AUDIO -WRITE_CONTACTS	
Goodware	-greater filesize -more components	-more read /dev/ -more crypto operations
Other Malware	-SYSTEM_ALERT_WINDOW -MOUNT_UNMOUNT_FILESYSTEMS -WRITE_EXTERNAL_STORAGE	-write .dex classes -load .dex classes -servicestart AdminService

label. For example, the presence or absence of a certain permission (e.g., READ_PHONE_STATE) may be used by DTs to split feature vectors into two separate groups to decrease impurity. We consider the traditional version with the *Gini* definition of impurity [32]. We use MDI scores to build feature-value histograms that show the distributions of feature values for the two classes (e.g. spyware vs. goodware or spyware vs. other malware). The Appendix also shows the most relevant feature histograms and decision trees which support our characterization of spyware.

5.1 Spyware vs. Goodware and Other Malware

Table 6 shows a summary of the features which best separate spyware from goodware and from other malware (the Appendix reports the full feature histograms). Each group-row in Table 6 corresponds to a different category (i.e., spyware, goodware, other malware), and the presence of a static (resp. dynamic) features implies a prevalence of that feature in that category, and its absence (or lower prevalence) from the other ones. We can observe that:

- The feature SEND_SMS is primarily prevalent in spyware and less present in goodware and other malware.
- Spyware apps tend to have a smaller filesize and fewer Android components than goodware. This may indicate that most spyware are not repackaged versions of goodware apps. They could be developed as ad-hoc malware to steal user information or may be repackaged versions of small apps.
- Static features (specifically requests for Android permissions) play an important role in separating spyware from other categories. The relevance of "permission" features is related to the fact that Android applications need to ask for permission in advance in order to access certain resources (e.g. software or hardware). Some of the most relevant permissions are related to: accessing sensitive information (e.g., READ_PHONE_STATE, GET_TASKS) and sending information to attackers (e.g., SEND_SMS, CHANGE_NETWORK_STATE).

Table 7. AceCard distinguishing features.

Category	Static features	Dynamic features
AceCard Family	-READ_SMS -SEND_SMS -RECEIVE_SMS -RECEIVE_BOOT_COMPLETED -GET_TASKS -READ_PHONE_STATE -CALL_PHONE -WRITE_SMS -specific authors -SYSTEM_ALERT_WINDOW -ACCESS_NETWORK_STATE -READ_LOGS -more background services	-write AppPrefs.xml -write new.apk -read/write JAR files -loading .dex classes
Other Spyware	-ACCESS_WIFI_STATE -ACCESS_FINE_LOCATION -ACCESS_COARSE_LOCATION -PROCESS_OUTGOING_CALLS -READ_HISTORY_BOOKMARKS -CAMERA -more dangerous perms.	
Goodware		-more read operations -more write operations
Other Malware	-MOUNT_UNMOUNT_FILESYSTEMS -WRITE_EXTERNAL_STORAGE -more std perms.	

- One of the main permissions that other types of malware request is the SYSTEM_ALERT_WINDOW permission, which allows displaying windows on top of the screen. For example, this permission can be used by mobile ransomware in order to block access to devices or to use hidden overlays for privilege escalation [64]. This permission also has legitimate uses such as showing Facebook chat notifications. Spyware requires this permission much less frequently, but other malware often use this permission to keep windows hidden from the user.
- Finally, it is interesting to observe that none of the Android spyware samples starts an Android Service named AdminService, whereas other types of malware do—likely to start stealthy background processes with a name that looks legitimate to users.

5.2 Analysis of 5 Android Spyware Families

In this section, we provide an in-depth view of the behavior of 5 major Android malware families: AceCard, HeHe, Pincer, UaPush, and USBCleaver. A more comprehensive analysis with feature histograms is reported in the Appendix.

5.2.1 AceCard family. Like many malware samples, AceCard [42] behaves in a benign fashion during an initial phase and then turns malicious. In this second phase, it masquerades as a system update package. Once executed, it asks the user to authorize device administrator permissions in order to allow it to run every time the device restarts. It shows fake login prompts in order to steal information from users. In a third phase, it launches attacks against a vast number of devices. AceCard's spyware capabilities enable it to also function as a banking trojan.

Table 8. HeHe distinguishing features.

Category	Static features	Dynamic features
HeHe Family	-MOUNT_UNMOUNT_FILESYSTEMS -MOUNT_FORMAT_FILESYSTEMS -SYSTEM_ALERT_WINDOW -GET_TASKS -CHANGE_CONFIGURATION -READ_PHONE_STATE -more dangerous perms. -ACCESS_WIFI_STATE - specific authors - more std perms. -CHANGE_NETWORK_STATE -CALL_PHONE -WAKE_LOCK -WRITE_EXTERNAL_STORAGE -more background services -ACCESS_COARSE_LOCATION -ACCESS_LOCATION_EXTRA_COMMANDS	-write `.xml` file -read `cpuinfo`
Other Spyware	-WRITE_SMS -RECEIVE_SMS -SEND_SMS -READ_SMS -RECEIVE_BOOT_COMPLETED	
Goodware	-larger filesize -more `Activities`	
Other Malware		

Table 7 presents the most relevant features that separate AceCard from goodware, other spyware and other (non-spyware) malware. The detailed feature histograms are reported in the Appendix. AceCard requests permissions to read and receive SMSs—these are factors rarely required by goodware. However, because the ability to access SMS-related functionalities is required by many malware and spyware, when we compare AceCard with other malware and other spyware, SMS activities become less important (and are not in the top 15 features anymore), and the ability to read/write various files such as XML, JAR and APKs becomes more important. For instance, AceCard samples write a configuration file named `AppPrefs.xml`—with an intentionally innocuous filename. In addition, AceCard writes some JAR and APK files (e.g., `read ybfhjzlav.jar`), and then loads some of them as classes (e.g., `dexclass ybfhjzlav.jar`). These suspicious signals, automatically identified by our classifiers, may suggest that the malware is unpacking malicious code [54], i.e., additional code that was intentionally encrypted or obfuscated to prevent static analysis. Finally, AceCard also uses a large number of background services (see `num_services` features), which may be used for resilience of the spyware and to keep collecting information when the malicious app is out of focus and the user is not using it directly.

5.2.2 HeHe family. The HeHe spyware family [23] consists of apps that hide their icons after installation. Under the guise of updating Android security, HeHe samples track SMSs and phone calls coming from an attacker-defined set of numbers of interest (e.g., banks) and suppress them so the user is not aware of the alerts that might thus be generated; these numbers are also sent back to the attacker's command and control server.

Table 8 presents a summary of the most distinguishing features that separate HeHe from goodware, other spyware and other (non-spyware) malware. Detailed features histograms are reported in the Appendix.

Table 9. UaPush distinguishing features.

Category	Static features	Dynamic features
UaPush Family	-READ_PHONE_STATE -SYSTEM_ALERT_WINDOW -ACCESS_WIFI_STATE -GET_TASKS -more std perms. -specific authors -ACCESS_COARSE_LOCATION -MOUNT_UNMOUNT_FILESYSTEMS -WRITE_EXTERNAL_STORAGE -RECEIVE_BOOT_COMPLETED -VIBRATE -more background services -ACCESS_NETWORK_STATE -READ_LOGS	-read `cpuinfo` -read `meminfo` -crypto operations with fixed keys -write specific `xml` file -write in `/data/` path
Other Spyware	-RECEIVE_SMS -READ_SMS -SEND_SMS -WRITE_SMS -num. dangerous perms.	
Goodware	-greater filesize	
Other Malware		

HeHe samples are different from goodware mainly in terms of requested permissions: to get active user tasks, to control system alert windows (e.g., also prevent their appearance), and to change system configurations. The number of Android Activities is slightly lower than goodware and most spyware—this suggests that some HeHe samples may hide in apps without many Activities. This is in line with the fact that HeHe hides its presence from the phone, so it actually does not need to implement many Activities (i.e., screens) in the code. When compared to other spyware, HeHe is also distinguished by its request to change configuration files and unmount the file system; it also requires less permissions that are deemed as dangerous by the Android official documentation, suggesting it wants to hide its traces.

5.2.3 UaPush family. The UaPush spyware family starts its malicious activity when the device is rebooted or an active network connection is detected. It steals personal information (call history, bookmarks, contacts) and tends to send outgoing SMS messages. Moreover, it displays advertisements in notification bars and/or alert windows. Table 9 presents the most important features that separate UaPush from goodware, other spyware and other (non-spyware) malware. Detailed histograms are in the Appendix.

UaPush is different from both AceCard and HeHe in the features used to conduct malicious activity. It is distinguishable from goodware because its filesize is smaller and it requests a higher number of permissions than most goodware. Moreover, it explicitly requests dangerous permissions such as `GET_TASKS` and `READ_LOGS` explicitly. In contrast to other spyware and other malware, UaPush samples encrypt and/or write specific files—this is possibly related to unpacking operations [54].

5.2.4 Pincer family. The Pincer spyware family pretends to be a certificate by often having the name *Certificate.apk*. Once installed, it intercepts and forwards SMS messages from the phone. In addition, it ships assorted information about the infected phone to a command and control server. This information includes the phone number, the international mobile subscriber identifier, the Android version, and more.

Table 10. Pincer distinguishing features.

Category	Static features	Dynamic features
Pincer Family	-RECEIVE_SMS -CALL_PRIVILEGED -MODIFY_PHONE_STATE -fixed num. dangerous perms. -SEND_SMS -RECEIVE_BOOT_COMPLETED -READ_LOGS -CALL_PHONE -READ_PHONE_STATE -higher num. Receivers -specific authors -small filesize	-write specific dex/apk files -read specific dex/apk files -load specific dex/apk files -write specific xml files -start specific services
Other Spyware	-higher num. perms. -higher num. dangerous perms. -ACCESS_NETWORK_STATE -GET_TASKS -WRITE_SMS	
Goodware	-ACCESS_NETWORK_STATE -WAKE_LOCK	
Other Malware	-SYSTEM_ALERT_WINDOW	

Table 10 presents histograms of the most important features that separate Pincer from goodware, other spyware and other (non-spyware) malware. Detailed histograms are in the Appendix.

Pincer is different from AceCard, HeHe and UaPush in that it requires privileged access to call, and starts many background services (e.g., *CheckQueueService*, *CheckCommandsService*, *ReceiverRegisterService*). The abundance of background services is probably for resilience of the malware. This way, even if the user detects and deletes one of these background services, the others remain active. The many *write* and *dexclass* operations suggest unpacking of malicious code [54], which is then loaded in memory. We also observe that the average filesize of Pincer samples is smaller than that of both goodware and other malware samples, suggesting that it does not contain any repackaged app, but instead starts many background services to achieve its monitoring purposes.

5.2.5 USBCleaver family. The USBCleaver family consists of apps that, once downloaded onto a phone, import malware from a command and control server that allows it to infect a computer to which the device is connected via USB. Phones are commonly attached to PCs for legitimate reasons (e.g., saving pictures, or recharging the battery). The result is that the malware can monitor and intercept various browser passwords, WiFi passwords, and more.

Table 11 presents the most important features that separate USBCleaver from goodware, other spyware and other (non-spyware) malware. Detailed histograms are in the Appendix.

USBCleaver is different from goodware because it requests permissions to read and execute dexclass operations on various APKs, probably for unpacking malicious code [54]. However, a major distinguishing factor is that it performs almost no read operations on the phone itself, but instead requests permissions to write on external storage devices, possibly because this is the main mechanism used by it to propagate. To stay off the radar, its filesize and number of permissions (including dangerous ones) are kept to a minimum, both with respect to goodware as well as to other spyware and other malware. In addition, the number of components (activities, services, intents) of the app is very small, suggesting that this spyware family does not repack other benign

Table 11. USBCleaver distinguishing features.

Category	Static features	Dynamic features
USBCleaver Family	-specific authors -small filesize -WRITE_EXTERNAL_STORAGE -few std perms. requested -few dang. perms. requested -READ_PHONE_STATE	-read specific apk/dex files -load specific apk/dex files -few read operations
Other Spyware	-RECEIVE_BOOT_COMPLETED -SEND_SMS -RECEIVE_SMS -READ_SMS -WRITE_SMS -ACCESS_FINE_LOCATION -READ_PHONE_STATE	
Goodware	-WAKE_LOCK -num. std perms. -num. dangerous perms. -VIBRATE -READ_EXTERNAL_STORAGE -GET_ACCOUNTS	-read /dev/ path
Other Malware	-RECEIVE_BOOT_COMPLETED -SYSTEM_ALERT_WINDOW -GET_TASKS -CHANGE_NETWORK_STATE -high num. receivers	

applications (e.g., games). USBCleaver also requires permission to READ_PHONE_STATE, possibly to infer when to start acting.

5.2.6 Family Comparison. We now compare and contrast the most important features of the 5 families evaluated in this section. Table 12 reports static and dynamic feature in rows, where features are also qualitatively grouped in sub-groups (e.g., *Permission (SMS)*). The *v-tick symbol* (✓) indicates that a feature plays an important role in distinguishing that family from others.

We observe that all spyware families, with the exception of USBCleaver, request a high number of permissions. This likely happens because it is much harder to get root access to the device [21] than to request more permissions from users, as the latter may go unnoticed upon installation. We observe that all families request the READ_PHONE_STATE permission, which is required by all spyware to keep track of the phone state (e.g., in-use or locked), so to keep acting as stealthily as possible. For each family, the variants often re-used the same set of certificates to sign the application (thus connecting them to the same author). This can be useful in recognizing variants of the same family.

There are also some important differences in the most relevant permissions requested by these 5 families. AceCard and Pincer request permission to access, alter and send SMSs, which is associated with both premium SMS activities and, for AceCard, hybrid banking-trojan characteristics (e.g., to steal bank tokens). HeHe and UaPush request permissions to access device locations. AceCard, Pincer and USBCleaver are the only ones characterized by dynamic class loading operations, which may correspond to runtime unpacking and de-obfuscation operations. AceCard, HeHe and UaPush define a high number of background services, which are likely used to silently siphon sensitive data from the phone of a victim.

Table 12. Comparison of the most distinguishing features of the 5 evaluated families. The v-ticks (✓) highlight prevalent features in each family.

			AceCard	HeHe	UaPush	Pincer	USBCleaver
Static	Permissions (SMS)	READ_SMS	✓				
		RECEIVE_SMS	✓			✓	
		SEND_SMS	✓			✓	
		WRITE_SMS	✓				
	Permissions (Calls)	CALL_PHONE	✓	✓		✓	
		CALL_PRIVILEGED				✓	
	Permissions (Storage)	WRITE_EXTERNAL_STORAGE		✓	✓		✓
		MOUNT_UNMOUNT_FILESYSTEMS		✓	✓		
		MOUNT_FORMAT_FILESYSTEMS		✓			
	Permissions (Location)	ACCESS_COARSE_LOCATION		✓	✓		
		ACCESS_LOCATION_EXTRA_COMMANDS		✓			
	Permissions (System)	SYSTEM_ALERT_WINDOW	✓	✓	✓		
		RECEIVE_BOOT_COMPLETED	✓		✓	✓	
		VIBRATE			✓		
		CHANGE_CONFIGURATION		✓			
		READ_LOGS	✓		✓	✓	
		ACCESS_WIFI_STATE		✓	✓		
		ACCESS_NETWORK_STATE	✓		✓		
		CHANGE_NETWORK_STATE		✓			
		MODIFY_PHONE_STATE				✓	
		READ_PHONE_STATE	✓	✓	✓	✓	✓
		GET_TASKS	✓	✓	✓		
		WAKE_LOCK		✓			
	Stats	specific authors	✓	✓	✓	✓	✓
		very small filesize				✓	✓
		high num. std perms.		✓	✓		
		high num. dangerous perms.		✓			
		low num. std/dangerous perms.		✓			✓
		high num. Receivers				✓	
		high num. background Services	✓	✓	✓		
Dynamic	Dynamic Class Loading	read specific apk/dex files				✓	✓
		write specific apk/dex files	✓			✓	
		load specific apk/dex files	✓			✓	✓
	Read Operations	low num. read operations					✓
		read cpuinfo		✓	✓		
		read meminfo			✓		
	Write Operations	write specific xml files	✓	✓	✓	✓	
		write in /data/ path			✓		
	Other	start specific background Services				✓	
		crypto operations with specific keys			✓		

There are also some relevant characteristics unique to each of the families. AceCard is the only one that simultaneously requests all SMS permissions, permission to control system alert windows, and permission to handle phone calls—all probably to carry out its hybrid banking-trojan activities while hiding its tracks. HeHe requests the highest number of permissions deemed "dangerous" by the Android documentation. UaPush is the only family out of these 5 that performs some sort of cryptographic operation, which may be used to encrypt sensitive data before transmission. Pincer is the only family that starts background Android services with specific names, shared among its spyware variants. USBCleaver has the lowest number of requested permissions, and performs a very low number of read operations, suggesting limited functionality apart from the installation of the malware on a victim's laptop connected via USB.

6 RELATED WORK

We organize the related work into 3 broad categories: information leakage detection, malware characterization, and malware detection and family identification.

6.1 Information Leakage Detection

Several efforts focus on detecting exfiltration of sensitive information by Android applications through two major techniques: *taint data flow tracking* [11, 26, 35, 61, 78], and—more recently—*network traffic analysis* [18, 38, 46, 50, 51, 65, 71, 82]. Taint data flow tracking systems (e.g. FlowDroid [11], TaintDroid [26] and DroidSafe [35]) use either static analysis or dynamic analysis to detect whether potentially sensitive information (e.g., location, call logs) is transmitted to some sink (e.g., the network or SMS) during code execution, indicating a possible information leak. However, attackers can evade such solutions by breaking the information flow [18] — these solutions also suffer from high false positives. More recent literature has focused on network traffic analysis to identify possible leaks: initial work [46, 50, 65, 71] has assumed plaintext communications or slightly obfuscated traffic, whereas the AGRIGENTO [18] method uses black box differential analysis which is robust to many obfuscation techniques. Some other papers [25, 43] perform a static and dynamic analysis of browser extensions to detect information leakage, but their approach is specifically targeted to PC browser extensions leaking sensitive information, and they are not directly applicable Android or non-browser based approaches.

The main difference between our paper and these works is that most papers on information leakage analysis do not distinguish between benign and malicious applications, and try to detect exfiltrated information even if the exfiltration is legitimate (e.g. shipping location information to Google Maps). To distinguish between legitimate and malicious data transmission, these past efforts would need to compare observed app behavior against user expectations or (when available) against app privacy policies [70, 85]. On the other hand, our work automatically detects modern Android spyware without the need to analyze privacy policies or user expectations. Moreover, one of our main objectives is also to *characterize* modern Android spyware, and to study how it differs from other malware and from goodware.

6.2 Malware Characterization

Another body of related work tries is on malware behaviors. Some solutions focus on traditional PC malware and study how malware spreads over the Internet [40, 80]. Farley et al. [30] study how malware can hijack microphones on Windows and Mac OS X in order to spy and gather sensitive information. An interesting survey on data leakage strategies is [69], but it does not consider the Android ecosystem and is not focused on spyware. A preliminary discussion on spyware is proposed in [68], but no experiments were performed. Other work on Android malware evaluates infection rates and risk indicators [76], identifies malware presence on third-party Android markets [49]—but these efforts do not try to characterize the behavior of spyware. [48] characterizes the difference between malware and benign apps, but does no prediction and does not focus on spyware and

only considers samples from June 2012 to June 2014. Finally, other works [36, 57] rely on existing taint analysis solutions to study which information is disclosed by advertisement libraries, while our focus is on data-driven detection and characterization of modern Android spyware.

6.3 Malware Detection and Family Identification

Traditional work on Android malware analysis has two main objectives: *malware detection* and *family identification*. The first problem investigates whether a given Android application is benign or malicious, while the second problem identifies the family to which a given malware sample belongs. Some malware detection methods include DREBIN [10], CrowDroid [13], DroidAPIMiner [7], DroidScope [79], MARVIN [47], DroidInjector [29], Milosevic et al. [56], Zhang et al. [81] and MaMaDroid [53]. Our paper differs from these efforts in two ways: (i) We focus on spyware while they cover all malware. For example, [56, 82] proposes an interesting computationally inexpensive way to use a machine learning aided approach for static analysis of general malware detection, but they do not focus on spyware detection and characterization. Another example [82] which proposes a model to analyze dynamic app behavior to detect stealthy malware activities. The intuition is that malware activities cannot be mapped to legitimate triggers; while this model is interesting and novel, it does not consider detection and characterization of Android spyware.

Papers considering family identification of an Android malware sample include DroidLegacy [22], DroidSIFT [83], DenDroid [73]. These papers seek to classify a sample into a *specific* family (e.g., `DroidKungFu`, `Geinimi`), whereas we are interested in extracting the characteristics of modern *spyware*. Moreover, they often consider outdated datasets. A more recent effort [15] looks at predicting the families to which malware belongs, but does not study the characteristics of spyware.

Other papers related to our Android spyware characterization include [8], [9], [16] and [39]. Andronio et al. [8] propose a method to discriminate between goodware, scareware and ransomware by generalizing three key common behaviors of mobile ransomware: threatening text, device locking, and encryption. These features were identified after manual reverse engineering of some samples, whereas our approach is driven by machine learning over a large number of samples and is hence scalable. Aresu et al. [9] propose an approach that extracts features from HTTP traffic in order to classify mobile botnets variants (e.g., `Zitmo`). Unlike these papers, we focus specifically on Android spyware, and, in addition to detection, we also consider automated data-driven understanding of modern Android spyware characteristics. Chatterjee et al. [16] design, implement, and evaluate a measurement pipeline that combines web and app store crawling with machine learning to find and label spyware used in Intimate partner spying (IPS). Their definition of spyware includes both overtly malicious apps and dual-use apps i.e. apps that have a legitimate use but can be easily used for spying on a partner. Unlike our paper, [16] only used features extracted from information available on the Google Play page. Javaheri et al. [39] propose a method for spyware classification that is based on a dynamic behavioral analysis through deep and transparent hooking of kernel-level routines. The authors explored the use of linear regression, JRIP, and J48 decision tree algorithms as a classifier and were able to achieve an accuracy of $\approx 93\%$ and an error rate near 7%.

6.4 Summary of Related Work

In summary, we differ from previous literature for the following reasons. To the best of our knowledge, (i) we are the first to compare the performance of several state-of-the-art shallow and deep learning classifiers for the task of identifying Android spyware, and we are the first to show that an ensemble late fusion leads to the best 10-fold performance in distinguishing spyware from goodware and from other categories of malware; (ii) instead of considering the traditional goodware vs. malware separation problem, we characterize spyware by examining the differences between spyware vs. goodware and spyware vs. other-malware, and also focus on 5 major spyware families; (iii) we show how machine learning can be used to explain behavior of a spyware, whereas other efforts

focused on spreading models or distribution of generic malware in Android marketplaces; (iv) we consider recent spyware samples found in the wild between July 1st 2016 and July 1st 2017, whereas most papers considered outdated datasets [10, 84].

7 CONCLUSIONS

7.1 Summary of Main Findings

This section summarizes the main findings of our characterization of modern Android spyware.

First, we find that Android spyware tends to explicitly request the permissions needed to access sensitive information within the manifest of the app (e.g., phone state, calls, locations, active tasks). Spyware also tends to ask for permissions for network operations, sending SMSs, and making phone calls. These permissions may be used to transfer data and communicate with the attacker. The fact that spyware requests these permissions directly suggests that most Android spyware does not rely on gaining root access to the device, which is perfectly reasonable because getting root access on updated Android devices is extremely hard and has low chances of success [21]. In contrast, asking uninformed users for more permissions is more likely to succeed. Moreover, Android spyware's filesize tends to be under 2MB, i.e., it is smaller than most goodware. This indicates that spyware is likely developed as an ad-hoc app or as a repackaged version of small apps.

Our results indicate that we obtain excellent predictive performance using lightweight static and dynamic features. More specifically, our Ensemble Late Fusion (ELF) method obtains 0.982 F_1-Score and 0.982 AUC for spyware vs. goodware and 0.960 F_1-Score and 0.963 AUC for spyware vs. other malware, outperforming all traditional and deep learning classifiers. This slight difference in accuracy (which is more evident if other traditional supervised classification algorithms are considered) indicates that the differences between spyware and goodware are greater than those between spyware and other malware types. This is reasonable since spyware, like other malware, performs malicious activities.

The most important feature that helps distinguish spyware from goodware is the SEND_SMS permission: 50% of our spyware samples request that permission, whereas less than 5% of goodware samples ask for it. This permission can be used to leak information in the absence of network connections [48],to steal money from users by sending SMSs to premium rate numbers.[6]

The permission CALL_PHONE is the second most important feature that distinguishes spyware from other malware. This may indicate that spyware uses this permission to control/prevent phone calls, or to inform attackers that a phone has been compromised, or possibly to call premium rate numbers in order to siphon off funds from users without their consent.

By analyzing 5 spyware families using machine learning and comparing each family against goodware, other spyware, and other malware, we were able to automatically infer some interesting distinguishing characteristics. All 5 families have lower filesize than goodware, but some spyware families (e.g. UaPush and Pincer) have filesizes comparable to other malware. Moreover, all 5 families usually require at least as many permissions as goodware with the exception of USBCleaver, which requires only few permissions. As its main goal is to propagate to Windows devices via USB access, USBCleaver mainly requires permissions to access external storage. AceCard, Pincer and USBCleaver also perform many read/write/dexclass operations on APKs with randomized names which may indicate unpacking activities [54]. In addition to these activities, UaPush differs by performing certain cryptographic operations which are not usually performed by other spyware and other malware that we studied. A distinguishing feature of Pincer is that it starts many background services in order to (probably) hide its activities and keep monitoring the device. Moreover, unlike goodware and other malware,

[6] Although sending premium SMSs is not a primary objective of many spyware, some of them may still perform this activity to increase income [48]

all 5 families have a small number of app components, indicative of apps that do not offer strong functionalities and that hide their presence after installation (e.g., HeHe).

In addition to providing insights into modern Android spyware, the characterization methodology presented in this paper is general enough to be used by security practitioners to perform data-driven characterization of other spyware families as they appear in the wild as well as other categories of malware, such as ransomware and premium-SMS senders.

7.2 Limitations of Static and Dynamic Analyses

The types of *program analysis* techniques to analyze an application can be roughly divided into *static* and *dynamic* analysis. Static analysis evaluates an app without executing it, and offers an *overapproximation* of its behavior, as some paths may never be executed; however, it is very weak against code obfuscation strategies. Dynamic analysis is more robust against code obfuscation and generally offers an *underapproximation* of an application behavior, as it is not feasible to dynamically execute all possible paths, and some behaviors may be triggered only under some specific attacker-defined conditions (e.g., time- and logic-bombs). This is why a combination of static and dynamic analysis is usually preferred in order to maximize coverage and increase costs for the attacker.

The dynamic analysis we perform in this work through the Koodous [3] online service does not cover all possible execution traces. In particular, Koodous relies on the Droidbox sandbox [2], executes samples for 60 seconds, and collects any system and network activity detected during execution. Since the malware may detect that it is running in a sandbox and/or because its behavior may be triggered only by certain events (e.g., initiated by the attacker's command and control server [52]), some dynamic behavior may be missed in this paper. Very recent work (e.g., [29, 33, 60]) study how to trigger malicious behaviors of malware by simulating user interactions, but [66] shows that they are imperfect and do not overcome random input generation, which is suboptimal as it cannot simulate complex user interactions. Hence, the problem remains an open issue and future work could look at integrating these approaches into dynamic analysis.

Nevertheless, we note that since we rely on both static and dynamic analyses we are able to identify static indicators of maliciousness even if the app does not trigger the malicious behavior when executed. This is confirmed in our characterization of the 5 spyware families, where for some families static features are more relevant in describing and capturing them. While our feature space with ELF remains able to detect the majority of the spyware applications, more advanced dynamic analysis techniques that are robust to anti-detection mechanisms may improve the quality of the characterization but do not compromise the validity of our results.

It is worth observing that about 20% of our samples crashed during dynamic analysis on Koodous and therefore are not included in the analysis in this paper. While this could indicate that our study is not complete, this limitation is common in prior work that uses dynamic analysis [15, 20, 59, 62]. Some strategies to prevent crashing of samples may be worth investigating in future work.

7.3 Future Work

In this paper, we characterize spyware with respect to goodware and to other malware. We rely on machine learning features derived from lightweight static and dynamic analysis of Android samples. In future work, it would be interesting to consider additional techniques to gain more insights about spyware. Examples of such techniques include (i) designing new features to represent sequence and timing of dynamic operations; (ii) integrating dynamic analysis techniques for detecting which specific information is likely leaked the channels (e.g. network, SMS, files on disk) through which such leaks occur; (iii) performing in-depth inspections of application source code and control-flow to link spyware behavior to the code; (iv) considering an adversarial model in which the adversaries tampers with our training data (poisoning) or performs test-time evasion attacks.

Since the main focus of this work is on characterization, we operated in a setting without concept drift; future work may use collective classification to leverage dependencies between samples [45] and to evaluate the impact of non-stationarity on detection of spyware [63]. Our study has also shown that some spyware families perform additional malicious activities, such as sending premium SMS or stealing banking credentials. Another interesting research direction is to explore *multi-label classification*—for instance, AceCard is both spyware and a banking trojan, and a system that can automatically predict a *set* of labels for each app would be useful for capturing multiple malicious behaviors at once.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their suggestions, and VirusTotal for granting us academic access to their Intelligence API. This work was supported in part by the Office of Naval Research via grants N00014-15-1-2007 and N00014-16-1-2896, the Army Research Office under grant W911NF1410358, and the North Atlantic Treaty Organization (NATO) Science for Peace and Security (SPS) programme under grant G5319.

REFERENCES

[1] [n.d.]. https://www.virustotal.com/.
[2] [n.d.]. https://github.com/pjlantz/droidbox.
[3] [n.d.]. Koodous. https://koodous.com/. Accessed: July 2017.
[4] [n.d.]. Whaling emerges as major cybersecurity threat. https://www.cio.com/article/3059621/security/whaling-emerges-as-major-cybersecurity-threat.html.
[5] 2017. McAfee Mobile Threat Report [Internet]. https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2017.pdf.
[6] 2017. Kaspersky IT Threat Evolution Statistics [Internet]. https://securelist.com/it-threat-evolution-q1-2017-statistics/78475/.
[7] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. DroidAPIMiner: Mining API-level features for robust malware detection in android. In *SecureComm*. Springer.
[8] Nicoló Andronio, Stefano Zanero, and Federico Maggi. 2015. HelDroid: dissecting and detecting mobile ransomware. In *RAID*. Springer.
[9] Marco Aresu, Davide Ariu, Mansour Ahmadi, Davide Maiorca, and Giorgio Giacinto. 2015. Clustering Android malware families by HTTP traffic. In *10th Int. Conf. on Malicious and Unwanted Softw. (MALWARE)*. IEEE.
[10] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In *NDSS*.
[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2013. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM PLDI*.
[12] Battista Biggio, Konrad Rieck, Davide Ariu, Christian Wressnegger, Igino Corona, Giorgio Giacinto, and Fabio Roli. 2014. Poisoning behavioral malware clustering. In *AISec Workshop*. ACM.
[13] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: behavior-based malware detection system for android. In *SPSM*. ACM, Chicago,IL,USA, 15–26.
[14] Carlos Castillo. 2016. Android Banking Trojan Asks for Selfie with your ID. (2016). https://securingtomorrow.mcafee.com/mcafee-labs/android-banking-trojan-asks-for-selfie-with-your-id/
[15] Tanmoy Chakraborty, Fabio Pierazzi, and VS Subrahmanian. 2017. EC2: Ensemble Clustering and Classification for Predicting Android Malware Families. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2017).
[16] Rahul Chatterjee, Periwinkle Doerfler, Hadas Orgad, Sam Havron, Jackeline Palmer, Diana Freed, Karen Levy, Nicola Dell, Damon McCoy, and Thomas Ristenpart. 2018. The Spyware Used in Intimate Partner Violence. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, 441–458. https://doi.org/10.1109/SP.2018.00061
[17] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. ACM, 7–10.
[18] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *NDSS*. https://doi.org/10.14722/ndss.2017.23465
[19] Nokia Corp. 2017. *Nokia Threat Intelligence Report 2017*. https://pages.nokia.com/18259.threat.intelligence.report.lp.html
[20] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. 2016. DroidScribe: Classifying android malware based on runtime behavior. In *IEEE Security and Privacy Workshops (SPW)*.

[21] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. 2010. Privilege escalation attacks on android. In *International Conference on Information Security*. Springer, 346–360.

[22] Luke Deshotels, Vivek Notani, and Arun Lakhotia. 2014. Droidlegacy: Automated familial classification of android malware. In *Proc. Program Protection and Reverse Engineering Workshop*. ACM.

[23] H. Dharmdesani. 2014. Android.HeHe: Malware now Disconnects Phone Calls. (2014). https://www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.html

[24] Marko Dimjašević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. 2016. Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*. ACM, 1–8.

[25] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. 2007. Dynamic spyware analysis. (2007).

[26] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS* (2014).

[27] F-Secure Corp. [n.d.]. Report. https://www.f-secure.com/weblog/archives/00002573.html.

[28] F-Secure Corp. 2013. Trojan:Android/Pincer.A. https://www.f-secure.com/weblog/archives/00002538.html.

[29] Wenhao Fan, Yaohui Sang, Daishuai Zhang, Ran Sun, and Yuan'an Liu. 2017. DroidInjector: A process injection-based dynamic tracking system for runtime behaviors of Android applications. *Computers & Security* 70 (2017), 224–237.

[30] Ryan Farley and Xinyuan Wang. 2010. Roving BugNet: Distributed surveillance threat and mitigation. *Computers & Security* (2010).

[31] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. 1996. A sense of self for Unix processes. In *IEEE Symp. Security and Privacy*.

[32] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2001. *The Elements of Statistical Learning*. Springer series in statistics New York, NY, USA:.

[33] Andrea Gianazza, Federico Maggi, Aristide Fattori, Lorenzo Cavallaro, and Stefano Zanero. 2014. Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications. *Tech Report* (2014).

[34] Google. 2017. Android Security 2016 Year in Review, Tech. Report.

[35] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information-Flow Analysis of Android Applications in DroidSafe. In *NDSS*. Internet Society. https://doi.org/10.14722/ndss.2015.23089

[36] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. ACM Press, 101. https://doi.org/10.1145/2185448.2185464

[37] Info-security magazine. [n.d.]. Exaspy, a New Android Spyware, Targets Execs . https://www.infosecurity-magazine.com/news/exaspy-a-new-android-spyware/. Accessed: Aug 2017.

[38] Sushil Jajodia, Noseong Park, Fabio Pierazzi, Andrea Pugliese, Edoardo Serra, Gerardo I Simari, and VS Subrahmanian. 2017. A Probabilistic Logic of Cyber Deception. *IEEE Transactions on Information Forensics and Security* (2017).

[39] Danial Javaheri, Mehdi Hosseinzadeh, and Amir Masoud Rahmani. 2018. Detection and Elimination of Spyware and Ransomware by Intercepting Kernel-Level System Routines. *IEEE Access* 6 (2018), 78321–78332. https://doi.org/10.1109/ACCESS.2018.2884964

[40] Chanhyun Kang, Noseong Park, B Aditya Prakash, Edoardo Serra, and Venkatraman Sivili Subrahmanian. 2016. Ensemble Models for Data-driven Prediction of Malware Infections. In *WSDM*. ACM, San Francisco, CA, USA.

[41] Kaspersky. [n.d.]. Criminals Blackmail Users with Sensitive Information. https://www.kaspersky.com/blog/beware_sextortion/5796/. Accessed: Aug 2017.

[42] Kaspersky Labs. 2016. Android Trump Card: AceCard. https://www.kaspersky.com/blog/acecard-android-trojan/11368/.

[43] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. 2006. Behavior-based Spyware Detection. In *USENIX Security*.

[44] Koodous. [n.d.]. Droidbox. https://docs.koodous.com/yara/droidbox/. Accessed: July 2017.

[45] Eric Lancaster, Tanmoy Chakraborty, and V.S. Subrahmanian. 2018. MALT^P: Parallel Prediction of Malicious Tweets. *IEEE Transactions on Computational Social Systems* 5, 4 (2018), 1096–1108.

[46] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. 2015. AntMonitor: A System for Monitoring from Mobile Devices. ACM Press, 15–20. https://doi.org/10.1145/2787394.2787396

[47] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. 2015. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *IEEE COMPSAC*.

[48] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. 2014. ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. *Proc. Int. Workshop BADGERS* Sep (2014). https://doi.org/10.1109/BADGERS.2014.7

[49] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. 2014. AndRadar: Fast discovery of android applications in alternative markets. In *DIMVA*. Springer, Egham, UK, 51–71.

[50] Yabing Liu, Han Hee Song, Ignacio Bermudez, Alan Mislove, Mario Baldi, and Alok Tongaonkar. 2015. Identifying Personal Information in Internet Traffic. ACM Press, 59–70. https://doi.org/10.1145/2817946.2817947

[51] Mirco Marchetti, Fabio Pierazzi, Michele Colajanni, and Alessandro Guido. 2016. Analysis of high volumes of network traffic for Advanced Persistent Threat detection. *Computer Networks* (2016).

[52] Mirco Marchetti, Fabio Pierazzi, Alessandro Guido, and Michele Colajanni. 2016. Countering Advanced Persistent Threats through security intelligence and big data analytics. In *IEEE International Conference on Cyber Conflict (CyCon)*.

[53] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2017. MaMaDroid: Detecting android malware by building markov chains of behavioral models. *NDSS* (2017).

[54] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. 2007. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 431–441.

[55] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. 2016. Reviewer integration and performance measurement for malware detection. In *DIMVA*. Springer.

[56] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. 2017. Machine learning aided Android malware classification. *Computers & Electrical Engineering* 61 (July 2017), 266–274. https://doi.org/10.1016/j.compeleceng.2017.02.013

[57] Suman Nath. 2015. MAdScope: Characterizing Mobile In-App Targeted Ads. ACM Press, 59–73. https://doi.org/10.1145/2742647.2742653

[58] D. Oktavianto and I. Muhardianto. 2013. *Cuckoo Malware Analysis*. Packt Publishing. https://books.google.com/books?id=KXNZAQAAQBAJ

[59] Lucky Onwuzurike, Mario Almeida, Enrico Mariconti, Jeremy Blackburn, Gianluca Stringhini, and Emiliano De Cristofaro. [n.d.]. A Family of Droids–Android Malware Detection via Behavioral Modeling: Static vs Dynamic Analysis. In *16th Annual Conference on Privacy, Security and Trust 16th Annual Conference on Privacy, Security and Trust (PST 2018)*.

[60] Xiaorui Pan, Xueqiang Wang, Yue Duan, Xiaofeng Wang, and Heng Yin. 2017. Dark Hazard : Learning-based , Large-scale Discovery of Hidden Sensitive Operations in Android Apps. *NDSS* March (2017). https://doi.org/10.14722/ndss.2017.23265

[61] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. 2017. Dark Hazard: Learning-based, Large-Scale Discovery of Hidden Sensitive Operations in Android Apps. In *NDSS*. San Diego, CA. https://doi.org/10.14722/ndss.2017.23265

[62] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. 2010. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*. ACM.

[63] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. {TESSERACT}: Eliminating Experimental Bias in Malware Classification across Space and Time. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 729–746.

[64] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. 2018. ClickShield: Are You Hiding Something? Towards Eradicating Clickjacking on Android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1120–1136.

[65] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. ACM Press, 361–374. https://doi.org/10.1145/2906388.2906392

[66] Irwin Reyes, Primal Wijesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-Rodriguez, and Serge Egelman. 2018. Won't Somebody Think of the Children? Examining COPPA Compliance at Scale. *PETS* 2018, 3 (2018), 63–83.

[67] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. 2008. Learning and classification of malware behavior. In *DIMVA*. Springer.

[68] E Eugene Schultz. 2003. PandoraâĂŹs Box: spyware, adware, autoexecution, and NGSCB. *Computers & Security* (2003).

[69] Asaf Shabtai, Yuval Elovici, and Lior Rokach. 2012. *A survey of data leakage detection and prevention solutions*. Springer Science & Business Media.

[70] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. 2016. Toward a framework for detecting privacy policy violations in android application code. ACM Press, 25–36. https://doi.org/10.1145/2884781.2884855

[71] Yihang Song and Urs Hengartner. 2015. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. ACM Press, 15–26. https://doi.org/10.1145/2808117.2808120

[72] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. 2017. DroidSieve: Fast and accurate classification of obfuscated android malware. In *ACM CODASPY 2017*.

[73] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. 2014. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications* (2014).

[74] Syaifuddin Syaifuddin, Zamah Sari, and Mohammad Khairul Masduqi. 2018. Analysis of Uapush Malware Infection using Static and Behavior Method on Android. 3 (01 2018), 83.

[75] Symantec. [n.d.]. A-Z Listing of Thearts and Risks. https://www.symantec.com/security_response/landing/azlisting.jsp. Accessed: July 2017.

[76] Hien Thi Thu Truong, Eemil Lagerspetz, Petteri Nurmi, Adam J Oliner, Sasu Tarkoma, N Asokan, and Sourav Bhattacharya. 2014. The company you keep: Mobile malware infection rates and inexpensive risk indicators. In *WWW*. ACM, Seoul,KR, 39–50.
[77] Tzu-Yen Wang, Shi-Jinn Horng, Ming-Yang Su, Chin-Hsiung Wu, Peng-Chu Wang, and Wei-Zen Su. 2006. A surveillance spyware detection system based on data mining methods. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*. IEEE, 3236–3241.
[78] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. 2015. Effective Real-Time Android Application Auditing. IEEE. https://doi.org/10.1109/SP.2015.60
[79] Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In *USENIX Security*.
[80] Shui Yu, Guofei Gu, Ahmed Barnawi, Song Guo, and Ivan Stojmenovic. 2015. Malware propagation in large-scale networks. *IEEE TKDE* 27, 1 (2015), 170–179.
[81] Hao Zhang, Danfeng (Daphne) Yao, and Naren Ramakrishnan. 2016. Causality-based Sensemaking of Network Traffic for Android Application Security. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security - ALSec '16*. ACM Press, Vienna, Austria, 47–58. https://doi.org/10.1145/2996758.2996760
[82] Hao Zhang, Danfeng Daphne Yao, Naren Ramakrishnan, and Zhibin Zhang. 2016. Causality reasoning about network events for detecting stealthy malware activities. *Computers & Security* (2016).
[83] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In *ACM CCS*.
[84] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *IEEE Symp. on Security and Privacy*.
[85] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven M. Bellovin, and Joel Reidenberg. 2017. Automated Analysis of Privacy Requirements for Mobile Apps. In *NDSS*. https://doi.org/10.14722/ndss.2017.23034

APPENDIX

A DOWNLOADED SPYWARE

Table 13 reports a list of all the spyware families we have downloaded from VirusTotal. For the sake of completeness, we also report the detail on which samples we have been able to statically and dynamically execute on the Koodous service without any exception or crash.

B DEEP LEARNING CLASSIFIERS AND HYPER-PARAMETER TUNING

We report some additional details and clarifications for the deep learning algorithms used in the detection section, with detailed deep learning architectures and a thorough description of how we performed hyperparameter optimization.

- **CNN**: Convolutional Neural Networks can abstract from low-level features of data to higher-level features, where the higher-level features are a combination of the low-level ones. When the high-level features become more and more abstract, they can express more and more general characteristics of the data. The higher the level of abstraction, the less uncertainty there is, and the easier is for the model to classify samples. We have used the following CNN configuration.
  Input layer: 1D convolution layer; Number of units: the dimension of the sample's feature vector;
  Activation function: ReLU
  Max pooling layer: the size of max-pooling windows: 3
  Second 1D convolution layer; Number of units: 512; Activation function: ReLU
  First Dropout layer: rate = 0.5
  Third 1D convolution layer; Number of units: 256; Activation function: ReLU
  Global Average Pooling layer
  Second Dropout layer: rate = 0.5
  Dense connected NN layer: Number of units: 1; Activation function: Sigmoid
  Number of examples per minibatch: 16
  Optimizer: Adam

The learning rate for weight updates: 0.0001
Number of iterations over the training dataset to perform during training: 20

- **Wide & Deep** [17]: Designed to enable trained models to simultaneously acquire memorization and generalization capabilities: Memorization finds the correlation between samples from historical data. Generalization is the transfer of correlations, discovering new combinations of features that rarely or not appear in historical data. The Deep model is a feed-forward neural network. Deep neural network models usually require continuous dense features. For sparse, high-dimensional class features, they are usually converted to low-dimensional vectors and then trained. Memorization reflects the accuracy of the model, while generalization reflects the novelty of the model. Description of the used parameters of Deep & Wide in the experiments:
  Wide: Input layer with the number of units: the dimension of the sample's feature vector
  Deep: Input layer: Number of units: the dimension of the sample's feature vector
  First hidden layer: Number of examples: 256; Activation function: ReLU
  Second hidden layer: Number of examples: 1024; Activation function: ReLU
  First Dropout layer: rate = 0.5
  Third hidden layer: Number of examples: 512; Activation function: ReLU
  Fourth hidden layer: Number of examples: 256; Activation function: ReLU
  Fifth hidden layer: Number of examples: 128; Activation function: ReLU
  Second Dropout layer: rate = 0.5
  Number of examples per minibatch: 16
  Optimizer: Adam
  The learning rate for weight updates: 0.0001
  Number of iterations over the training dataset to perform during training: 20

- **MLP**: Multi-Layer Perceptron is a version of feed-forward neural networks. We refer to the architecture suggested in Python's sklearn library, with 100 hidden layers, ReLU activation functions, and Adam solver.
- **BRBM**: Bernoulli Restricted Boltzmann Machine is an unsupervised non-linear feature learner based on probability models. Features extracted by RBM often give good results when inputting linear classifiers such as linear SVMs or perceptrons. In the experiments, we use Bernoulli RBM model in scikit-learn, which assumes that the input is a binary value or a value between 0 and 1. Description of the used parameters of BRBM in the experiments:
  Number of binary hidden units: 256
  Number of examples per minibatch: 10
  The learning rate for weight updates: 0.1
  Number of iterations over the training dataset to perform during training: 10

We performed extensive hyper-parameter optimization, with particular attention to the deep learning algorithms, since they overall achieve a lower performance than other shallow learning algorithms (e.g., RF). In particular, we have tried the following methods:

(1) **Drawing figures of the accuracy on the training set and validation set.** If the model does not converge for a while, we can stop the current training and using other hyper-parameters. If the accuracy on both the training set and validation set are low, which means that the model is under-fitting. Thus, we increase the fitting ability of the model like increasing the layers of the neural network, increase the number of nodes, decrease the dropout value, and reduce the l_2 regularizer value and so on. If the accuracy on the training

set is high while low on the validation set, it means the model is over-fitting. Then we would adjust the hyper-parameters in the direction of improving the generalization ability of the model.

(2) **Tuning the hyper-parameters from coarse to fine.** Generally, a preliminary range search is performed first, and then a narrower search is performed according to where the good results appear. We start with the parameters that are more important while fixing other parameters, then adjust other parameters based on the current result. For example, the learning rate is generally more important than the regular value and the dropout value.

(3) **Automatic hyper-parameters tuning**. We tried both *Gird Search* and *Random Search* in our experiments. The disadvantage of grid search is that it's time-consuming, especially when applied to neural networks, and generally can't traverse too many parameter combinations, while Random Search is more effective in practice. So we use Gird Search's method to get all the candidate parameters at first, and then randomly select training from each time.

(4) **A smaller model for hyper-parameters tuning.** In the experiment, we first try to tune the parameters on a randomly chose smaller data set, and we assume the smaller dataset share the same distribution as the whole dataset. Therefore, we increase the speed in the training process and try more parameters combination.

(5) **Logarithmic search.** We performed a hyper-parametric search on a logarithmic scale. In the experiment, we can try the learning rate from 0.001 0.01 0.1 1 10 in 10 steps.

(6) **Empirical parameter search.** The Learning rate generally starts from 1, and the learning rate generally decays with the process of training, and we use adaptive gradient methods like Adam and default values during the training process. The number of nodes per layer is always like 16, 32, 64, 128, the multiples of 16 are usually adopted. Batch size: starts from 128. The increase in the batch size value does increase the training speed. However, there is a possibility that the convergence result is deteriorated. If the memory size allows, we should start with a larger value. If the batch size is large, it generally does not have much effect on the result, otherwise, the result may be poor.

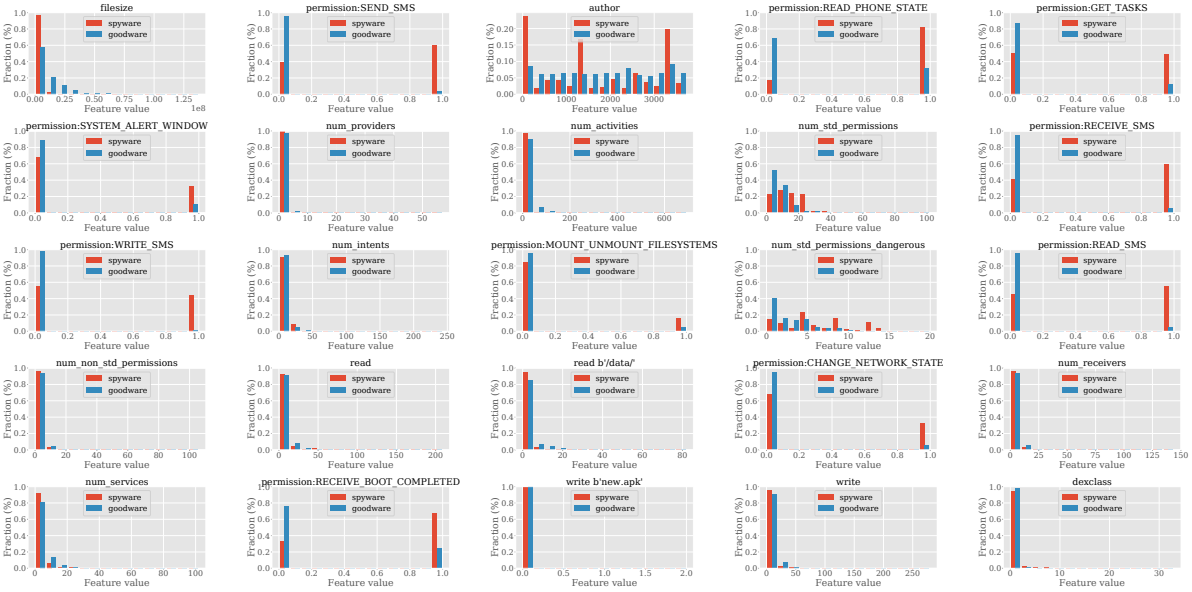
C FEATURE HISTOGRAMS

C.1 Spyware vs. Goodware

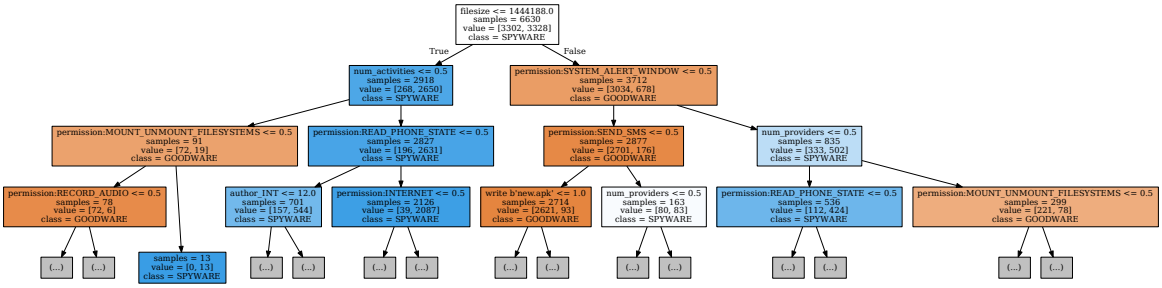
Figure 2(a) shows the distribution of the values of the top-25 features used by RF to classify each Android sample as either spyware or goodware. Each histogram reports feature values on the X-axis (note that many features are binary—and assume only values 0 or 1), and the percentage of samples of each class that have that feature value on the Y-axis. To better understand how these histograms should be interpreted, consider the `WRITE_SMS` permission in the third row of Figure 2(a). We observe that about 50% of spyware samples request permission to write SMS, while almost no goodware require this permission. From this histogram, we may infer that a decision rule that checks *whether an Android sample asks for this permission* would have high precision in detecting spyware (as it is requested by very few goodware), but uncertain recall (because if the permission is not requested, we cannot determine whether it is spyware or goodware). Figure 2(b) provides a visualization of a decision tree (limited to the first three decision levels for readability). We report the histograms corresponding to Mean Decreased Impurity (MDI) as described in Section 5.

C.2 Spyware vs. Other-malware

Figure 3(a) shows the histograms of feature values of the top-25 features used by RF to perform classification (average over 10-fold cross-validation) and Figure 3(b) shows a visualization of first three levels a decision tree.



(a) Histogram of feature values (top-25 RF features).

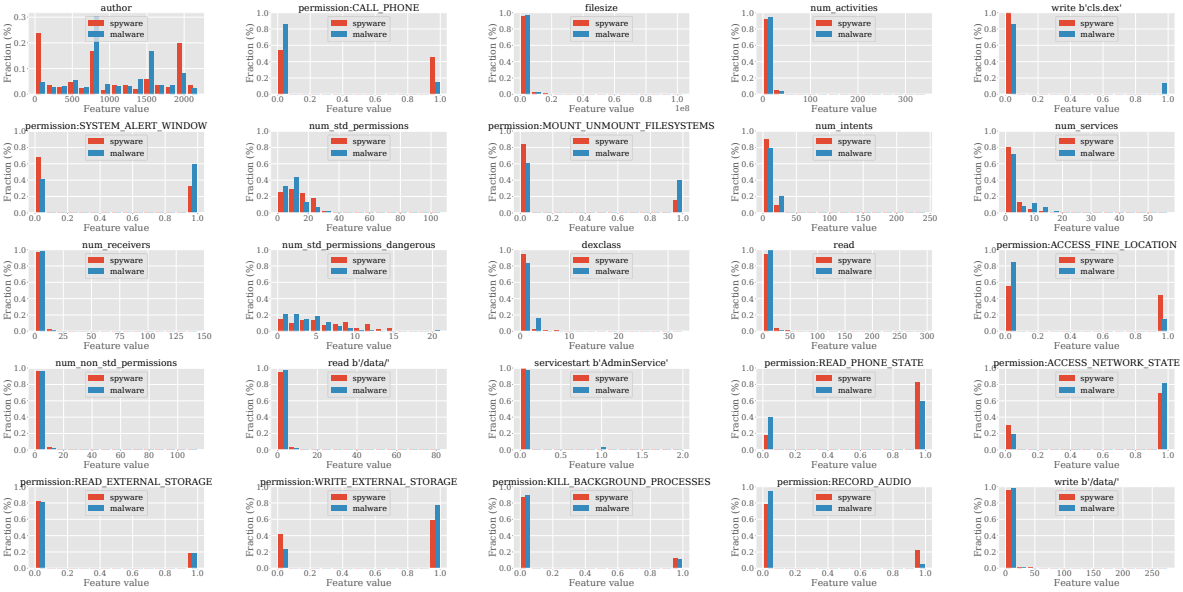


(b) Decision tree (first three levels).

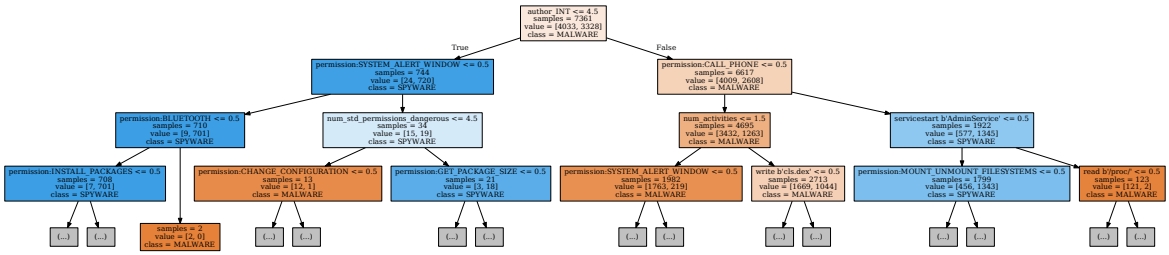
Fig. 2. Spyware vs. goodware.

C.3 Spyware Families

We include the feature-value histograms of the top features found by the classifier for specific spyware families (in particular, the top-15 features of each family vs. each other category)..



(a) Histogram of feature values (top-25 RF features).

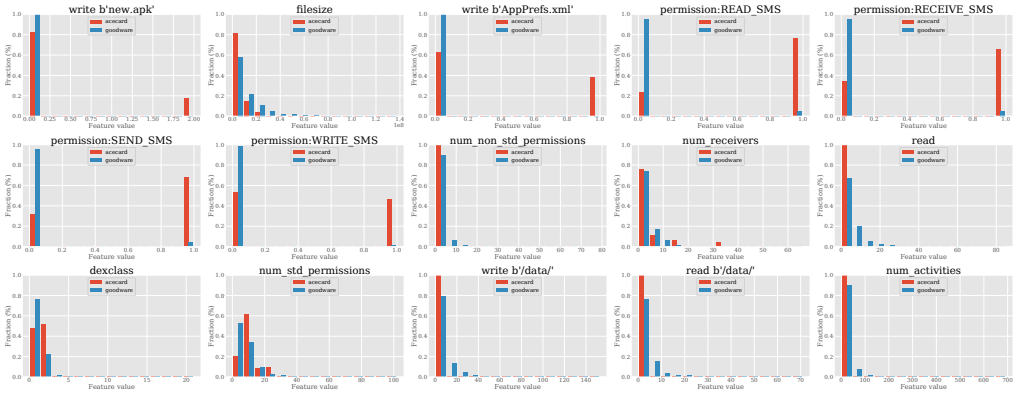


(b) Decision tree (first three levels).

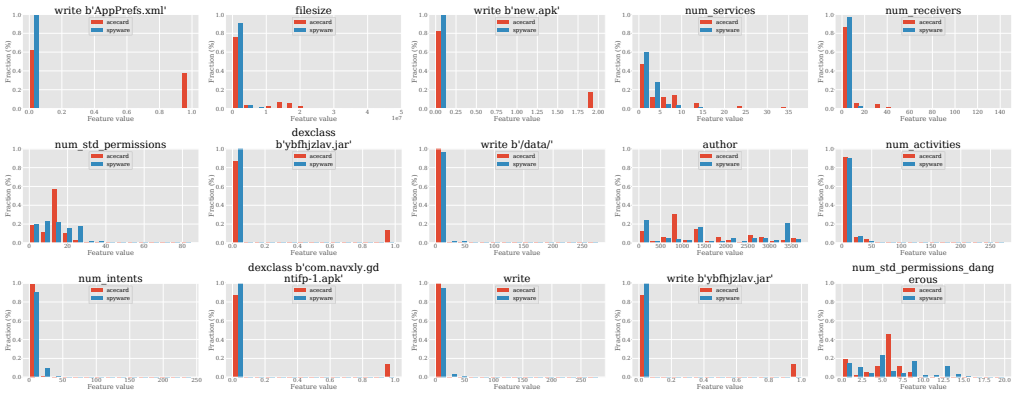
Fig. 3. Spyware vs. other-malware.

Table 13. Spyware families in [75] downloaded from VirusTotal. Families are reported in alphabetical order. Each family reports two values: the number of successfully analyzed samples, and the total number of downloaded samples from VirusTotal. The five families in bold are the ones analyzed in-depth in our paper.

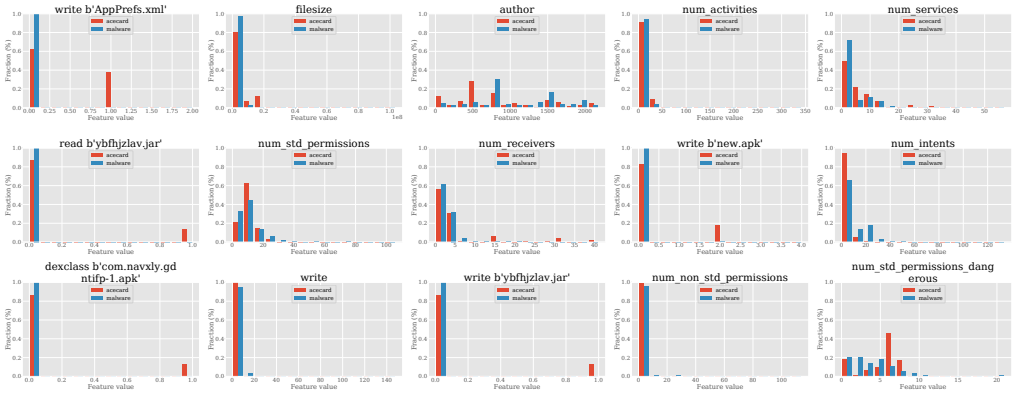
Family	**# Samples**
acecard	127/150
backflash	008/011
bankosy	016/021
basedao	018/018
beita	009/012
biigespy	006/012
claco	061/125
cosha	314/400
crisis	158/173
dendoroid	005/005
exprespam	022/041
fakebank	394/423
fakedaum	009/009
fakegame	046/047
fakelogin	117/132
fakeplay	073/149
faketaobao	025/035
farmbaby	037/041
flexispy	067/068
godwon	021/033
gugespy	107/139
hehe	140/150
jollyserv	002/002
lastacloud	007/007
machinleak	002/002
milipnot	001/001
obad	008/150
phospy	018/024
pincer	176/269
qitmo	004/004
roidsec	019/028
rootnik	157/172
rusms	130/150
sandorat	150/150
scipiex	008/008
smsstealer	117/143
spyagent	233/384
spyoo	315/400
spytrack	183/225
stealthgenie	007/009
stels	009/009
tetus	107/211
uapush	154/249
uracto	006/006
usbcleaver	020/043
uupay	076/150
windseeker	009/009
Total:	3,698/5,000



(a) AceCard vs goodware

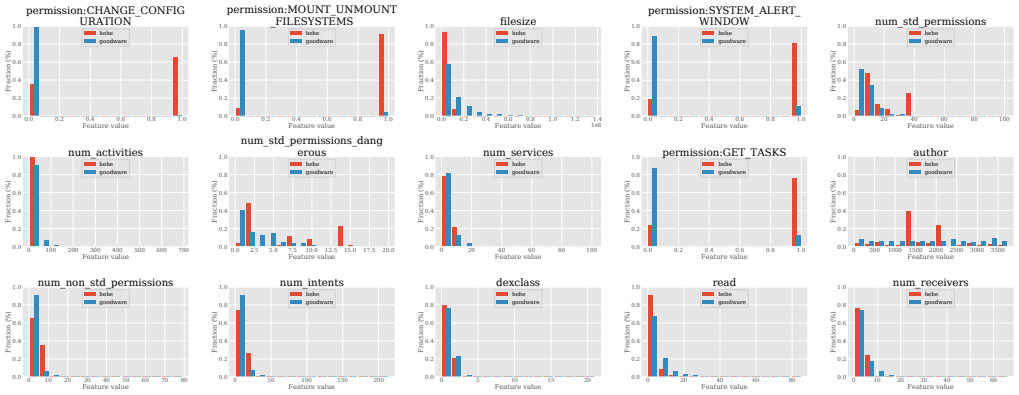


(b) AceCard vs spyware

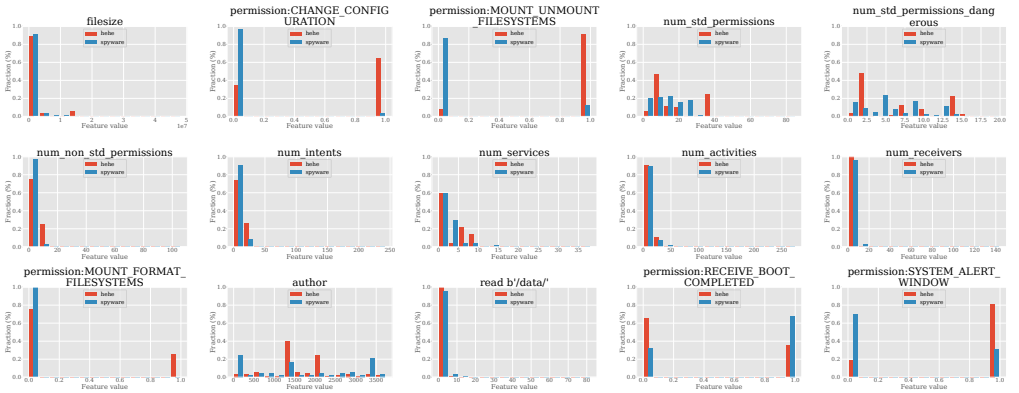


(c) AceCard vs other-malware

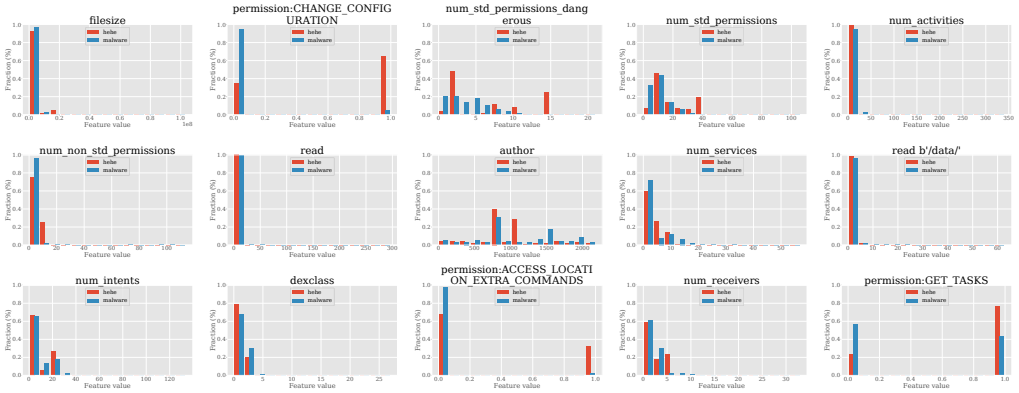
Fig. 4. **[MDI]** Feature value histograms for AceCard spyware family.



(a) HeHe vs goodware

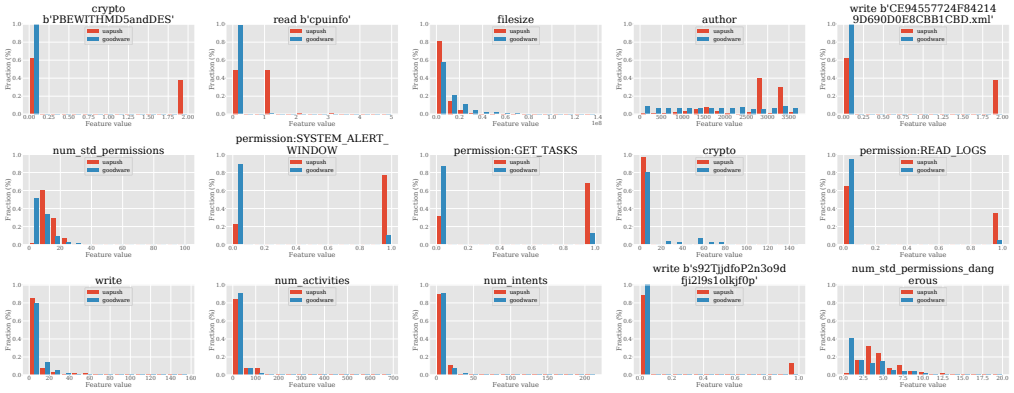


(b) HeHe vs spyware

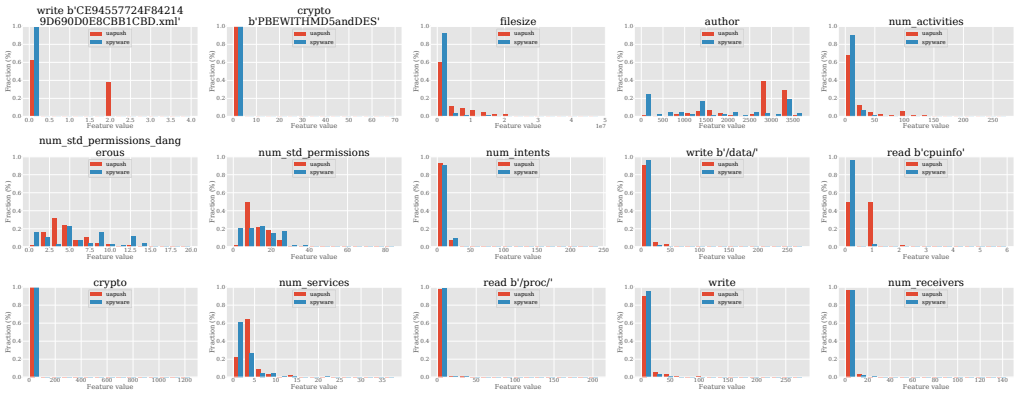


(c) HeHe vs other-malware

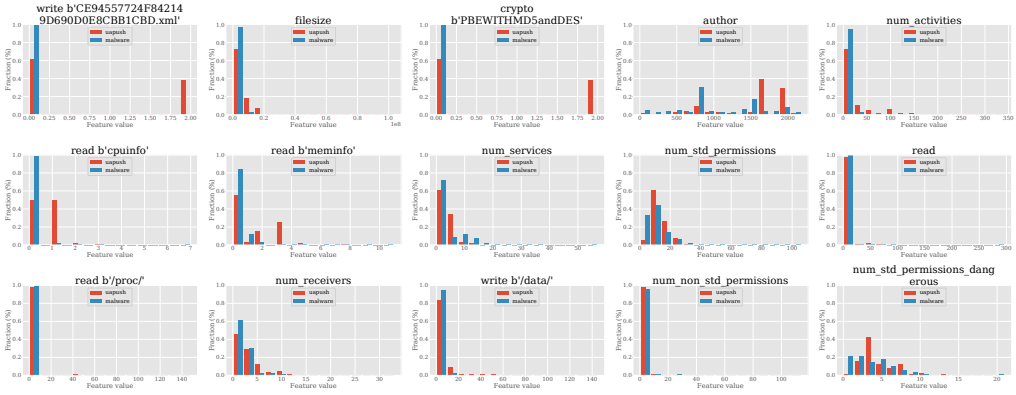
Fig. 5. **[MDI]** Feature value histograms for HeHe spyware family.



(a) UaPush vs goodware

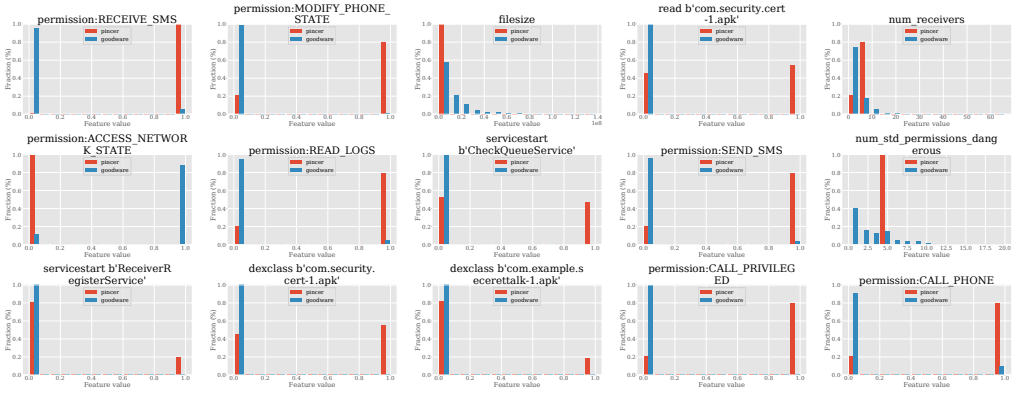


(b) UaPush vs spyware

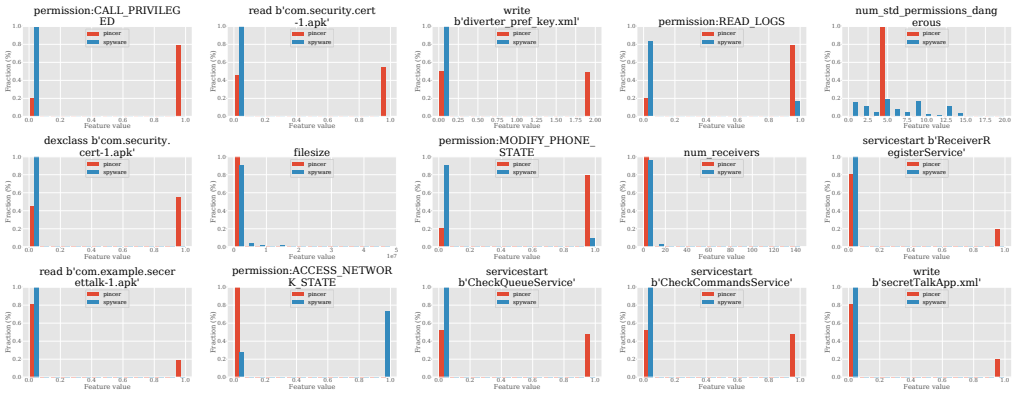


(c) UaPush vs other-malware

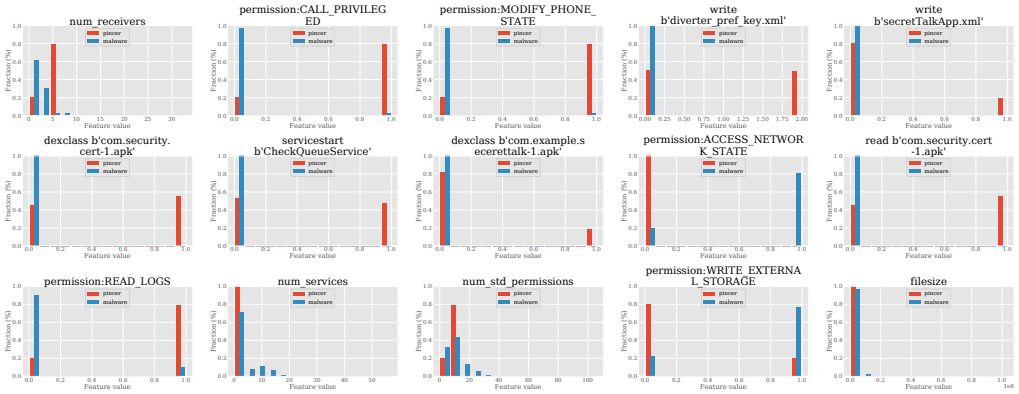
Fig. 6. **[MDI]** Feature value histograms for UaPush spyware family.



(a) Pincer vs goodware

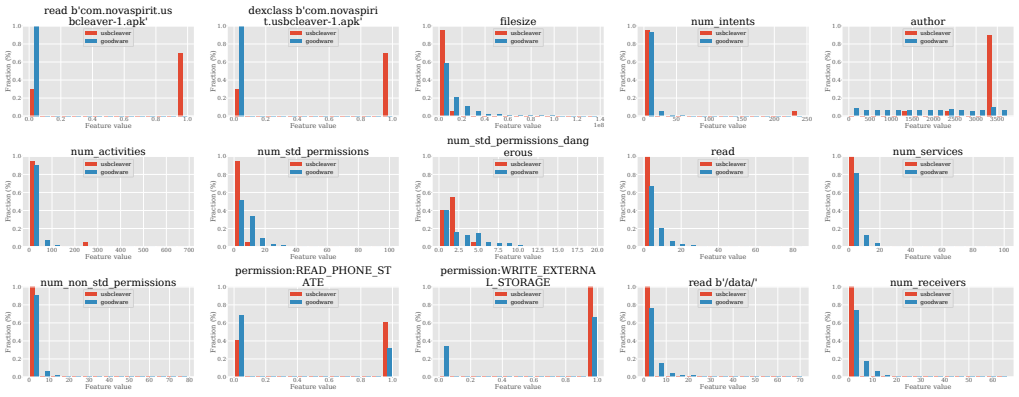


(b) Pincer vs spyware

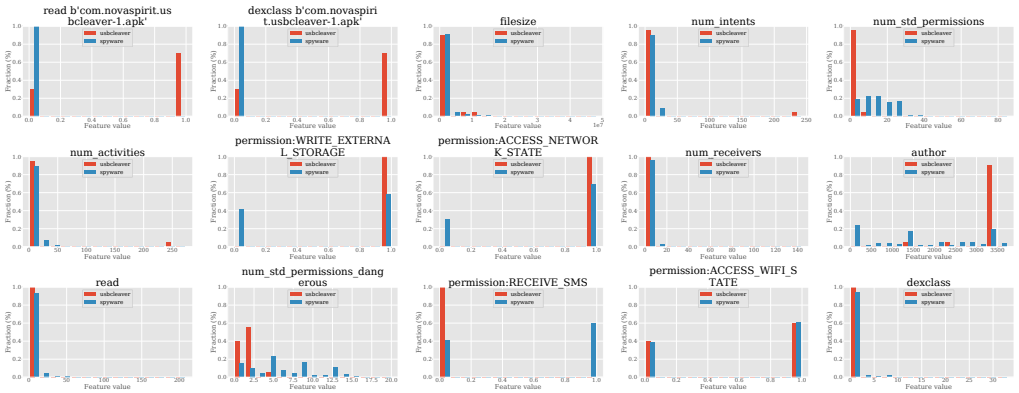


(c) Pincer vs other-malware

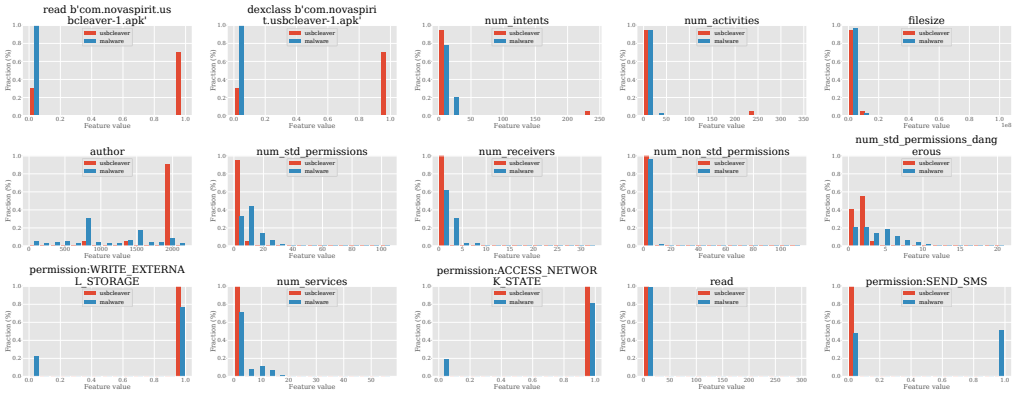
Fig. 7. **[MDI]** Feature value histograms for Pincer spyware family.



(a) USBCleaver vs goodware



(b) USBCleaver vs spyware



(c) USBCleaver vs other-malware

Fig. 8. **[MDI]** Feature value histograms for USBCleaver spyware family.