DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware

Guillermo Suarez-Tangil[1], Santanu Kumar Dash[1], Mansour Ahmadi[2],
Johannes Kinder[1], Giorgio Giacinto[2] and Lorenzo Cavallaro[1]

[1] Royal Holloway, University of London*
[2] University of Cagliari†

ABSTRACT

With more than two million applications, Android marketplaces require automatic and scalable methods to efficiently vet apps for the absence of malicious threats. Recent techniques have successfully relied on the extraction of lightweight syntactic features suitable for machine learning classification, but despite their promising results, the very nature of such features suggest they would unlikely—on their own—be suitable for detecting obfuscated Android malware. To address this challenge, we propose DroidSieve, an Android malware classifier based on static analysis that is fast, accurate, and resilient to obfuscation. For a given app, DroidSieve first decides whether the app is malicious and, if so, classifies it as belonging to a family of related malware. DroidSieve exploits obfuscation-invariant features and artifacts introduced by obfuscation mechanisms used in malware. At the same time, these purely static features are designed for processing at scale and can be extracted quickly. For malware detection, we achieve up to 99.82% accuracy with zero false positives; for family identification of obfuscated malware, we achieve 99.26% accuracy at a fraction of the computational cost of state-of-the-art techniques.

Keywords

Android Malware Detection, Malware Family Identification, Obfuscation, Native Code, Security, Machine Learning, Classification, Scalability

1 Introduction

The Android ecosystem continues to grow, and with close to two million apps published on marketplaces today, it is clear that fast and reliable mechanisms are required to detect and analyze potentially dangerous apps. The first problem we look at is *malware detection*: operators of app markets wish to automatically check submitted apps for malicious or potentially harmful code to protect users. The second problem we are interested in is *family identification*: an important step of forensic analysis of malicious apps is to differentiate families of related or derived malware [35]. For both detection and family identification, we strongly prefer light-weight and scalable methods to cope with the numbers of apps, both benign and malicious.

In general, static analysis techniques are computationally cheaper than emulation-based dynamic analysis; unfortunately, many static analysis techniques are easily thwarted by obfuscation, which is becoming increasingly common on Android [28]. Family identification in particular also suffers from the widespread code reuse in malware, which leads to different malware families sharing code and entire modules.

To address these challenges, we introduce DroidSieve, a system for malware classification whose features are derived from a fast and scalable, yet accurate and obfuscation-resilient static analysis of Android apps. DroidSieve relies on several features known to be characteristic of Android malware, including API calls [1, 38, 5], code structure [35], permissions [40], and the set of invoked components [5]. In addition, DroidSieve performs a novel deep inspection of the app to identify discriminating features missed by existing techniques, including native components, obfuscation artifacts, and features that are invariant under obfuscation. In particular, we make the following contributions to the state of the art:

- We introduce a novel set of features for static detection of Android malware that includes the use of embedded assets and native code; it is at the same time robust and computationally inexpensive. We evaluate its robustness on a set of over 100K benign and malicious Android apps. For detection, we achieve up to 99.82% accuracy with zero false positives. The same features allow family identification with an accuracy of 99.26%.
- We analyze the relative importance of our features and demonstrate that artifacts introduced by state-of-the-art obfuscation mechanisms provide high-quality features for reliable detection and family identification. Moreover, we show that there is a small set of features that perform consistently well regardless of whether they are derived from obfuscated or plain malware.

The rest of the paper is organized as follows: We first mo-

*{guillermo.suarez-tangil, santanu.dash, johannes.kinder, lorenzo.cavallaro}@rhul.ac.uk
†{mansour.ahmadi, giacinto}@diee.unica.it

CODASPY'17, March 22-24, 2017, Scottsdale, AZ, USA
 ISBN 978-1-4503-4523-1/17/03…$15.00
DOI: http://dx.doi.org/10.1145/3029806.3029825

tivate our choice of features by briefly reviewing obfuscation techniques in Android malware (§2). We then describe our two main classes of features (§3) before presenting our experimental setup and results (§4). Finally, we review related work (§6) and conclude (§7).

2 Obfuscation in Android

We now briefly review the state of the art in Android obfuscation as it motivates our work. Thorough taxonomies of software obfuscations are available in the literature [10, 31].

String Obfuscation. Recent approaches to fingerprinting malware have made use of string-based features such as permissions and apps/package names [1, 5, 21]. Some strings, such as the declaration of application permissions, follow a strict syntax and must appear in the clear; other strings, such as names and identifiers, can be easily randomized or encrypted [9, 24].

Native Code. Native code is also frequently used to offload malicious functionality from the main Dalvik executable (DEX) to dynamically linked libraries or other executables (ELF files), which are then invoked at runtime.

Dynamic Code Loading. Native code and additional Dalvik bytecode can be loaded from a library included in the app's assets, from another app (collusion attack) or from a remote system after being retrieved at runtime. In our experiments, we found many examples of dynamic code loading, including cases where code was loaded from outside of the app. However, the mere presence of dynamic code loading is not malicious in itself, since many regular software frameworks employ this technique, which makes it even more attractive to malware writers.

Code Hiding. Malware authors often proactively hide malicious components to make the overall application look benign to cursory inspection [4]. For instance, the *GingerMaster* malware hides Bash scripts for its packaged root exploit under innocuous file names such as `install.png` and `gbfm.png` in its resources [33]. Other malicious apps go a step further and use a form of steganography, e.g., by hiding malicious code inside a valid image file [34]. The app loads the image through a seemingly benign action but uses a decoding algorithm to extract a malicious executable payload[1].

Finally, Android malware can also hide its malicious payload in an APK file hosted as a resource of the main app. When the app is executed, the user is lured into installing the hidden APK and the system then dynamically loads the hidden component. In the rest of the paper, we refer to these apps as *incognito apps*. In a related scenario, the *update attack*, the app just contains a component that downloads and executes a malicious payload from an external server. Such attacks are hard to detect and mitigate as the app misleads the user to grant the additional permissions while pretending to update itself [26].

The aforementioned methods for code hiding can easily be combined with encryption to further obfuscate the malicious payloads and decrypt them only at runtime [4]. While encryption makes it harder to assess the component statically, its presence can be detected by measuring the entropy of the component. However, encryption is also commonly employed by benign apps, and during our experiments, we particularly found that many benign apps were using encrypted strings.

Reflection. Reflection is a commonly used feature in various Java frameworks, but it is also a notorious impediment to static analysis, since it may be infeasible to statically determine which code is executed at runtime. As a consequence, malware writers have long discovered reflection for obfuscating sensitive API calls and libraries [4]. In a recent large-scale study, Lindorfer et al. [22] showed that the general use of reflection among apps has increased significantly.

The state of obfuscation on Android has caught up with that on desktop systems, and there are already automatic frameworks available for obfuscating Android app components [9, 24, 28]. Hence, obfuscation now poses a serious challenge for static malware analysis on Android and has to be addressed to achieve robust malware classification.

3 Feature Engineering

We now introduce our proposed set of features for both malware detection and identification of malware families. Based on an analysis of existing malware (§3.1), we identify two major classes: *resource-centric* features are derived from resources used by the app (§3.2); *syntactic* features are derived from the code and metadata of the app (§3.3). A map relating classes of features is shown in Figure 1. We use both binary and continuous features. The presence or absence of a particular trait, such as a permission, is encoded as a binary feature; numeric properties, such as string lengths or opcode frequency, are encoded as continuous features.

3.1 Prevalence of Features

Robust classification requires a diverse set of features. Features such as API calls are highly relevant for classifying non-obfuscated malware but are susceptible to obfuscation. The presence of obfuscations may indicate malware, but it is not by itself sufficient to form judgment, since benign software can also use the same techniques for legitimate purposes. Therefore, we propose to employ a portfolio of features that covers both non-obfuscated and obfuscated malware.

As a first step towards selecting effective features, we measured the prevalence of a wide range of features that could be effective at identifying both obfuscated and non-obfuscated Android malware. We hypothesize that features centered around steganography, where the sample hides its malicious payload in its assets, or inconsistent nomenclature of components of an app by a careless malware developer are important features. To test our hypothesis, we run an assessment on a collection of over 100,000 benign and malicious samples from multiple sources. To put our findings in perspective, we also select some features from published works on Android malware identification.

For benign samples, we obtained a dataset of clean apps vetted by McAfee (McGW). For the malicious samples, we relied on two commonly used datasets: the Malgenome Project (MgMW) [41] and the Drebin dataset [5]. We further extended our dataset with the goodware (MvGW) and malware (MvMW) collected by Lindorfer et al. [21]. To measure feature prevalence in obfuscated malware, we also include the recent PRAGuard (PgMW) dataset [24]. The samples in PRAGuard were obtained by obfuscating the samples of the

[1] A recent example is *Android/TrojanDropper.Agent.EP* (*MD5:1f41ba0781d51751971ee705dfa307d2*), November 2015. b0n1.blogspot.co.uk/2015/11/android-malware-drops-banker-from-png.html

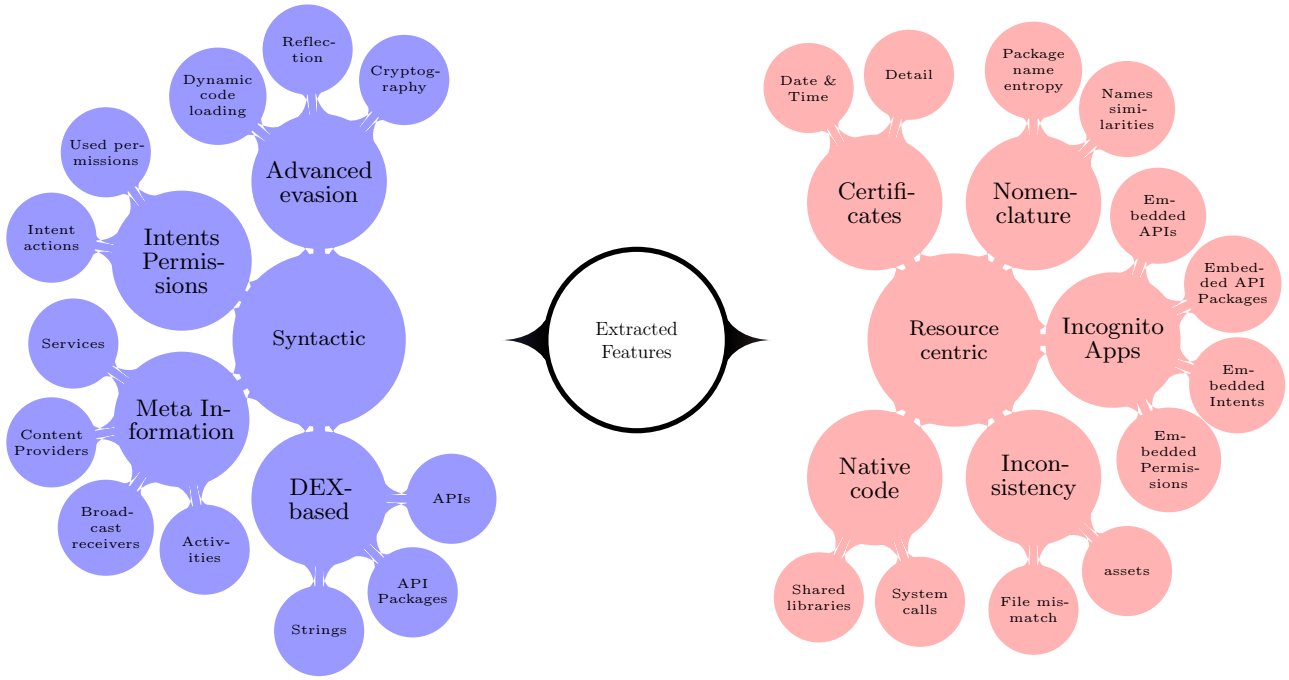


Figure 1: Non-exhaustive map of extracted features. The left side shows syntactic features derived from the source code of the app; the right side shows resource-centric features derived from the assets of the app.

MgMW dataset with techniques such as class and method renaming, reflection, and class encryption, among others.

Table 1 summarizes the results of our investigation. We can observe that most of the features are more prevalent in malware than in goodware. In particular, structural and logical inconsistencies are between 5% and 35% more prevalent in malware. In fact, the difference in prevalence of these features is comparable to well-understood features such as permissions, sensitive API calls, and those related to SMS messaging. Thus, inconsistencies are an important class of feature that have not been reported in the literature so far.

Our work also identifies obfuscated malware. In view of this, we also looked for prevalence of features that may hint at obfuscation in the form of reflection or the use of native code. In our study, McGW contained a prevalence higher than either MvGW and most of the malware datasets. This is because McGW is a more recent dataset with samples ranging from 2012 to 2016 and use advanced coding techniques while other datasets with the exception of PgMW are from 2012 and 2013. One may wonder the utility of including these features as a part of the classifier if they cannot be used to classify modern samples. A key assumption that we make here is that the classification model should evolve over time as pointed out recently in the literature [12]. Features that are relevant today will naturally become irrelevant in the future and it is the responsibility of malware analyst to purge obsolete features from the model while retraining. For our experiments, we retain these features as we test our features over a large timespan.

The PgMW dataset deserve special mention as it highlights how standard forms of obfuscation can confound the classification model. For the PgMW dataset, it can be seen that some features that are common in malware can be easily obfuscated. For example, methods that are crucial for the detection of malicious activities, such as communications (*SMS*) or the access to sensitive information (*getSimSerialNum.*), have been nearly eliminated in the obfuscated dataset. Therefore, relying on these features alone when dealing with obfuscation is detrimental to malware analysis and detection. These findings further reinforce our original suggestion of using a diverse portfolio of features for resilient classification.

3.2 Resource-centric Features

We propose a set of new features extracted from the app's resources stored in the APK. An excerpt of the resource-based features that we use can be seen in Table 1.

The two main guiding criteria that we use for building the set of resource-based features are *structural inconsistencies* and *logical inconsistencies*. Structural inconsistencies refer to the artifacts left behind after hiding a malicious component. Logical inconsistencies refer to the footprints typically left when repackaging a piece of malware as part of a benign app.

Certificates. We check whether the times at which the app was signed and at which the certificate was generated are similar. The intuition behind this feature is that automated repackaging tools modify existing apps and sign them using auto-generated ad-hoc certificates before distribution. Thus, if the date when the certificate was created is close to the date on which the app was signed, it can reveal the use of an automated tool for app repackaging. We mark apps where the time difference was below ten minutes. For each certificate, we also build features from the timezone and the common name's string length, which allows to identify similar certificates generates by repackaging tools.

Nomenclature. For each of the components in the app, we verify whether the correct package name is used as a prefix of the components in a package directory which is

Type	Capability	Goodware		Malware				Summary	
		McGW	**MvGW**	**MgMW**	**Drebin**	**PgMW**	**MvMW**	**Goodware**	**Malware**
Logical Inconsistencies	Main Activity	15.01%	8.85%	29.44%	18.71%	29.60%	8.23%	9.31%	13.13%
	Service	43.44%	4.60%	72.62%	54.17%	74.29%	35.34%	7.51%	44.18%
	Receiver	46.23%	13.57%	74.29%	56.06%	75.87%	36.60%	16.02%	45.66%
Structural Inconsistencies	APK File Match	1.77%	0.07%	24.21%	6.51%	24.13%	2.23%	0.20%	5.18%
	APK File Extension Mismatch	1.41%	0.02%	23.89%	6.28%	24.13%	2.22%	0.12%	5.10%
	Image File Extension Mismatch	3.69%	1.48%	19.92%	8.22%	18.17%	1.44%	1.65%	4.82%
Sensitive API	Package: SMS	5.63%	1.92%	20.79%	36.53%	0.00%	57.80%	2.20%	46.82%
	TelephonyManager.getSimSerialNum.	9.24%	4.69%	50.63%	24.06%	0.08%	14.22%	5.03%	16.34%
Permissions	READ_CONTACTS	22.93%	6.26%	36.27%	23.29%	38.8%	17.20%	7.52%	20.71%
	ACCESS_FINE_LOCATION	28.04%	16.40%	34.29%	30.04%	32.30%	15.53%	17.28%	21.38%
Obfuscation	Dynamic Code	32.22%	0.44%	19.60%	6.98%	0.00%	2.04%	2.83%	3.47%
	Reflection	74.08%	39.37%	67.62%	56.04%	99.21%	40.14%	41.97%	49.50%
	Native Code	49.61%	3.69%	54.13%	19.51%	0.16%	6.43%	7.14%	10.15%
	Native Code without ELF	8.10%	0.58%	1.67%	0.70%	0.00%	0.52%	1.14%	0.54%
Total Number of samples		8,041	99,037	1,260	5,560	1,260	10,581	107,078	17,401
Total Number of families		–	–	49	179	49	–	–	–

Table 1: Percentages of apps with given properties in the McAfee Goodware (McGW), Malgenome (MgMW), Drebin malware, PRAGuard's obfuscated Malgenome (PgMW), Marvin Goodware (MvGW) and malware (MvMW) dataset. Note that the summary shows the total number of apps after removing overlapping samples.

the usual practice in most apps. If there is a mismatch, we treat it as a potential case of tampering with the original contents of a benign app. Table 1 shows an overview of the percentage of samples that exhibit such a mismatch. For each of the package names, we also derive its length and its Shannon entropy, which help to identify automatically generated names.

Inconsistent Representations. We check whether the file extensions match the file contents (as identified by the file header or a magic number) to allow highlighting apps that try to hide shell scripts or ELF binaries as images or other resources. Table 1 shows that such inconsistencies are good indicators of malicious intent in some (e.g., Malgenome) but not all (e.g., Marvin) datasets, potentially owing to trends in malware writing and repackaging tools.

Incognito Apps. In some cases the payload of a malicious app is in an APK that is disguised among the assets of the *host* app. To capture this malicious payload, we recursively extract both syntactic and resource centric features for any *incognito* APK and DEX found within the app. We pigeonhole these features under a different category in order to separate these statistics from the ones related to the *host* app. For instance, *permission.INTERNET* counts the static number of accesses to the Internet, while *icg.permission.INTERNET* does the same for the incognito app.

Native Code. We also scan the assets of the app to identify any native ELF files. The files are parsed to extract features from the header and individual sections of the file. We extract the number of entries in the program header, the program header size, and the number and size of the section headers. From individual sections, we extract the flags of the section to understand if they are W (writable), A (allocatable), X (executable), M (mergeable), S (strings), etc. and use them as Boolean features. Within code sections, we also look for instructions invoking critical system calls such as `ioctl`, which is used for Android's inter-procedural and inter-component communication.

3.3 Syntactic Features

We present our syntactic features; several of these, such as API calls [1] and permissions [40], are already known to perform well with non-obfuscated malware. We don't claim novelty by including these features. Instead, we use them to build a classifier that is robust against both well-known and modern malware which tends to be increasingly obfuscated. To enrich the set of syntactic features, we propose some new features such as *explicit intents* and additional ones mined from the *meta-information*. These are discussed below. We reiterate here that a combination of diverse features is crucial for robust detection of both plain and obfuscated malware. This is corroborated in Section 4.2 where important features come from diverse categories, yet they all rank highly in relation to other features (see Figure 2 and 3).

DEX-based Features. We tag each method based on the libraries it invokes from the Android Framework (*method tag*). These tags represent the class of APIs used by the method and are encoded as binary features. We also scan the app for the presence string variables in DEX files containing keywords we obtained from reverse engineering malware from the Malgenome data set. For instance, `su` relates to executing code with super user privileges; `emulator` and `sdk` suggest that the app checks for the presence of an emulator.

Intents and Permissions. We parse the Manifest to identify all implicit intents that can be received from other apps. We also scan the code to identify any explicit intents, which are used to start services within the same app. The count of individual intents is used as a continuous feature for classification. We break down the set of intents into sub-categories for further granularity: (i) intents containing the keywords `android.net.*`, which are related to the connection manager; (ii) intents containing `com.android.vending.*` for billing transactions; (iii) intents that target framework components (`com.android.*`); (iv) all intent actions, beginning with `android.intent.action.*`; and (v), a catch-all category for the reminder intents. Finally, we also extract the set of permissions declared in the manifest of the app.

Meta-information. Apart from the specific type of permission used, we also count the number of Android framework permissions and custom third-party permissions used by the app. The number of times that a permission is used throughout the code is encoded as a feature. Similarly, we count the number of activities, broadcast receivers, content providers, services, and entry points of the app. Entry points are ways in which an app can be invoked or executed.

Evasion Techniques. We further look for techniques that are frequently used to confuse analysis systems, i.e., native code, cryptographic libraries, or reflection. For example, `Ldalvik/system/DexClassLoader` indicates dynamic code loading, `Ljava/lang/reflect/Method` is required for invoking a method through reflection, and any access to `Ljavax/crypto` is a sign for the use of cryptography. For native code invocations, we count the number of times the Dalvik opcode `0x100` is present in the bytecode, which corresponds to loading and executing native code.

4 Experiments and Results

We implemented our proposed feature set in DroidSieve, a system for static detection and family identification of Android malware. We begin our evaluation by describing our experimental setup and evaluation metrics (§4.1). We then address the following questions:

- **Feature Engineering** (§4.2): Which types of features are most effective for regular and obfuscated malware?
- **Classification of Standard Samples** (§4.3): How effective is DroidSieve in classifying non-obfuscated malware only, and how does it compare to other approaches that address the same problem?
- **Classification of Obfuscated Samples** (§4.4): How effective is DroidSieve in classifying obfuscated malware or a mix of non-obfuscated and obfuscated malware?
- **Computational Efficiency** (§4.5): Do the computational costs of using DroidSieve allow its application at scale?

4.1 Experimental Setup

We mean to evaluate the choice of our features for two distinct problems:

Evaluation Categories. We evaluate DroidSieve along two dimensions, the classification task and the type of dataset. The classification task is either (1) *malware detection* among a set of malicious and benign samples or (2) *family identification* among a set of samples known to be malicious. The type of dataset is either non-obfuscated, obfuscated, or mixed. We use the datasets introduced in §3.1 and combinations thereof; details are shown in Table 2a.

Choice of Learning Algorithm. We implemented both malware detection and family identification in DroidSieve using Extra Trees. As alternatives we considered one-vs-all Support Vector Machines (SVM), Random Forests, and eXtreme Gradient Boost (XGBoost). In the past, SVM and Random Forest have been successfully applied to malware detection [5, 32] and they have been shown to have better performance than others after comparing them to 180 classifiers on various datasets [15]. Ensemble tree-based classifiers perform well on many real world settings, however. For example, Extra Tree [18] and Gradient Tree Boosting [19] have been achieving great performance in most of recent "Kaggle" competitions [2] on various domains, including malware classification[2] or spam detection[3].

We use feature selection to restrict the classifier to important discriminating features. A feature is selected when the importance score assigned to the feature by the classifier is higher than the mean of all the features' scores. For decision trees, this importance is computed from the mean decrease impurity (MDI) where a higher score implies a more important feature.

Evaluation Metrics. For evaluating the classification results, we use the detection rate (DR), the false positive rate (FPR), the accuracy (ACC), and the F_1-score (F1) which is the harmonic mean of the precision and recall as quality metrics. Detection rate is the correct number of predictions made over the set of malware, whereas accuracy reports the number of correct predictions made after considering both goodware and malware. We only use the detection rate for the case of malware detection and we report this metric together with the false positive rate, i.e., the number of goodware samples wrongly classified as malware divided by the total number of goodware samples in the dataset.

For assessing the performances of the proposed models, we use hold-out validation to avoid overfitting [14]; samples used to fit the model are different from the ones used to validate it. We retained one third of the samples for validation and trained the model on the remaining two thirds of the data. For each sample that was retained, we ensured that we trained on samples from the same category. For malware detection, a category for a sample indicates whether it is benign or malicious. For family identification, a category indicates the name of family. Consequently, we do not have a case of testing on samples from unseen families or categories; this would be an instance of zero-shot learning [25], a problem we consider out of scope for this paper. We did not use any form of re-sampling, such as cross-validation, to avoid biasing our results [27].

4.2 Ranking of Features

We now analyze the quality of our features, ranking them when used on unobfuscated and obfuscated datasets. We expect features that are easily obfuscated to decrease in importance, whereas features that are invariant under obfuscation should remain stable.

We pass the feature vectors for our samples to the *Extra Tree* algorithm and rank them by *mean decrease impurity* [23]. As decision trees split the dataset by considering one feature at a time, it is easy to measure how much *impurity* is introduced in the classification by choosing a particular feature. Note that these rankings are informative and do not dictate our choice of features in all sets of experiments in §4.3 and §4.4. For classification, DroidSieve uses automatic feature ranking and chooses the top features for the respective training set.

For malware detection, we passed all samples in McGW + MgMW and McGW + PgMW through the Extra Tree classifier. Figure 2a and Figure 2b depict the top 30 features for these cases, respectively. In the case of McGW + MgMW,

[2]http://blog.kaggle.com/2015/05/26/microsoft-malware-winners-interview-1st-place-no-to-overfitting/
[3]http://mlwave.com/winning-2-kaggle-in-class-competitions-on-spam/

ID	Dataset Name	Ground Truth	#samples
—	Drebin [5]	Malware	5,560
MgMW	MalGenome [41]	Malware	1,260
PgMW	PRAGuard * [24]	Malware	1,260
McGW	McAfee	Goodware	8,041
McMW	McAfee	Malware	13,289
MvGW	Marvin [21]	Goodware	99,037
MvMW	Marvin [21]	Malware	10,581

(a) Dataset sources

Set	Detection	Family Identification
1	{McAfee Goodware, Drebin}	Drebin
2	{McAfee Goodware, MalGenome}	MalGenome
3	{McAfee Goodware, PRAGuard*}	PRAGuard*
4	{Marvin Goodware, Marvin Malware}	–
5	{McAfee Goodware, McAfee Malware}	–
Hold-out Ratio: 67% Training – 33% Testing		

(b) Dataset combinations

Table 2: Overview of chosen datasets for malware detection and family identification. The set of experiments involving obfuscated samples is marked with an asterisk(*). The holdout ratio shows the percentage of samples retained for validation. For the case of Marvin and McAfee malware we retain the splitting given by the authors, otherwise we use a random split.

these 30 features account for the top 40% features, while in the case of McGW + PgMW these features account for the top 36% features. We repeated a similar experiment for the case of family identification and the top features for samples from MgMW and PgMW are presented in Figure 3a and Figure 3b, respectively. They denote the top 26% and the top 43% most important features for identifying Android malware families from MgMW and PgMW, respectively.

For both plain and obfuscated malware, it may be seen from Figures 2a and 2b that permissions (prepended with PER) play an important role in the detection process. Permissions are hard to obfuscate as scrambling them would break the Android programming model. Alongside permissions, novel syntactic features such as used-permissions (prepended with used.PER) also rank highly. These features derived after scanning the code to understand what permissions are being used and how often.

Apart from syntactic features, there are many resource-centric features which also rank highly. In particular, features derived from assets such as ELF files (prepended with elf) as well as intents, and API calls from incognito apps (prepended with icg) rank highly when detecting plain malware samples as shown in Figure 2a.

The high-ranked features for malware detection is similar for both plain and obfuscated apps. A noticeable difference in the case of obfuscated malware is that the top-ranked feature is Stat(cert_diff.1). which is derived from the certificate of the app. It checks whether the time difference between the date when the certificate was issued and time when the app was signed is within a day. A temporal proximity means that the app was signed during a time when the malware developer piggybacked the app with malicious code. This is a common practice which signals that the malware developer may be using automated tools to repackage the app.

The ranking of features for classifying malware into families for plain and obfuscated malware is shown in Figures 3a and 3b, respectively. The high-ranked features in both cases are similar to those observed in the case of classification except for two noticeable differences. Firstly, incognito features are not as important for classifying malware into families as they are for malware detection. This is understandable as incognito apps are a means to achieve a malicious action but they do not characterize what malicious action is carried out or how it is carried out. Secondly, we can see that features derived from the file type of the assets (prepended with file) and those related to logical inconsistencies (features such as Stat(PackageMismatchService) and Stat(PackageMismatchReceiver)) are highly ranked. This could point to the fact that the app is repackaged using an attack vector that is specific to a given family.

4.3 Classification Results

In this section we evaluate the effectiveness of DroidSieve in classifying unobfuscated malware, to allow a comparison against approaches from the literature. To not put DroidSieve at a disadvantage, we therefore start with a feature set that includes all features, including those that are susceptible to obfuscation.

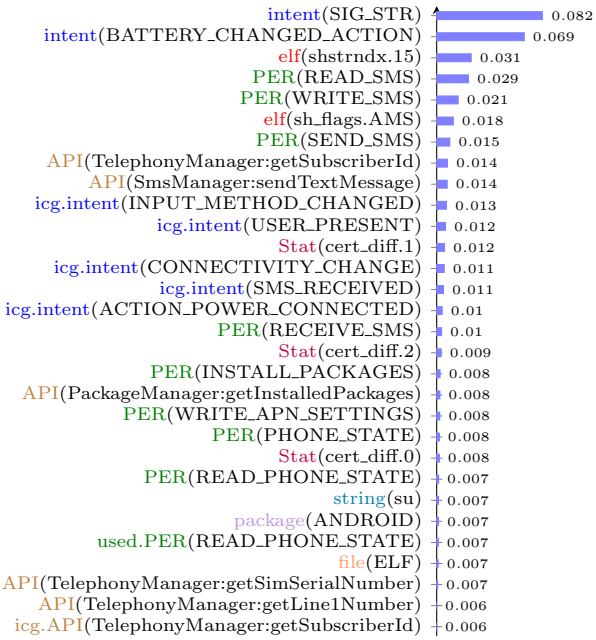
As datasets, We first evaluate on detection of malware samples where we use the dataset obtained by combining malicious samples from the Drebin dataset with the Goodware set as shown in Table 2b. Note that we only report results for the Drebin dataset here because it includes all MalGenome samples and is both larger and more recent.

Malware Detection. The table shows that in our best scenario we are able to identify if a given app is malicious or benign with accuracy of 99.64% for the case of Drebin, and 99.82% for MvGW. The breakdown of the accuracy shows a detection rate of 99.44% for Drebin, with 0.226% of false positives. Similarly, the detection rate for MvGW is 98.42% with only 0.008% of false positives. For the case of Drebin we obtained slightly higher detection rate with respect to MvGW. However, the false positive rate is better in the case of MvGW. In fact, in this case the number of goodware classified as malware is negligible (2 out of 25493). In most cases, the performance is improved with feature selection. It allows to drastically reduce the complexity of the feature space, e.g., from over 20,000 features to less than 1,000. This means that we are able to reduce redundant or irrelevant features and improve the performance of classification.

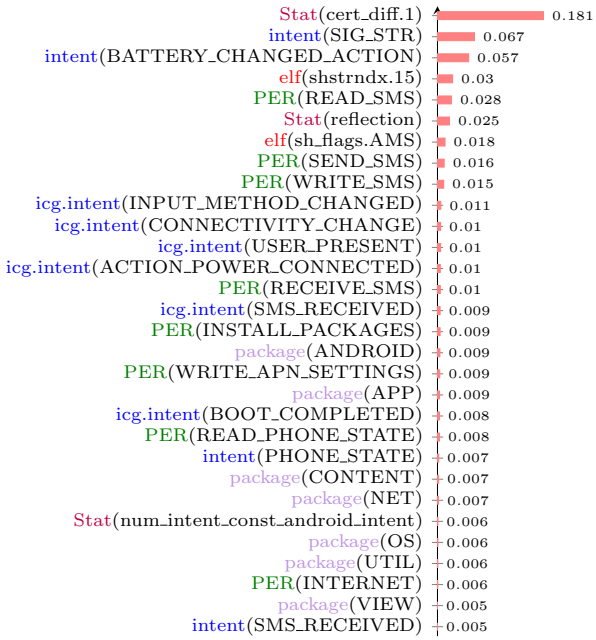
Family Identification. After detection, DroidSieve is also able to determine if the given malware belongs to a known family. Our experiments on the Drebin dataset show that Extra Trees achieve an accuracy of 97.68% when considering all 2,564 the features (see Table 2b). Interestingly, keeping the top 320 most informative features increases the accuracy to 98.12% while adding features that are not unimportant can hurt classification accuracy [29].

4.4 Obfuscation Evaluation

We now evaluate the effectiveness of our system against obfuscated malware and against a mix of obfuscated and unobfuscated malware, as it would be encountered in an actual deployment. In particular, we ran three sets of experiments for both malware detection and family identification.

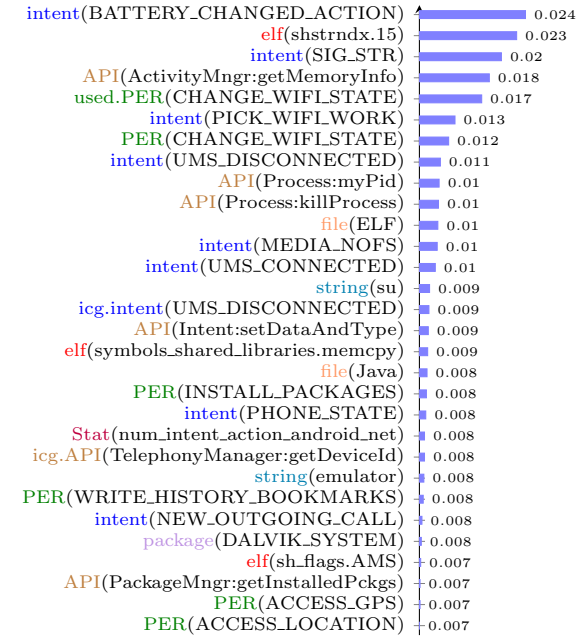


(a) Non-obfuscated malware.

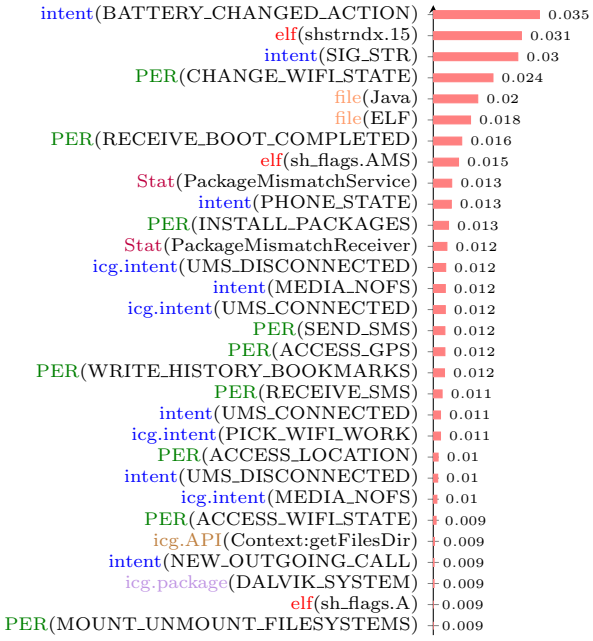


(b) Obfuscated malware.

Figure 2: Ranking of features for malware detection: Figure 2a shows importance of features by considering all features on MalGenome while Figure 2b shows importance of features for the MalGenome obfuscated (PRAGuard) dataset.



(a) Non-obfuscated malware.



(b) Obfuscated malware.

Figure 3: Ranking of features for family identification.

The three cases are based on scenarios where the training and/or testing samples are obfuscated. Note that our original dataset consists of samples from the Goodware set and samples from the MalGenome project. For each malware sample, we obtain the corresponding obfuscated sample from the PRAGuard project.

Detection of Obfuscated Malware. Our training sets for malware detection are as follows:

Type	Classifier	#F	ACC(%)	F1(%)	DR(%)	FPR(%)
Malware Detection		**Drebin + McGW**				
	Extra Trees	22,584	**99.64**	**99.64**	**99.44**	**0.226**
	Extra Trees + FS	**859**	99.57	99.57	99.39	0.302
		MvGW + MvMW				
	Extra Trees	26,396	99.72	99.72	97.58	0.012
	Extra Trees + FS	**634**	**99.82**	**99.81**	**98.42**	**0.008**
Family Identification		**Drebin** (108 families)				
	Extra Trees	2,564	97.68	97.31	–	–
	Extra Trees + FS	**320**	**98.12**	**97.84**	–	–

Table 3: Results for detection and family classification on unobfuscated malware with and without Feature Selection (FS) for the Marvin, McAfee and Drebin datasets. *#F* stands for number of features, *ACC* for Accuracy, *F1* for F_1-Score, *DR* for the detection rate, and *FPR* for False Positive Rate. Best scores for each setting are shown in bold. Although feature selection drastically reduces the number of features, it mostly outperforms the full-feature setting.

1. **McGW + MgMW**: We train on samples from the Goodware and MalGenome data sets only to show a baseline classification without obfuscation.
2. **McGW + PgMW**: We train on the obfuscated malware samples from PRAGuard and include the Goodware.
3. **McGW + MgMW + PgMW**: We train on both the original and obfuscated versions of the malware obtained from MalGenome and PRAGuard, respectively, together with the Goodware.

We chose our test cases for the trained model to highlight that the choice of our features performs consistently well regardless of whether we train on the obfuscated samples or on the original ones. In the first experiment on detecting malware, we retained 33% of samples from PgMW and McGW and trained with the rest. With the retained samples, we obtained accuracies of 100% for the McGW samples (0% false positives) and 99.02% for the PgMW samples. We repeated the experiment with the non-obfuscated set of samples (MgMW + McGW) and obtained similar accuracy values.

To further validate our features and trained models, we also tested on malware samples from a dataset that is different from the one used for training (i.e., 100% hold-out). First, we trained on all MgMW + McGW samples, and tested on PgMW samples. Then, we trained on all PgMW + McGW samples, and tested on MgMW samples. For these two experiments, our accuracy was 92.38% and 96.11% respectively. As a final experiment to validate our features for detection, we also performed a hold-out validation of the 33% of the dataset on all samples i.e. McGW + MgMW + PgMW and obtained an accuracy of 99.71%. A summary of our results for the detection task can be found in Table 4. These results show that our features are effective at distinguishing benign and malicious samples, a task made more difficult by many obfuscation techniques also having valid use cases in benign software (see Table 1).

To compare our performance with recent approaches, we use Drebin framework [5] to extract features from MgMW, McGW and PgMW datasets. We trained on all MgMW + McGW samples and tested on the obfuscated set of samples (i.e.: PgMW) using the same classification algorithm (Random Forest) used by DroidSieve. The detection rate obtained with Drebin's feature engineering is 0%. Note that our framework reported 92.38% on this experiment. We repeated the same experiment by training over the original set of malware samples collected by Drebin and testing again on PgMW. The feature set of Drebin in this setting is of 101,055 features while ours is of 22,584. After applying feature selection, Drebin retained 13,602 while we retained 859 informative features. For this experiment, the features used by the Drebin framework reported a detection rate of 11% while our framework reported 100%. Among the most important features used by Drebin were different strings such as URLs which are a soft target for obfuscation. Contrarily, our framework retained several *logical inconsistencies* (e.g.: PackageMissmatchIntentConsts and PackageMissmatchService), other resource-centric features (e.g.: PackageNameEntropy), ELF features and other statistical features (such as the number of third party permissions found).

Identification of Obfuscated Families. We now demonstrate the effectiveness of our features for identifying the classes each malware sample belongs to. Our training sets for the identification of malware families are as follows:

1. **MgMW**: We train on samples from MalGenome only.
2. **PgMW**: We train on the obfuscated malware samples from PRAGuard.
3. **MgMW + PgMW**: We train on both the original and obfuscated versions of the malware obtained from MalGenome and PRAGuard respectively.

By following the same settings as in the previous experiments, for each dataset we retained 33% of the samples from each family, when that dataset was used for both training and testing. A summary of our results on family identification can also be found in Table 4. The accuracy after training on MgMW samples was 97.79% and the accuracy after training on PgMW was 99.26%. Additionally, we also applied 100% hold-out validation between MgMW and PgMW showing accuracies of 97.94% and 97.86% respectively. It is worth noting here that training on obfuscated malware enables our classifier to perform better. On the contrary, when obfuscated samples are not included in the training set, the resulting model is not able to prioritize all features needed to perform higher than 97.79%. Finally, we tested the trained models on a combination of both obfuscated and non-obfuscated samples (MgMW + PgMW) and obtained an overall accuracy of 99.15%.

Malware Detection				Family Identification		
	Test				Test	
Training	McGW	MgMW	*PgMW*	**Training**	MgMW	*PgMW*
MgMW + McGW	100.00	99.02	*92.38**	MgMW	97.79	*97.94**
PgMW + McGW	100.00	96.11*	*99.02*	*PgMW*	97.86*	*99.26*
MgMW + PgGW + McGW	***99.71***			MgMW + *PgMW*	***99.15***	

Table 4: Evaluation of classification on the *McAfee Goodware* (McGW), *Malgenome* (MgMW), and *PRAGuard* (Malgenome obfuscated–*PgMG*) dataset with feature filtering and using hold-out validation (*100% hold-out ratio, otherwise we use the hold-out ration described in Table 2b).

4.5 Efficiency

A main design point for DroidSieve was to allow computationally inexpensive feature extraction. Figure 4 shows the runtime for feature extraction on the Marvin dataset, which contains more than 100,000 samples. The median lies at just 2.53 seconds for processing one sample on a single core (Intel Xeon E5-2697 v3 @ 2.60GHz). The overall time for feature extraction on the Marvin dataset took less than 8 hours when executed in parallel on 40 cores.

Other approaches that have been proposed and shown effective for obfuscation-resilient Android malware detection are based on analyzing information flow [7, 16]. However, information flow analysis requires running times that are several orders of magnitude above those seen in DroidSieve's feature extraction. In particular, when attempting to process the 5,560 samples of the Drebin dataset with FlowDroid [6], we were only able to finish half the dataset within three days. Hence, we believe DroidSieve to be better suited for deployment of obfuscation-resilient detection at scale.

For the sake of simplicity, in this section we only report the runtime for feature extraction, we refer the reader to [12] additional details on the running times for training and testing the underlying classification algorithms. As expected, the time taken to process the test samples is negligible.

5 Limitations

The techniques proposed in this paper use lightweight code and resource parsing to build features. Our framework is not built on traditional program analysis techniques such as flow analysis or proofs of non-interference but on mining patterns for API invocations in individual apps. Additionally, we collect lightweight statistics on resources which are derived using simple heuristics which do not required involved program/resource analysis. Consequently, our analysis scales to a large number of apps without hitting performance bottlenecks such as those observed in flow analysis. Having said that, our techniques may not be robust against mimicry attacks as the features that we mine can be contrived. For contemporary mobile malware, though, our features seem to work well. For more sophisticated attacks, it may be necessary to explore formal program analysis techniques.

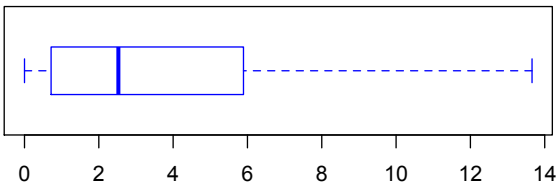


Figure 4: Frequency distribution of running times for feature extraction, in seconds. Most samples require less than six seconds to be analyzed.

Some of our features are built on the use of cryptography in the app. For building such features, we used a two-pronged approach. Firstly, we check the entropy of resources of the app and secondly, we parse the DEX files to identify any calls to the Java cryptography libraries. We expect the conjunction of these features to be indicative of packing and consequently, obfuscation. However, it is possible that the app uses its own libraries or a weaker encryption scheme that produces little observable difference in entropy. In such cases, we would only be relying on features from the DEX to identify uses of cryptography thus weakening our feature engineering.

The performance of the classifier might naturally degrade over time as malware becomes increasingly sophisticated. This phenomenon is commonly referred to as *concept drift*. Our system does not take into account concept drift and we do not provide any retraining strategies that would fortify the system against it. Having said that, dealing with concept drift is crucial in the face of obfuscation. While concept drift is well-known problem in machine learning, it is only recently that approaches have been proposed to provide retraining guidance for malware detection [12]. Whether these approaches can be adapted to detect concept drift in the presence of obfuscated malware remains unknown. In this paper, we argue that it is the responsibility of analyst to purge obsolete features from the model while retraining and incorporate novel features when needed.

There are a number of evasion techniques against machine learning that can potentially target the learning process and the models rather than feature extraction phase. These kind of threats are commonly known as adversarial machine learning attacks [20]. We can find a number of practical techniques to evaluate the robustness of classifiers under attack [8, 17]. Additional experiments would be needed to make a quantitative evaluation of our system in an adversarial environment.

6 Related Work

We now give an overview of the most relevant Android malware detection and classification techniques (see Table 5 for a summary).

Malware Detection. Several systems perform Android malware detection, i.e., perform binary classification [1, 5]. DroidAPIMiner [1] is a detection system based on features generated at API level. Drebin [5] is a lightweight detec-

Year	Method	Type		Feature	# Malware	DR/FP(%)	ACC(%)	Time(s)	Environment
		Det	**Class**						
2014	Dendroid [35]	–	✓	CFG	1,247	–	94	–	
2014	DroidAPIMiner [1]	✓	–	API,PKG,PAR	3,987	99/2.2	–	15	Core i5,6G RAM
2014	DroidMiner [38]	✓	✓	CG,API	2,466	95.3/0.4	92	19.8	–
2014	Drebin [5]	✓	–	PER,STR,API,INT	5,560	94.0/1.0	–	0.75	Core 2 Duo, 4G RAM
2014	DroidSIFT [39]	✓	✓	API-F	2,200	98.0/5.15	93	175.8	Xeon, 128G RAM
2014	DroidLegacy [13]	✓	✓	API	1,052	93.0/3.0	98	–	–
2015	AppAudit [37]	✓	–	API-F	1,005	99.3/0.61	–	0.6	Core i7, 8G RAM
2015	MudFlow [7]	✓	–	API-F	10,552	90.1/18.7	–	–	–
2015	Marvin [21]	✓	–	PER, INT, ST, PN	15,741	98.24/0.0	–	–	–
2015	RevealDroid [16]	✓	✓	PER,API,API-F,INT,PKG	9,054	98.2/18.7	93	95.2	8-Core, 64G RAM
2016	DroidScribe [11]	–	✓	SYSC, BIND, FILE, NET	5,246	–	94	–	–
2016	Madam [30]	✓	–	SYSC, API, PER, SMS, USR	2,800	96/0.2	–	–	–
Ours	**DroidSieve**	✓	✓	*As described in §3*	16,141	**99.3/0.0**	**99**	2.5	40-Core Xeon, 378G RAM

`API`: Application Programming Interface, `API-F`: Information Flow between APIs, `INT`: Intents, `CG`: Call Graph, `PER`: Requested Permissions, `CFG`: Control Flow Graph, `STR`: Embedded strings, `PKG`: Package information of API, `ST`: Statistical features, `PN`: Package names, `SYSC`: System calls, `BIND`: Binder transactions, `FILE`: Filesystem Transactions, `NET`: Network Transactions, `USR`: User Activity, `SMS`: SMS Monitoring

Table 5: Static analysis techniques on Android malware. Results are reported based on the most representative setting. (Almost all of the systems have difficulty against reflection as they are mostly based on API). The performance time of different systems is subjected to specification of computing environments.

tion method that uses static analysis to gather the most important characteristics of Android applications such as permissions, API calls, and network addresses declared in clear text. It uses machine learning (Support Vector Machines) to detect whether a given sample is malicious or benign. DroidSIFT [39] builds contextual API dependency graphs that provide an abstracted view of the possible behaviors of malware and employs machine learning and graph similarity to detect malicious applications. MudFlow [7] and AppAudit [37], however, leverage the analysis of flows between APIs to detect malware.

The main weakness of semantics-based static analysis is that it generally shows poor performance against encryption, reflection, native code, and other cross-platform code such as HTML5. These drawbacks motivate dynamic analysis and hybrid approaches [21, 30]. Marvin [21] shows how the combination of static and dynamic analysis can improve the detection rate as well as reducing the number of false positives. It uses a number of statically extracted features and combines them with additional dynamically extracted features, overall more than 490,000. Moreover, it leverages machine learning to detect malware as well as providing a risk score associated with a given unknown sample. Madam [30] proposed a host-based malware detection system that analyzed features at four levels: kernel, application, user and package. It derived features such as system calls, sensitive API calls and SMS through dynamic analysis while complementing these with statically derived features such as permissions, the app's metadata and market information.

Malware Family Classification. In addition to malware detection systems, a number of methods have been proposed just for classification [35, 11] and others [38, 39] evaluated the features used by their detection system for classification. DroidLegacy [13] is a system using API signature similarity to detect and classify malware. Dendroid [35] proposed an approach based on text mining to automatically classify malware samples and analyze families based on the control flow structures found in them. Similarly, RevealDroid [16] aims at identifying Android malware families. Their approach uses information flow analysis and sensitive API flow tracking built on top of two machine learning classifiers, i.e., C4.5 and 1NN. DroidScribe [11] used a purely dynamic approach to malware classification and classified malware into families by observing system calls, Android ICC through the Binder protocol and file/network transactions made by the app. To classify apps that could not be satisfactorily stimulated during dynamic analysis, DroidScribe built on a statistical evaluation framework of the underline machine learning approach [36] to properly trigger a set-based classification scheme that identified the top matching families for a malware sample, given a desired statistical confidence level.

Discussion. We summarize the most prominent static analysis approaches for Android malware analysis tailored to either detection or classification in Table 5. The column `Type` shows whether a system was mainly proposed for detection or classification. The `Feature` column shows the extracted attributes from malware. `# Malware` is the total number of malware considered for evaluation. `DR/FP` refers to the detection rate and false positive rate of a detection system, and `ACC` shows the accuracy of the system when it is applied for malware family classification. `Time` shows the average required time for analysis of every application.
Systems like DroidMiner, DroidAPIMiner and Drebin are mainly based on APIs, which are inherently vulnerable to reflection. API-flow based approaches like RevealDroid, AppAudit, MudFlow, and DroidSIFT are more precise and consider features related to the semantics of application, but they are still vulnerable to reflection. Furthermore, flow extraction is expensive unless done in the manner of AppAudit where efficiency is derived from incomplete flow coverage.

In contrast, our system is robust against obfuscation techniques like reflection and encryption while still being computationally efficient. Authors in [3] also evaluate their system against common types of obfuscation. However, we evaluate our system on a wide variety of datasets and combinations to avoid unreliable performance measurements. Specifically, we use more than 100,000 goodware apps and over 17,000 malware apps, while authors in [3] limit their evaluation to 207 goodware apps and 1,192 malware apps. Additionally, while past studies focus on a smaller set of behaviors, our method

encompasses a larger set of characteristics and behaviors to distinguish goodware from malware and to identify Android malware families more effectively.

Finally, Roy et al. [29] discuss design choices for evaluating detection systems. Going forward, we plan on taking their important lessons into account. As of now, the focus in DroidSieve lies decidedly on comparing our novel feature engineering for potentially obfuscated malware against existing results in their published settings.

7 Conclusion

In this paper, we have presented a fast, scalable, and accurate system for Android malware detection and family identification based on lightweight static analysis. DroidSieve uses deep inspection of Android malware to build effective and robust features suitable for computational learning. This is key in scenarios where security analysts require intelligent instruments to automate detection and further analysis of Android malware.

We have introduced a novel set of characteristics and showed the importance of systematic feature engineering to achieve a diversified and large range of features that can adjust to different malware. Our findings show that static analysis for Android can succeed even when confronted with obfuscation techniques such as reflection, encryption and dynamically-loaded native code. While fundamental changes in characteristics of malware remain a largely open problem, we showed that DroidSieve remains resilient against state-of-the-art obfuscation techniques which can be used to quickly derive new and syntactically different malware variants.

Acknowledgments

This research has been partially supported by the UK EPSRC grant EP/L022710/1. We are equally thankful to the anonymous reviewers for their invaluable inputs, comments and suggestions to improve the paper. Moreover, we appreciate VirusTotal's collaboration for providing access to their private API so that we could query additional information about the malware.

8 References

[1] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *International Conference on Security and Privacy in Communication Networks (SecureComm)*. 2013.

[2] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 183–194, 2016.

[3] A. Ali-Gombe, I. Ahmed, G. G. Richard III, and V. Roussev. Opseq: Android malware fingerprinting. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, page 7. ACM, 2015.

[4] A. Apvrille and R. Nigam. Obfuscation in android malware, and how to fight back. *Virus Bulletin*, pages 1–10, 2014.

[5] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of Android malware in your pocket. In *Network and Distributed System Security Symposium (NDSS)*. 2014.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 259–269, New York, NY, USA, 2014. ACM.

[7] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *37th International Conference on Software Engineering (ICSE)*, 2015.

[8] B. Biggio, G. Fumera, and F. Roli. Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering*, 26(4):984–996, April 2014.

[9] Z. Cai and R. H. Yap. Inferring the detection logic and evaluating the effectiveness of android anti-virus apps. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 172–182, 2016.

[10] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report TR148, Department of Computer Science, University of Auckland, 1997.

[11] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro. Droidscribe: Classifying android malware based on runtime behavior. In *Mobile Security Technologies (MoST 2016)*, 2016.

[12] A. Deo, S. K. Dash, G. Suarez-Tangil, V. Vovk, and L. Cavallaro. Prescience: Probabilistic guidance on the retraining conundrum for malware detection. In *ACM Workshop on Artificial Intelligence and Security (AISec)*, 2016.

[13] L. Deshotels, V. Notani, and A. Lakhotia. DroidLegacy: Automated familial classification of Android malware. In *ACM SIGPLAN on Program Protection and Reverse Engineering Workshop (PPREW)*, 2014.

[14] T. Dietterich. Overfitting and undercomputing in machine learning. *ACM Computing Surveys (CSUR)*, 27(3):326–327, Sept. 1995.

[15] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we need hundreds of classifiers to solve real world classification problems? *The Journal of Machine Learning Research (JMLR)*, 15(1):3133–3181, Jan. 2014.

[16] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. Technical report, Dept. of Computer Science, George Mason University, 2015.

[17] J. Gardiner and S. Nagaraja. On the security of machine learning in malware c&c detection: A survey. *ACM Comput. Surv.*, 49(3):59:1–59:39, Dec. 2016.

[18] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006.

[19] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference, and prediction: with 200 full-color illustrations*. New York: Springer-Verlag, 2 edition, 2009.

[20] L. Huang, A. D. Joseph, B. Nelson, B. I. P. Rubinstein,

and J. D. Tygar. Adversarial machine learning. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, AISec '11, pages 43–58. ACM, 2011.

[21] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Proceedings of the 39th Annual International Computers, Software & Applications Conference (COMPSAC)*, volume 2, pages 422–433, July 2015.

[22] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis-1,000,000 apps later: A view on current android malware behaviors. In *3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[23] G. Louppe, L. Wehenkel, A. Sutera, and P. Geurts. Understanding variable importances in forests of randomized trees. In *Advances in Neural Information Processing Systems (NIPS)*, pages 431–439, 2013.

[24] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16 – 31, 2015.

[25] M. Palatucci, D. Pomerleau, G. E. Hinton, and T. M. Mitchell. Zero-shot learning with semantic output codes. In *Advances in neural information processing systems (NIPS)*, pages 1410–1418, 2009.

[26] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb 2014.

[27] R. B. Rao, G. Fung, and R. Rosales. On the dangers of cross-validation. an experimental evaluation. In *SIAM International Conference on Data Mining (SDM)*, pages 588–596. SIAM, 2008.

[28] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 329–334, New York, NY, USA, 2013. ACM.

[29] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara. Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 81–90. ACM, 2015.

[30] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1–1, 2016.

[31] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys*, 49(1), Apr. 2016.

[32] C. Smutz and A. Stavrou. Malicious pdf detection using metadata and structural features. In *28th Annual Computer Security Applications Conference (ACSAC)*, pages 239–248, New York, NY, USA, 2012. ACM.

[33] G. Suarez-Tangil, J. Tapiador, F. Lombardi, and R. Di Pietro. Alterdroid: Differential fault analysis of obfuscated smartphone malware. 2016.

[34] G. Suarez-Tangil, J. E. Tapiador, and P. Peris-Lopez. Stegomalware: Playing hide and seek with malicious components in smartphone apps. In *10th International Conference on Information Security and Cryptology (Inscrypt)*, pages 496–515. Springer, December 2014.

[35] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(1):1104–1117, 2014.

[36] R. J. Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Misleading Metrics: On Evaluating Machine Learning for Malware with Confidence. Technical Report 2016-1, Royal Holloway, University of London, 2016.

[37] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *IEEE Symposium on Security and Privacy (SP)*, pages 899–914, May 2015.

[38] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. A. Porras. DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *European Symposium on Research in Computer Security (ESORICS)*. 2014.

[39] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In *ACM SIGSAC Computer and Communications Security (CCS)*, 2014.

[40] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in Android apps with permission use analysis. In *ACM SIGSAC Computer and Communications Security (CCS)*, 2013.

[41] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (SP)*, 2012.