Explaining Black-box Android Malware Detection

Marco Melis*, Davide Maiorca*, Battista Biggio*†, Giorgio Giacinto*† and Fabio Roli*†
*DIEE, University of Cagliari, Piazza d'Armi, 09123, Cagliari
{marco.melis,davide.maiorca,battista.biggio,giacinto,roli}@diee.unica.it
† Pluribus One, Italy

***Abstract*—Machine-learning models have been recently used for detecting malicious Android applications, reporting impressive performances on benchmark datasets, even when trained only on features statically extracted from the application, such as system calls and permissions. However, recent findings have highlighted the fragility of such in-vitro evaluations with benchmark datasets, showing that very few changes to the content of Android malware may suffice to evade detection. How can we thus trust that a malware detector performing well on benchmark data will continue to do so when deployed in an operating environment? To mitigate this issue, the most popular Android malware detectors use linear, explainable machine-learning models to easily identify the most influential features contributing to each decision. In this work, we generalize this approach to any black-box machine-learning model, by leveraging a gradient-based approach to identify the most influential local features. This enables using nonlinear models to potentially increase accuracy without sacrificing interpretability of decisions. Our approach also highlights the global characteristics learned by the model to discriminate between benign and malware applications. Finally, as shown by our empirical analysis on a popular Android malware detection task, it also helps identifying potential vulnerabilities of linear and nonlinear models against adversarial manipulations.**

I. Introduction

With more than 400 millions of malicious applications discovered in the wild, Android malware constitutes one of the major threats in mobile security. Among the various detection strategies proposed by companies and academic researchers, those based on machine learning have shown the most promising results, due to their flexibility against malware variants and obfuscation attempts [1], [7]. Despite the impressive performances reported by such approaches on benchmark datasets, the problem of Android malware detection in the wild is still far from being solved. The validity of such optimistic, in-vitro evaluations has been indeed questioned from recent adversarial analyses showing that only few changes to the content of a malicious Android application may suffice to evade detection by a learning-based detector [6], [8]. Besides this fragility to well-crafted evasion attacks (a.k.a. adversarial examples) [4], [5], [10], [17], Sommer and Paxson [16] have more generally questioned the suitability of black-box machine-learning approaches to computer security. In particular, how can we thus trust the predictions of a machine-learning model *in vivo*, i.e., when it is deployed in an operating environment, to take subsequent reliable actions? How can we understand whether we are selecting a proper model before deployment? How about its security properties against adversarial attacks?

To partially address these issues, the most popular Android malware detectors restrict themselves to the use of *linear*, *explainable* machine-learning models that allow one to easily identify the most influential features contributing to each decision (Sect. II) [1], [2]. More generally, *intepretability* of machine-learning models has recently become a relevant research direction to more thoroughly address and mitigate the aforementioned issues, especially in the case of *nonlinear black-box* machine-learning algorithms [3], [9], [12]–[14]. Some approaches aim to explain *local* predictions (i.e., on each specific sample) by identifying the most influential features [3], [14] or prototypes from training data [12]. Others have proposed techniques and methodologies towards providing *global explanations* about the salient characteristics learned by a given machine-learning algorithm [9], [13].

In this work, we generalize current explainable Android malware detection approaches to *any* black-box machine-learning model, by leveraging a gradient-based approach to identify the most influential *local* features (Sect. III). For non-differentiable learning algorithms, like decision trees, we extract gradient information by learning a differentiable approximation. Notably, this idea has originally been exploited to construct gradient-based evasion attacks against non-differentiable learners, and evaluate their *transferability*, i.e., the probability that an attack crafted against a learning algorithm succeeds against a different one [4], [10], [15]. Accordingly, our approach enables the use of any nonlinear learning algorithm to potentially increase accuracy of current Android malware detectors without necessarily sacrificing interpretability of decisions. Moreover, by averaging the local relevant features across different classes of samples, our approach allows also highlighting the *global* characteristics learned by a given model to identify benign applications and different classes of Android malware.

We perform our experimental analysis with a popular Android malware detector named Drebin [1] (Sect. IV). It extracts information from multiple parts of the Android application through a static analysis, and provides interpretable decisions by leveraging a linear classification algorithm. To test the validity of our approach, we show how to retain the interpretability of Drebin on nonlinear learning algorithms, including Support Vector Machines (SVMs) and Random Forests (RFs). Interestingly, we also show that the interpretations provided by our approach can help identifying potential vulnerabilities of both linear and nonlinear Android malware detectors against adversarial manipulations.

We conclude the paper by discussing contributions and limitations of this work, and future research directions towards

developing more robust features and models for malware detection (Sect. V).

II. Android Malware Detection

In this section, we provide some background on how Android applications are structured, and then discuss Drebin [1], the state-of-the-art malware detector used in our analysis.

A. Android Background

Android applications are `apk` files, i.e., zipped archives that must contain two files: the Android `manifest` and the classes.dex. Additional `xml` and resource files are respectively used to define the application layout and to provide multimedia contents. As Drebin only analyzes the Android `manifest` and classes.dex files, we briefly describe them below.

Android Manifest. The `manifest` file holds information about how the application is organized in terms of its *components*, i.e., parts of code that perform specific actions; e.g., one component might be associated to a screen visualized by the user (*activity*) or to the execution of audio in the background (*services*). It is also possible to perform actions on the occurrence of a specific event (*receivers*). The actions of each component are further specified through *filtered intents*; e.g., when a component sends data to other applications, or is invoked by a browser. Special types of intent filters (e.g., LAUNCHER) can specify that a certain component is executed as soon as the application is opened. The `manifest` also contains the list of *hardware components* and *permissions* requested by the application to work (e.g., Internet access).

Dalvik Bytecode (`dexcode`). The classes.dex file embeds the compiled source code of an application, including all the user-implemented methods and classes. Classes.dex may contain specific API calls that can access sensitive resources such as personal contacts (*suspicious calls*). Additionally, it contains all system-related, *restricted API calls* whose functionality require *permissions* (e.g., using the Internet). Finally, this file can contain references to *network addresses* that might be contacted by the application.

B. Drebin

Drebin performs a lightweight static analysis of Android applications. The extracted features are used to embed benign and malware apps into a high-dimensional vector space, train a machine-learning model, and then perform classification of never-before-seen apps. An overview of the system architecture is given in Fig. 1, and discussed more in detail below.

Feature extraction. First, Drebin statically analyzes a set of available Android applications to construct a suitable feature space. All features extracted by Drebin are presented as *strings* and organized in 8 different feature sets, as listed in Table I. Android applications are then mapped onto the feature space as follows. Let us assume that an app is represented as an object $\boldsymbol{z} \in \mathcal{Z}$, being $\mathcal{Z}$ the abstract space of all `apk` files. We then denote with $\Phi : \mathcal{Z} \mapsto \mathcal{X}$ a function that maps an `apk` file $\boldsymbol{z}$ to a d-dimensional feature vector $\boldsymbol{x} = (x^1, \ldots, x^{\mathsf{d}})^\top \in \mathcal{X} = \{0,1\}^{\mathsf{d}}$, where each feature is set to 1 (0) if the corresponding *string* is present (absent) in the `apk` file $\boldsymbol{z}$. An application encoded in feature space may thus look like the following:

TABLE I
Overview of feature sets.

Feature sets		
`manifest`	S_1	Hardware components
	S_2	Requested permissions
	S_3	Application components
	S_4	Filtered intents
`dexcode`	S_5	Restricted API calls
	S_6	Used permission
	S_7	Suspicious API calls
	S_8	Network addresses

$$\boldsymbol{x} = \Phi(\boldsymbol{z}) \mapsto \begin{pmatrix} \cdots \\ 0 \\ 1 \\ \cdots \\ 1 \\ 0 \\ \cdots \end{pmatrix} \begin{matrix} \cdots \\ \texttt{permission::SEND\_SMS} \\ \texttt{permission::READ\_SMS} \\ \cdots \\ \texttt{api\_call::getDeviceId} \\ \texttt{api\_call::getSubscriberId} \\ \cdots \end{matrix} \begin{matrix} \\ \left.\begin{matrix} \\ \\ \end{matrix}\right\} S_2 \\ \\ \left.\begin{matrix} \\ \\ \end{matrix}\right\} S_5 \\ \\ \end{matrix}$$

Learning and Classification. Drebin uses a linear SVM to perform detection. It can be expressed in terms of a linear function $f : \mathcal{X} \mapsto \mathbb{R}$, as:

$$f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b\,, \tag{1}$$

where $\boldsymbol{w} \in \mathbb{R}^{\mathsf{d}}$ denotes the vector of *feature weights*, and $b \in \mathbb{R}$ is the so-called *bias*. These parameters, optimized during training, identify a hyperplane that separates the two classes in feature space. During classification, unseen apps are then classified as malware if $f(\boldsymbol{x}) \geq 0$, and as benign otherwise.

Explanation. Drebin explains its decisions by reporting, for any given application, the most influential features, i.e., the features that are present in the given application and are assigned the highest absolute weights by the classifier. For instance, in Fig. 1, it is easy to see, from its most influential features, that a malware sample is correctly identified by Drebin as it connects to a suspicious URL and uses SMS as a side channel for communication. As we aim to extend this approach to nonlinear models, in this work we also consider an SVM with the Radial Basis Function (RBF) kernel and a random forest to learn nonlinear functions $f(\boldsymbol{x})$.

III. Interpreting Decisions of Learning-based Black-box Android Malware Detectors

We discuss here our idea to generalize the explainable decisions of Drebin and other locally-explainable Android malware detectors [1], [2] to any black-box (i.e., nonlinear) machine-learning algorithm. In addition, we also propose a method to explain the *global* characteristics influencing the decisions of the learning-based malware detector at hand.

Local explanations. Previous work has highlighted that gradients and, more generally, linear approximations computed around the input point $\boldsymbol{x}$ convey useful information for explaining the local predictions provided by a learning algorithm [3], [14]. The underlying idea is to identify as *most*

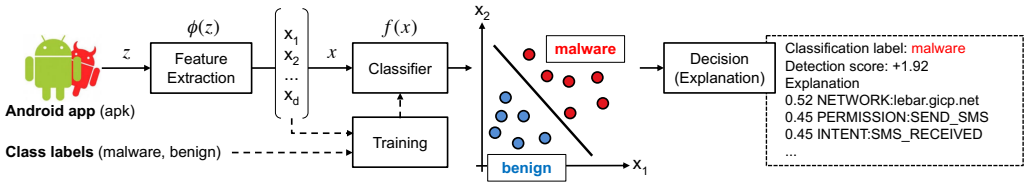

Fig. 1. A schematic representation of Drebin, adapted from [8]. First, applications are represented as binary vectors in a d-dimensional feature space. A linear classifier is then trained on an available set of malware and benign applications, assigning a weight to each feature. During classification, unseen applications are scored by the classifier by summing up the weights of the present features: if $f(\boldsymbol{x}) \geq 0$, they are classified as malware. Drebin also explain each decision by reporting the most suspicious (or benign) features present in the app, along with the weight assigned to them by the linear classifier [1].

influential those features associated to the highest (absolute) values of the local gradient $\nabla f(\boldsymbol{x})$, being f the confidence associated to the predicted class. However, in the case of sparse data, as for Android malware, these approaches tend to identify a high number of influential features which are *not* present in the given application, thus making the corresponding predictions difficult to interpret. For this reason, in this work we consider a slightly different approach, inspired from the notion of directional derivative. In particular, we project the gradient $\nabla f(\boldsymbol{x})$ onto $\boldsymbol{x}$ to obtain a *feature-relevance* vector $\boldsymbol{\nu} = \nabla f(\boldsymbol{x}) \cdot \boldsymbol{x} \in \mathbb{R}^{\mathsf{d}}$, where $\cdot$ denotes the element-wise product. We then normalize $\boldsymbol{\nu}$ to have a unary ℓ_1 norm, i.e., $\boldsymbol{r} = \boldsymbol{\nu}/\|\boldsymbol{\nu}\|_1$, to ensure that only non-null features in $\boldsymbol{x}$ are identified as relevant for the decision. Finally, the absolute values of $\boldsymbol{r}$ can be ranked in descending order to identify the most influential *local* features.

Global explanations. In contrast to other locally-explainable malware detectors [1], [2], we also provide a *global* analysis of the interpretability of the considered machine-learning models, aimed to identify the most influential features, on average, which characterize benign and malware samples. Our idea is simply to average the relevance vectors $\boldsymbol{r}$ over different samples, e.g., separately for benign and malware data. Then, as in the local case, the absolute values of the average relevance vector $\bar{\boldsymbol{r}}$ can be ranked in descending order to identify the most influential *global* features.

Non-differentiable models. Our approach works under the assumption that $f(\boldsymbol{x})$ is *differentiable* and that its gradient $\nabla f(\boldsymbol{x})$ is sufficiently smooth to provide meaningful information at each point. When $f(\boldsymbol{x})$ is not differentiable (e.g., for decision trees and random forests), or its gradient vanishes (e.g., if $f(\boldsymbol{x})$ becomes constant in large regions of the input space), we compute approximate feature-relevance vectors by means of surrogate models. The idea is to train a differentiable approximation $\hat{f}(\boldsymbol{x})$ of the target function $f(\boldsymbol{x})$, similar to what has been done in [3] for interpretability of non-differentiable models, and in [4], [15] to craft gradient-based evasion attacks against non-differentiable learning algorithms. For instance, to reliably estimate a non-differentiable algorithm $f(\boldsymbol{x})$ (e.g., a random forest), one can train a nonlinear SVM on a training set relabeled with the predictions provided by $f(\boldsymbol{x})$ [15].

IV. Experimental Analysis

In this section, we use our approach to provide local and global explanations of linear and nonlinear (including non-differentiable) classifiers on the feature representation used by Drebin. As we will see, this also reveals interesting insights on the security properties of such algorithms against adversarial manipulations [6], [8].

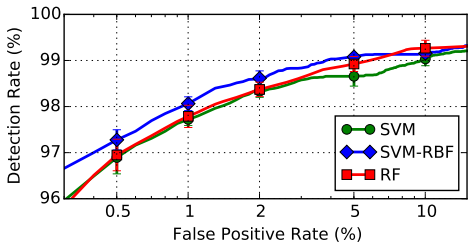

Fig. 2. Average ROC curves for the given classifiers on the *Drebin* data.

Datasets. We use here the *Drebin* data [1], consisting of $121,329$ benign applications and $5,615$ malicious samples, labeled with VirusTotal. A sample is labeled as malicious if it is detected by at least five anti-virus scanners, whereas it is labeled as benign if no scanner flagged it as malware.

Training-test splits. We average our results on 5 runs. In each run, we randomly select 60,000 apps from the *Drebin* data to train the learning algorithms, and use the rest for testing.

Classifiers. We compare the standard *Drebin* implementation based on a linear SVM (SVM) against an SVM with the RBF kernel (SVM-RBF) and a (non-differentiable) Random Forest (RF). As discussed in Sect. III, a surrogate model is needed to interpret the RF; to this end, we train an SVM with the RBF kernel on the training set relabeled by the RF (yielding an approximation with accuracy higher than 99% on average on the relabeled testing sets). The Receiver Operating Characteristic (ROC) curve for each classifier, averaged over the 5 repetitions, is reported in Fig. 2.

Parameter setting. We optimize the parameters of each classifier through a 3-fold cross-validation procedure. In particular, we optimize $C \in \{10^{-2}, 10^{-1}, \ldots, 10^{2}\}$ for both linear and non-linear SVMs, the kernel parameter $\gamma \in \{10^{-4}, 10^{-3}, \ldots, 10^{2}\}$ for the SVM-RBF, and the number of estimators $n \in \{5, 10, \ldots, 30\}$ for the RF.

A. Local Explanations

Table II reports the top-10 influential features, sorted by their (absolute) *relevance* values, for three distinct samples classified by the linear SVM and the RF classifier, along with their probability of being present in each class. Notably, relevant features can also be *rare*. This means that a feature

TABLE II

TOP-10 INFLUENTIAL FEATURES FOR SVM (TOP ROW) AND RF (BOTTOM ROW) ON (i) A BENIGN SAMPLE (FIRST COLUMN), (ii) A MALWARE SAMPLE OF THE SMSWATCHER FAMILY (SECOND COLUMN), AND (iii) A MALWARE SAMPLE OF THE PLANKTON FAMILY (THIRD COLUMN). THE PROBABILITY OF EACH FEATURE BEING PRESENT IN BENING (p_B) AND MALWARE (p_M) IS ALSO REPORTED.

Set	Feature Name	r (%)	p_B (%)	p_M (%)
S2	SEND_SMS	26.89	3.19	53.89
S4	LAUNCHER	-15.40	96.42	93.56
S6	SEND_SMS	9.42	3.11	44.76
S2	GET_ACCOUNTS	8.61	2.57	8.06
S8	code.google.com	6.38	1.10	1.73
S7	Ljava/io/IOException;->printStackTrace	6.09	49.82	66.85
S2	READ_CONTACTS	-4.61	7.25	23.75
S2	INTERNET	4.30	83.29	96.26
S8	ajax.googleapis.com	-3.19	0.76	0.54
S4	android.intent.action.MAIN	2.91	97.52	95.88

Set	Feature Name	r (%)	p_B (%)	p_M (%)
S2	SEND_SMS	25.82	3.19	53.89
S4	LAUNCHER	-18.49	96.42	93.56
S2	READ_CONTACTS	-10.24	7.25	23.75
S7	Ljava/io/IOException;->printStackTrace	7.90	49.82	66.85
S5	android/telephony/SmsManager;->sendTextMessage	7.75	1.77	34.73
S7	android/telephony/SmsManager;->sendTextMessage	7.65	1.77	34.73
S6	INTERNET	-4.43	77.74	85.43
S8	ajax.googleapis.com	-2.88	0.76	0.54
S5	android/telephony/SmsManager;->getDefault	1.77	2.01	37.63
S7	android/telephony/SmsManager;->getDefault	1.66	2.01	37.63

Set	Feature Name	r (%)	p_B (%)	p_M (%)
S2	SEND_SMS	10.94	3.19	53.89
S3	com.rjblackbox.swl.SMSActivity	9.72	0.00	0.04
S3	com.rjblackbox.swl.SMSForwarder	9.72	0.00	0.04
S3	com.rjblackbox.swl.SettingsActivity	9.72	0.00	0.04
S4	android.provider.Telephony.SMS_RECEIVED	8.13	1.09	20.08
S4	LAUNCHER	-6.26	96.42	93.56
S2	RECEIVE_SMS	-4.82	2.43	38.36
S6	SEND_SMS	3.83	3.11	44.76
S7	Lorg/apache/http/client/methods/HttpPost	3.52	29.95	51.89
S7	android/telephony/SmsMessage;->createFromPdu	-3.51	1.44	16.19

Set	Feature Name	r (%)	p_B (%)	p_M (%)
S2	SEND_SMS	14.04	3.19	53.89
S4	LAUNCHER	-13.65	96.42	93.56
S4	SMS_RECEIVED	8.39	1.09	20.08
S2	RECEIVE_SMS	-8.02	2.43	38.36
S7	android/net/Uri;->withAppendedPath	-6.96	9.24	16.96
S5	LocationManager;->getLastKnownLocation	-6.45	27.09	31.65
S7	Lorg/apache/http/client/methods/HttpPost	4.80	29.95	51.89
S7	android/net/Uri;->encode	-4.64	9.52	8.17
S7	getPackageInfo	-3.74	53.80	49.50
S2	READ_CONTACTS	-3.57	7.25	23.75

Set	Feature Name	r (%)	p_B (%)	p_M (%)
S7	TelephonyManager;->getNetworkOperator	3.00	6.01	46.57
S4	LAUNCHER	-2.50	96.42	93.56
S7	TelephonyManager;->getNetworkOperatorName	-2.46	5.08	28.99
S6	ACCESS_NETWORK_STATE	-2.32	47.92	56.40
S7	android/net/Uri;->fromFile	2.13	16.81	43.10
S2	INSTALL_SHORTCUT (launcher)	2.04	1.51	26.37
S2	READ_HISTORY_BOOKMARKS (browser)	1.73	0.52	17.89
S5	LocationManager;->isProviderEnabled	-1.70	12.53	17.12
S7	com.apperhand.device.android.AndroidSDKProvider	1.70	0.00	10.95
S7	java/lang/reflect/Method;->getReturnType	-1.52	5.97	12.22

Set	Feature Name	r (%)	p_B (%)	p_M (%)
S4	LAUNCHER	-2.75	96.42	93.56
S2	INSTALL_SHORTCUT (launcher)	2.19	1.51	26.37
S2	ACCESS_WIFI_STATE	1.81	10.59	43.10
S7	TelephonyManager;->getNetworkOperator	1.74	6.01	46.57
S5	Contacts$People;->createPersonInMyContactsGroup	-1.64	3.53	0.89
S6	READ_CONTACTS	-1.55	12.89	7.79
S4	BOOT_COMPLETED	1.51	6.73	66.08
S2	WRITE_SETTINGS	-1.49	3.67	12.34
S5	LocationManager;->isProviderEnabled	-1.48	12.53	17.12
S7	android/net/Uri;->encode	-1.44	9.52	8.17

is deemed relevant even if it characterizes well only a small subset of samples in a given class (e.g., a malware family).

Case 1. The first example is a benign application misclassified by the SVM, and correctly classified by the RF. By observing the features through their relevance scores, it is evident that the RF is able to correctly classify this sample as benign as several features are assigned a negative relevance score, while almost all of them are considered as malicious (positive score) by the SVM. In both cases the use of SMS messages for communication is retained suspicious; however, for the RF this is not a sufficient evidence of maliciousness.

Case 2. The second example is a malware sample of the `SmsWatcher` family, which is correctly classified by SVM, but not by the RF model, for a reason similar to the previous case: permissions (S_2) and API calls (S_7) related to SMS usage are not a sufficient evidence of maliciousness for the RF. Indeed, this classifier does not even identify as suspicious the application components (S_3) related to SMS usage, which instead constitute a *signature* for this malware family, as correctly learned by the linear SVM model.

Case 3. The last case is a malware sample of the `Plankton` family, correctly classified by both models, as they correctly identified the behavioral patterns of this family associated to HTTP communication and actions.

B. Global Explanations

We performed a global analysis of the models learned by each algorithm by averaging the local relevance vectors $\boldsymbol{r}$ over different classes of samples: benign, malware, and the top-15 malware families with the largest number of samples in the *Drebin* data (Table III). This gives us a global (mean) relevance vector $\bar{\boldsymbol{r}}$ for each class. Then, for each class of samples, we report a *compact* and a *fine-grained* analysis of the *global* feature-relevance values $\bar{\boldsymbol{r}}$. In the *compact* analysis, we further average the global relevance $\bar{\boldsymbol{r}}$ over each *feature set* $S_1, \ldots, S_2$ (Table I). In the *fine-grained* analysis, we simply report the global relevance score $\bar{\boldsymbol{r}}$ for the top 44 features (selected by aggregating the top 5 features with the highest average relevance score for each class of samples).

The results are shown in Fig. 3. The compact analysis highlights the importance of permissions (S_2) and suspicious API calls (S_7 group) in identifying malware. This is reasonable, as the majority of malware samples require permissions to perform specific actions, like stealing contacts and opening SMS and other side communication channels. The fine-grained analysis provides a more detailed characterization of the aforementioned behavior, highlighting how each classifier learns a specific *behavioral signature* for each class of samples. In particular, malware families are characterized by their communication channels (e.g., SMS and HTTP), by the amount of stolen information and accessed resources, and by specific application components or URLs (S_3 and S_8).

Finally, note that all classifiers tend to assign high relevance to a very small set of features in each decision, both at a local and at a global scale. Given that manipulating the content of Android malware can be relatively easy, especially due to the possibility of injecting dead code, this behavior highlights the potential *vulnerability* of such classifiers. In fact, if the decisions of a classifier rely on few features, it is intuitive that detection can be easily evaded by manipulating only few of them, as also confirmed in previous work [6], [8]. Conversely, if a model distributes relevance more evenly among features, evasion may be more difficult (i.e., require manipulating a higher number of features, which may not be always feasible). More robust learning algorithms for these tasks have been proposed based exactly on this rationale, which has also a theoretically-sound interpretation [8].

Another interesting point regards the *transferability* of evasion attacks across different models, i.e., the fact that an attack crafted against a specific classifier may still be successful with high probability against a different one. From our analysis, it is clear that in this case this property depends more on the available training data rather than on the specific learning algorithm: the three considered classifiers learn very similar patterns of feature relevances, as clearly highlighted in Fig. 3, which simply means that they can be evaded with very similar modifications to the input sample.

V. CONTRIBUTIONS, LIMITATIONS AND FUTURE WORK

In this paper, we provided a general approach to achieve explainable malware detection on Android, applicable to any black-box machine-learning model. Our explainable approach

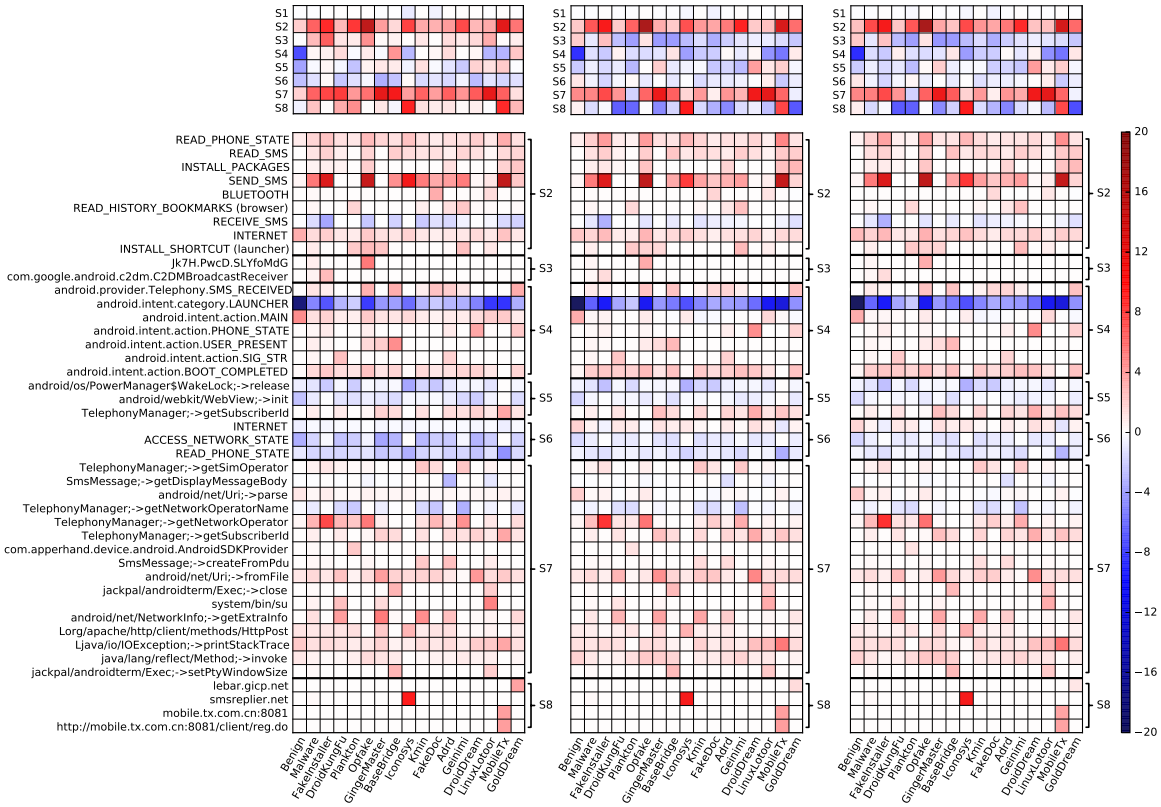

Fig. 3. Average values of the relevance score computed with respect to benign, malware and the top-15 malware families (Table III). The compact representation (*top*) reports feature relevances averaged over the feature sets $S_1, \ldots, S_8$ (Table I). The fine-grained representation (*bottom*) reports relevance values for the top 44 features with the highest average relevance score per family. Positive (negative) relevance values denote malicious (benign) behavior.

TABLE III
TOP 15 MALWARE FAMILIES IN THE TEST SET.

Family	#	Family	#	Family	#
FakeInstaller	901	BaseBridge	318	Geinimi	88
DroidKungFu	640	Iconosys	149	DroidDream	81
Plankton	609	Kmin	144	LinuxLotoor	69
Opfake	591	FakeDoc	128	MobileTx	68
GingerMaster	332	Adrd	88	GoldDream	67

can help analysts to understand possible vulnerabilities of learning algorithms to well-crafted evasion attacks along with their transferability properties, besides providing a local and global understanding of how a machine-learning model makes its decisions. This is a relevant issue also towards the development of interpretable models, as required by the novel European General Data Protection Regulation (GDPR) [11]. The *right of explanation* stated by GDPR imposes to develop models that are transparent with respect to their decisions. We believe that this work is a first step towards this direction.

REFERENCES

[1] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of android malware in your pocket. In *Proc. 21st NDSS*. The Internet Society, 2014.

[2] M. Backes and M. Nauman. LUNA: quantifying and leveraging uncertainty in android malware analysis through bayesian machine learning. In *EuroS&P*, pp. 204–217. IEEE, 2017.

[3] D. Baehrens, T. Schroeter, S. Harmeling, M. Kawanabe, K. Hansen, and K.-R. Müller. How to explain individual classification decisions. *J. Mach. Learn. Res.*, 11:1803–1831, 2010.

[4] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *ECML*, vol. 8190, *LNCS*, pp. 387–402. Springer, 2013.

[5] B. Biggio and F. Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *ArXiv*, 2018.

[6] A. Calleja, A. Martin, H. D. Menendez, J. Tapiador, and D. Clark. Picking on the family: Disrupting android malware triage by forcing misclassification. *Expert Systems with Applications*, 95:113 – 126, 2018.

[7] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu. Stormdroid: A streaminglized machine learning-based system for detecting Android malware. In *ASIA CCS*, pp. 377–388, 2016. ACM.

[8] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! A case study on Android malware detection. *IEEE Trans. Dependable and Secure Computing*, In press.

[9] F. Doshi-Velez and B. Kim. Towards A Rigorous Science of Interpretable Machine Learning. *ArXiv*, 2017.

[10] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[11] B. Goodman and S. Flaxman. European Union regulations on algorithmic decision-making and a "right to explanation". *ArXiv*, 2016.

[12] P. W. Koh and P. Liang. Understanding black-box predictions via influence functions. In *ICML*, 2017.

[13] Z. C. Lipton. The mythos of model interpretability. In *ICML Workshop on Human Interpretability in Machine Learning*, pp. 96–100, 2016.

[14] M. T. Ribeiro, S. Singh, and C. Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *KDD*, pp. 1135–1144, 2016. ACM.

[15] P. Russu, A. Demontis, B. Biggio, G. Fumera, and F. Roli. Secure kernel machines against evasion attacks. In *AISec*, pp. 59–69, 2016. ACM.

[16] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE Symp. Security and Privacy*, pp. 305–316, 2010. IEEE CS.

[17] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *ICLR*, 2014.