Low-Resource Footprint, Data-Driven Malware Detection on Android

Simone Aonzo, Alessio Merlo, Mauro Migliardi, Luca Oneto, Francesco Palmieri

***Abstract*—Resource-constrained systems are becoming more and more common as users migrate from PCs to mobile devices and as IoT systems enter the mainstream. At the same time, it is not acceptable to reduce the level of security hence it is necessary to accommodate the required security into the system-imposed resource constraints. This paper introduces BAdDroIds, a mobile application leveraging machine learning for detecting malware on resource constrained devices. BAdDroIds executes in background and transparently analyzes the applications as soon as they are installed, i.e., before infecting the device. BAdDroIds relies on static analysis techniques and features provided by the Android OS to build up sound and complete models of Android apps in terms of permissions and API invocations. It uses ad-hoc supervised classification techniques to allow resource-efficient malware detection. By exploiting the intrinsic nature of data, it has been possible to implement a state-of-the-art data-driven model which provides deep insights on the detection problem and can be efficiently executed on the device itself as it requires a very limited computational effort. Besides its limited resource footprint, BAdDroIds is extremely effective: an extensive experimental evaluation shows that it outperforms the currently available solutions in terms of accuracy, which is around 99%.**

***Index Terms*—Android Security, Malware Analysis, Supervised Learning, Data-Driven Models, Model Selection, Feature Ranking**

I. Introduction

In the field of computing the problem of sustainability may be tackled from several different angles. A first approach takes into account the problem of reducing the resource consumption of computing centers [1], while a second one is dedicated to the greening of the network infrastructures [2]. At the same time, the need to control the resource consumption cannot interfere with the need to guarantee the desired levels of security [3], [4]. However, as the number of users relying on mobile devices for their daily routines increases and IoT systems enter the mainstream, sustainability cannot be seen only as an effort to reduce the resource footprint of systems that are intrinsically not constrained, but also as the need to develop new methodologies that allow both performing traditionally resource hungry activities on resource constrained devices and reduce the impact of attacks on the energy consumption of the device [5][6][7]. In particular, the problem of providing appropriate security levels without depleting the resources of devices is of paramount importance. To this aim, in this paper we focus on the problem of providing a methodology to detect malicious software on resource constrained devices with a very low resource footprint. Furthermore, to prove the efficacy of our methodology we also provide an actual tool implementing it on a very common mobile platform, namely, the Android operating system. In the mobile world, the number of devices adopting the Android Operating System (hereafter, Android) and depending on online markets to install mobile applications (hereafter, apps) has been steadily growing for years and is still growing at present [8]. Consequently, the number of users relying on an Android-based device to perform common daily tasks is increasing and, because of this fact, the number of malicious apps (hereafter, malware) is likewise raising. As the traditional signature-based mechanism cannot cope with the increasing number of malware variants and polymorphic code exploited in them, in recent years several efforts have been put forward by the research community to define new approaches to malware detection. Among others, the most promising ones rely on data-driven techniques whose aim is to learn how to classify apps into two sets, namely *legal* and *malware* apps, based on the analysis of already classified apps.

This new approach, however, is usually resource hungry and has some drawbacks that may limit its actual applicability. The first issue is that data-driven techniques need to extract meaningful features from the apps that have to be classified. A very rich set of features may allow to build a very precise classification model but it may also overload the device beyond acceptable usability, thereby preventing the implementation of model on the device. For this reason, it is necessary to extract a set of features which is large enough to support the definition of a reliable model, but also small enough to be computed on the resource constrained device in a sustainable way. Another limitation of this kind of models arises when the behavior of the program to be classified is evaluated at runtime (i.e., dynamic analysis). In fact, in such a situation, there is always the risk to recognize a threat only after the system has been compromised: therefore, this kind of analysis is generally carried out off-device, in secure *sandboxed* environments, albeit some proposals for anomaly detection at runtime on mobile have been recently put forward [9]. When dealing with the need of on-device analysis, the safest solution is to rely only on features that may be statically extracted from the program code without the need to execute it (i.e., static analysis). Finally, from a data-scientist point of view, there is the need to deal with the *overfitting* problem that occurs when the model excessively adapts to the set of apps on which the model has been trained, without guaranteeing an adequate generalization performance needed to detect previously unseen

S. Aonzo, L. Oneto and A. Merlo are with the Department of Informatics, Bioengineering, Robotics and Systems Engineering (DIBRIS) at the University of Genoa. E-mail: {name.surname@unige.it}.

M. Migliardi is with the Department of Electrical Engineering at the University of Padua. Email: mauro.migliardi@unipd.it

Francesco Palmieri is with the Department of Computer Science at the University of Salerno. Email: fpalmier@unisa.it.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2017.2774184, IEEE Transactions on Sustainable Computing

apps (i.e., zero-day malware).

In this paper we present BAdDroIds, an Android app capable of analyzing apps as soon as they are installed on the device, thereby allowing to classify an app as legal or malware before its execution, with a high degree of accuracy and a low resource footprint. We adopted state-of-the-art data-driven machine learning techniques for building the malware detection system, in such a way that they can efficiently execute on a resource-constrained mobile device. In detail, we studied both qualitatively and quantitatively how the model works and how malware can be detected. Our tests have been performed by merging the most recently updated malware databases for a total of more than seven thousand malware samples, and have demonstrated to correctly classify the apps in approximately the 99% of the cases. The performance of BAdDroIds suggests that it can be adopted on actual mobile devices to execute on-the-fly analysis of new apps with a very limited impact on the user experience.

Structure of the paper. The rest of the paper is organized as follows: Section II provides some background and discusses some related work, while Section III presents the data-driven model at the basis of BAdDroIds. Section IV discusses the experimental results of BAdDroIds on the field. Finally, V concludes the paper by discussing some future development of BAdDroIds.

II. Related work

The aim of this work is to define a new approach to binary classification of malware on Android through data-driven techniques that take into account as features both the permissions required by apps as well as the Android API (hereafter, AAPI) they invoke. In order to justify the reasons behind this choice, we briefly introduce here some relevant basics on the Android security mechanisms. The main security mechanisms in Android are the *app sandbox* and the usage of *permissions*. Android executes each app in a restricted *sandbox*, built by taking advantage of the multi-user nature of the Linux Kernel. In a nutshell, upon installation each app gets assigned a Linux User ID (UID) and whenever the app executes, it runs in its own user space. The aim of sandboxing is to improve the separation among apps by leveraging the isolation natively granted by the Linux Kernel to different system users. Beyond sandboxing, Android requires that each app declares its resource requirements as a set of *permissions* upon installation. Without loss of precision, permissions can be seen as strings that denote the possibility for an app to require specific functionalities offered by the operating system. Some basic permissions are automatically granted at install time and never revoked. Other permissions (defined as *dangerous* [10]) handle the privacy of the user (i.e., they allow the app to profile the user by accessing his contacts, messages and call logs, as well as activate camera and audio recording or access the user's position) and have to be granted/denied at runtime by the user himself. At the time of writing, there are 24 dangerous permissions in Android; however, these 24 are divided into 9 groups that represent the granularity at which they may be granted/denied. When a dangerous permission is requested by the app for the first time, the user is prompted to grant/deny the use of the permission to the app. As described above, if the user grants the permission, she automatically grants all the permissions in the corresponding group. This implies that if the same app requires to exploit another permission of the same group for the first time, the user is not prompted again. This choice allows limiting the number of permission requests at runtime (that can be annoying for the user), at the cost of a more coarse-grained control over dangerous permissions. It is also worth noticing that once a group of dangerous permissions is granted, it is never revoked by the system; however, the user can remove a permission from an app by explicitly modifying the app settings.

An app obtains a system functionality by invoking a specific AAPI. The set of AAPI invocations can be mostly inferred from the app bytecode (i.e., the executable app code) through static analysis techniques. This inference is possible even if the code is obfuscated. Code obfuscation refers to a set of techniques applied at compile time by the developer and used to deter the reverse engineering of the app (i.e., in this case, the process of disassembling the app and analyzing its components by a human being). An AAPI invocation may require specific dangerous permissions. Thus, an AAPI is executed properly if the app has been granted the corresponding dangerous permissions. An official mapping between AAPI methods and permissions is not published. However, some work has empirically inferred this mapping [11]. An example of the correspondence between AAPIs and permissions is the following: the class `android.net.wifi.WifiManager` provides the AAPI method `isWifiEnabled()` that returns true or false whether the Wi-Fi interface is enabled or disabled and needs the permission `android.permission.ACCESS_WIFI_STATE` to check the state of the Wi-Fi. Since a mapping (albeit potentially incomplete) exists, checking both the requested permissions and the AAPI invocations could appear redundant, as considering the AAPI invocation inferred though static analysis could suffice. Unfortunately, the inference of the AAPI set by static analysis alone may be incomplete, in fact, an app could also leverage advanced Java mechanisms such as Reflection [12] and Java Native Interface (JNI) [13] to execute code that is not directly identifiable in the bytecode (i.e., it is impossible to infer all the specific invocations through static analysis alone). Yet, an AAPI invocation, especially a dangerous one, can still require some permissions to execute properly when invoked at runtime. Hence, we argue that taking into account both permissions and AAPI invocations as features for binary classifying malware allows detecting more malicious behaviors than each of the two by itself.

Several examples of data-driven Android malware detection systems exist in literature. One of the earliest work is KIRIN [21]. It defines a set of security rules describing potentially dangerous permission patterns. For instance, an app requiring both the `RECEIVE_SMS` and the `SEND_SMS` permissions is considered risky by KIRIN. The approach has been assessed on a very limited set of apps (i.e., 311), and allowed to recognize 10 apps violating all inferred security

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2017.2774184, IEEE Transactions on Sustainable Computing

Table I: Related work comparison

Paper	DroidMat[14]	PBMD[15]	PUMA[16]	TLPD [17]	Drebin[18]	MDLS[19]	PIndroid[20]	**BAdDroIds**
Year	2012	2013	2013	2014	2014	2016	2017	2017
Sustainability	L	L	H	M	M	L	H	H
Static features								
Req. permissions	✓	✓	✓	✓	✓	✓	✓	✓
Used permissions		✓		✓	✓			
App components	✓				✓			
Intents	✓				✓		✓	
API calls*	✓				✓	✓		✓
Inter-Comp Comm.	✓							
Market description						✓		
String pattern					✓			
Dynamic analysis		✓						
N°of apps	1738	21684	606	30084	129013	78649	1745	14988
Legal	1500	20548	357	28548	123453	52251	1300	7494
Malware	238	1136	249	1536	5560	26398*	445	7494
Accuracy	97.87	98	78	98.6	94	94	99.8	98.9

rules. Among them, 5 has been proved to be real malware through manual code inspection.

Table I provides a comparison among some of the most influential works that use ML techniques for malware detection, ordered by year of publication. We defined three labels, namely L(ow) M(edium) H(igh), describing the sustainability of the approach. The labeling is based on the resource footprint of the described tool, taking into account how much computational power is needed for collecting the features and apply the model to them. We always assume that the model is generated only once, during the training phase, not on the resource constrained device, hence the cost of the model generation is not used to evaluate the sustainability of the approach.

DroidMat[14], PBMD[15] and MDLS[19] are considered Low-sustainable for the following reasons: DroidMat requires to create the Inter-procedural control flow graph of the app, PBMD uses dynamic analysis, and MDLS downloads and parses the market description. In this respect, we choose to avoid considering the whole invocation chain our work as this would require to statically build a data structure (e.g., a flow graph) that is expensive in terms of memory and computational power.

TLPD [17] and Drebin[18] are considered Medium-sustainable because for generating the *Used permissions* set they must check if every method invocation in the code requires some permissions.

PUMA[16], PIndroid[20] and BAdDroIds are considered High-sustainable because they collect the features with a linear analysis of the bytecode and the Android Manifest file.

Previous works focus on the extraction of the following static features: requested permissions, used permissions, app components, intents, inter-component communication (ICC), meta data extracted from online market description, string patterns (e.g. URLs, IP addresses, base64, etc.) and API calls. For the sake of clarification, ours is the only approach that considers *every* API, i.e. every method invocation (that cannot be obfuscated) given by the language and the Android framework. For example we also consider the `java.lang.String` constructor. Usually authors check different subsets of API, often related to privacy or permissions declared in the manifest, but this selection lacks of important feature like Reflection and the loading of native code.

We chose a small subset of the most significant features among the ones that were already being researched extensively in previous publications and we demonstrated that they are enough for a very good classification. Obviously, other alternatives can be taken into consideration, but they would require too much memory or computation in order to be efficiently usable on a mobile device.

None of the cited articles use our set of features but our approach has higher accuracy with respect to any other paper in literature except PIndroid[20]. However, the high level of accuracy granted by PIndroid is calculated on a very reduced dataset of malware samples. Furthermore, no other work in literature also provides a freely available app for testing and the whole set of data allowing to replicate and check the presented statements. Finally, it is worth noting that our testbed is the second biggest malware dataset (7494) w.r.t. other works. Indeed, only in the MDLS experience presented in [19] the authors tested the solution on a dataset of 26398 malware samples.

III. The Data Driven Classification Model

For our specific malware classification purposes, we consider the supervised learning framework, with particular reference to the binary classification problem, where an input space $\mathcal{X}$ and an output space $\mathcal{Y}$ are available [22]. In our case $\mathcal{X} = \{0,1\}^d$, where each element of the space represents the presence or the absence of a particular declared permission or AAPI invocation (as AAPI invocations are retrieved from code through static analysis we will call them retrieved AAPI), and $\mathcal{Y} = \{\pm 1\}$, since the possible labels are legal $(+1)$ or malware (-1). Note that, the same problem can be faced as a novelty detection task [23]. In fact, in real world situations, the number of malware applications is much lower with respect to the number of legal ones. We made a preliminary study on the available data by exploiting the most recent tools in the novelty detection context [24], but results were not satisfying both in terms of accuracy and also in terms of resource requirements because of the need for the use of the kernel and a huge number of legal apps. In the supervised learning framework, the goal is to estimate the unknown rule $\mu : \mathcal{X} \to \mathcal{Y}$ which

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2017.2774184, IEEE Transactions on Sustainable Computing

associates a label $Y \in \mathcal{Y}$ to an element $X \in \mathcal{X}$. In general, μ can be non-deterministic [22] (i.e.,different apps may have the same sets of declared permissions and/or retrieved AAPI invocation but different label). A data driven technique estimates μ through a learning algorithm $\mathscr{A}_{\mathcal{H}} : \mathcal{D}_n \times \mathcal{F} \rightarrow f$, characterized by its set of hyperparameters $\mathcal{H}$, which maps a series of examples of the input/output relation, contained in a dataset of n samples $\mathcal{D}_n : \{(X_1, Y_1), \cdots, (X_n, Y_n)\}$ sampled from μ (or in other words n different labelled Android apps), into a function $f : \mathcal{X} \rightarrow \mathcal{Y}$. The error that f commits, in approximating μ, is measured with reference to a loss function $\ell : \mathcal{X} \times \mathcal{Y} \times \mathcal{F} \rightarrow [0, \infty)$. In our case, we will make use of the Hard loss function which counts the number of errors $\ell_{\mathrm{H}}(f(X), Y) = [f(X) \neq Y] \in \{0, 1\}$ [22]. The purpose of any learning procedure is to select the best set of hyperparameters $\mathcal{H}$ such that the expected error $L(f) = \mathbb{E}_{\mu}\ell(f(X), Y)$ – which unfortunately is unknown since μ is unknown – is minimum. Obviously, the error that f commits over $\mathcal{D}_n$ is optimistically biased since $\mathcal{D}_n$ has been used for building f itself. For this reason another set of fresh data, composed of m samples and called test set $\mathcal{T}_m = \{(X_1^t, y_1^t), \cdots, (X_m^t, y_m^t)\}$, needs to be exploited. Note that, $X_i^t \in \mathcal{X}$ and $Y_i^t \in \mathcal{Y}$ with $i \in \{1, \cdots, m\}$, and the association of Y_i^t to X_i^t is again made based on μ.

Many different algorithms exist in literature such as the Kernel-based method [23], the Neural Network-based one [25], [26], [27], the Ensemble Methods [28], the Bayesian approaches [29], the Local Methods [30], among others. In our case, we need to keep in mind that the classification model $f = \mathscr{A}_{\mathcal{H}}(\mathcal{D}_n)$ needs to run on a mobile device. For this reason we have to exploit a model which requires as little computational effort as possible [31]. In particular, the computational requirements of the training phase, namely the time needed to build f, are not important since the training phase can be performed offline. What is instead crucial are the computational requirements needed in order to compute $f(X)$ since it must be done on the device. In this context Kernel-based method are usually the best suited choice [32], [31]. Other alternatives exist such as Extreme Learning Machines [33], Deep Neural Networks [34], Random Forests [35], or Gaussian Processes [36] but the forward phase of these methods usually requires much more memory and computations with respect to our proposal [31].

We use two learning algorithms, one linear and one nonlinear by carefully considering the computational requirements of computing $f(X)$. Moreover, we will try to get insight on the problem of detecting a malware based on declared permissions and retrieved AAPI invocations by detecting the most important subset of them and their weight (i.e., if the presence of an invocation or a declared permissions is an indication of malware or not). Finally, we will show how to tune the hyperparameters of the different algorithms and how not to simply get a binary answer from $f(X)$ (legal or malware) but also a reliability estimation of such response.

For what concerns the linear approach, let us define $\mathcal{F}$ as the set of all the possible linear separators in the space $\mathcal{X}$: $f(X) = W^T X + b$ with $W \in \mathbb{R}^d$ and $b \in \mathbb{R}$ [22]. Based on this choice the most intuitive way of choosing W and b is to choose the solution which minimizes the error over the available data [22]:

$$(W, b) : \arg\min_{W,b} \sum_{i=1}^{n} \ell_{\mathrm{H}}(W^T X_i + b, Y_i). \tag{1}$$

Unfortunately, the Problem (1) has two drawbacks: (i) it is NP-Hard since the loss function is non-convex [31] and (ii) it is ill-posed and may overfit the available data and have large expected error [37]. In order to solve issue (i) it is necessary to approximate ℓ_{H} with one of its convex relaxations. The most suited one is the Hinge loss function $\ell_{\xi}(f(X_i), Y_i) = \max[0, 1 - Y_i f(X_i)]$ [22], the simplest convex upper bound of ℓ_{H}, which is also the best choice in this context [38]. By solving issue (i) we can also address the issue (ii) since, by exploiting ℓ_{ξ}, it is possible to introduce a regularization term, inspired by the Tikhonov regularization principle [39], which allows to derive a well posed alternative to Problem (1). Several regularization terms exists, form the L1 to the L2 and Lp norms [40], [41], [42], but in this work we will exploit the combination of the L1 and the L2 regularization schemes [42]. This choice is made since, in our case, $d \gg n$ and the presence of many declared permissions and retrieved AAPI invocations are correlated with each other. L1L2 regularization schema, also called elastic net regularization [42], is both a regularization and variable selection method. L1L2 often outperforms the L1, while providing a similar sparsity of representation. In addition, the L1L2 encourages a grouping effect, where strongly correlated features tend to be in or out of the model together. L1L2 is particularly useful when $d \gg n$ (as in our case) and, contrarily to L1, it is a very satisfactory variable selection method when $d \gg n$. Consequently Problem (1) can be reformulated as follows:

$$\begin{aligned}(W, b) : \arg\min_{W,b}\ & \lambda \|W\|_2^2 + (1 - \lambda)\|W\|_1 \\ & + \frac{C}{n} \sum_{i=1}^{n} \max[0, 1 - (Y_i W^T X_i + b)],\end{aligned} \tag{2}$$

which is a convex problem than can be solved with many tools developed in recent years [42], [43]. Note that $\lambda \in (0, 1)$ is a constant that balances sparsity characteristics with feature selection ability [42] while $C \in (0, \infty)$ is another constant which balances the tradeoff between underfitting and overfitting tendency [39]. The sparsity effect of the L1 regularizers also allows to reduce the number of $W_{j \in \{1, \cdots, d\}} \neq 0$ and then to obtain a model f which can run with reduced computational requirements [31].

The shape of the model f, built by solving Problem (2), together with the sparsity effect of the L1 regularizers and the fact that $\mathcal{X} = \{0, 1\}^d$, allows us to derive a simple yet effective and efficient feature selection and ranking method which also has the ability to infer if a feature is an indicator of malware or legal [44]. In fact, if some W_j with $j \in \{1, \cdots, d\}$ are equal to zero the meaning is straightforward: that feature j-th is not meaningful for distinguishing between legal or malware. If, instead, a particular W_j with $j \in \{1, \cdots, d\}$ is different from zero, since $X_j \in \{0, 1\}$, and $W_j > 0$ the feature j-th is an indication that the app is a malware. Analogously, if $W_j < 0$

the feature j-th is an indication that the app is legal. Finally, the magnitude of W_j gives its raw importance.

The limitation of Problem (2) is the shape of f which is linear [22]. In order to overcome this limitation, we can define f as a nonlinear function $f(X) = W^T\Phi(X) + b$ where $\Phi : \mathbb{R}^d \rightarrow \mathbb{R}^D$ with $D \gg d$ (since with d features we were not able to find a good classifier), $W \in \mathbb{R}^D$, and $b \in \mathbb{R}$. Then, we can substitute the new f in Problem (2) and exploit the representer theorem [45] in order to observe that the solution of Problem (2) can be expressed as $W = \sum_{i=1}^{n} \alpha_i \Phi(X_i)$ with $\alpha_i \in \mathbb{R}$ where $i \in \{1, \cdots, n\}$. By substituting these results in Problem (2) we obtain the following problem:

$$(W, b, \alpha) : \arg\min_{W,b,\alpha} \lambda\|W\|_2^2 + (1-\lambda)\|W\|_1 + \frac{C}{n}\sum_{i=1}^{n}\max[0, 1 - (Y_i W^T \Phi(X_i) + b)] \quad s.t.\ W = \sum_{i=1}^{n}\alpha_i\Phi(X_i). \tag{3}$$

Note that, Problem (3) suffers from the curse of dimensionality since the size of the problem depends on D. If D is large it may become intractable. For this reason, if we exploit the kernel trick [46], and, instead of applying the regularization over W we equivalently apply the regularization to α, we obtain the following problem:

$$(\alpha, b) : \arg\min_{\alpha,b} \lambda\|\alpha\|_2^2 + (1-\lambda)\|\alpha\|_1 + \frac{C}{n}\sum_{i=1}^{n}\max\left[0, 1 - \left(Y_i \sum_{j=1}^{n}\alpha_j K(X_i, X_j) + b\right)\right], \tag{4}$$

where $K(X_i, X_j) = \Phi(X_i)^T\Phi(X_j)$, Φ can remain unknown, $f(X) = \sum_{i=1}^{n}\alpha_j K(X_i, X) + b$, and the problem is still convex. We opt for a Gaussian Kernel $K(X_i, X_j) = e^{-\|X_i - X_j\|_2^2/\sigma}$ for the reason described in [47]. Obviously, the feature selection and ranking phase in this case is not possible but Problem (4) still takes into account the computational requirements of a mobile device. In fact, the L1 regularization allows to reduce the number of α_i with $i \in \{1, \cdots, n\}$ different from zero. The smaller the number of αs different from zero is, the less computational expensive is the computation of $f(X)$.

Another issue that we have to face is how to tune the hyperparameters of the proposed algorithms (λ, C, and σ for the nonlinear case) [48]. The values of the hyperparameters deeply affect the performance of the final classification model $\mathscr{A}_{\mathcal{H}}(\mathcal{D}_n)$ and for this reason they must be tuned carefully. Resampling techniques like cross validation [49] and non-parametric bootstrap [50] (BOO) are often used by practitioners because they work well in many situations [48]. Other alternatives exist, which are bases in the Statistical Learning Theory, which give more insight into the learning process. Examples of methods in this last category are: the seminal work of the Vapnik-Chervonenkis Dimension [22], its improvement with the Rademacher Complexity [51], [52], the theory of compression [53], [54], the Algorithmic Stability breakthrough [55], the PAC-Bayes theory [56], [57], and more recently the Differential Privacy theory [58], [59].

In our specific case the BOO will be exploited since it is the most effective one in cases like the one described in the paper, where the cardinality of the sample is reasonable [48]. BOO relies on a simple idea: the original dataset $\mathcal{D}_n$ is resampled many (n_o) times with replacement, to build two independent datasets called training and validation sets, respectively $\mathcal{L}_l^o$ and $\mathcal{V}_v^o$, with $o \in \{1, \cdots, n_o\}$. Note that $\mathcal{L}_l^o \cap \mathcal{V}_v^o = \varnothing$. Then, in order to select the best set of hyperparameters $\mathcal{H}$ in the set of possible ones $\mathfrak{H} = \{\mathcal{H}_1, \mathcal{H}_2, \cdots\}$ for the algorithm $\mathscr{A}_{\mathcal{H}}$ or, in other words, to perform the model selection phase, the following procedure needs to be applied:

$$\mathcal{H}^* : \arg\min_{\mathcal{H}\in\mathfrak{H}} \frac{1}{n_o}\sum_{o=1}^{n_o}\hat{L}^{\mathcal{V}_v^o}(\mathscr{A}_{\mathcal{H}}(\mathcal{L}_l^o)), \tag{5}$$

where $\hat{L}^{\mathcal{S}}(f) = {}^1/_{|\mathcal{S}|}\sum_{(X,Y)\in\mathcal{S}}\ell_H(f(X), Y)$. Since the data in $\mathcal{L}_l^o$ are different with respect to the ones in $\mathcal{V}_v^o$, the idea is that $\mathcal{H}^*$ should be the set of hyperparameters which allows to achieve a small error on a data set that is independent from the training set. Note that, in BOO, $l = n$ and $\mathcal{L}_l^o$ must be sampled with replacement from $\mathcal{D}_n$, while $\mathcal{V}_v^o = \mathcal{D}_n \setminus \mathcal{L}_l^o$.

Finally, it is worth underlining that the classifier that we have just proposed gives, as an output, only the answer legal or malware. In a real world scenario this information is not enough. What it is important is also the reliability of the models' answers. For this reason we exploit the proposal of [60] which is able to take the $f(X)$ (in our case $f(X) = W^T X + b$ for the linear model and $f(X) = \sum_{(X_i,Y_i)\in\mathcal{D}_n}\alpha_j K(X_i, X) + b$ for the nonlinear one) and associates a probability to the choice of the model. In particular:

$$\mathbb{P}\{f(X) = +1\} = \frac{1}{1 + e^{\gamma f(X) + \beta}}, \tag{6}$$

where $\gamma \in \mathbb{R}$ and $\beta \in \mathbb{R}$ are chosen by minimizing the negative log likelihood averaged over the different $\mathcal{V}_v^o$ which is a cross-entropy error function:

$$(\gamma, \beta) : \arg\min_{\gamma,\beta} -\sum_{o=1}^{n_o}\sum_{(X,Y)\in\mathcal{V}_v^o}\left[\left(\frac{Y+1}{2}\right)\log\left(\frac{1}{1+e^{\gamma f(X)+\beta}}\right) + \left(1 - \frac{Y+1}{2}\right)\log\left(1 - \frac{1}{1+e^{\gamma f(X)+\beta}}\right)\right]. \tag{7}$$

IV. Empirical Evaluation

We empirically evaluated the proposed data driven model on a set of 14988 APKs, half of which (i.e., 7494) are malware samples, while the remaining 7494 are legal apps downloaded from the Google Play Store [61]. We define the APKs in the latter set as *legal* as all of them have been previously analyzed through Virus Total [62] without being recognized as malware by any of its 59 different antivirus engines; therefore, it is reasonable to assume that they are not malware. The malware APKs have been downloaded from the AndroZoo dataset that contains officially recognized malware. Each entry in such dataset contains information about the source app market and the Virus Total scan result. We took into consideration any APK that has been recognized as malware by at least 30 engines in Virus Total. As previously pointed out, we considered

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2017.2774184, IEEE Transactions on Sustainable Computing

as features the required permissions, the AAPI invoked in the app, and a combination of both. Regarding AAPI, we were not interested in building the chain of invocations in the app that leads to invoke the specific AAPI as discussed in Section II. On the contrary, we were only interested in determining whether a specific AAPI is invoked somewhere in the app code, independently from how or when it is really invoked. Thus, we parsed the *AndroidManifest* file (i.e., the file that contains, among others, all the permissions required by the app) to extract the required permissions, and the DEX files to retrieve the AAPI invocations, thereby building, for each app A, the set P_A of required permissions, and the set I_A, of AAPI invocations.

It is worth pointing out that app developers can define their own custom permissions. Other apps declaring a custom permission can access to the specific functionality provided by the app defining it. This often happens for apps developed by the same developer. Even if there is a known attack [63] [64] that exploits a vulnerability in custom permission, we disregard them from our analysis because the exploited vulnerability has been fixed since Android 5, allowing only apps signed with the same signing key to define the same <permission> element, and the malware in our dataset very rarely use or define custom permissions. For these reasons, we considered only the official Android permissions [65], thereby discarding all custom permissions defined by apps.

Therefore, given ϕ_{And} as the set of all Android permissions, and an app A, we have that for each $p \in P_A$ then $p \in \phi_{And}$ and $P_A \subseteq \phi_{And}$. Regarding the extraction of AAPI invocations, we adopted *dexlib2* [66], a library that sequentially analyzes the bytecode and build up an abstract representation of the code. From this representation, we extracted all the AAPI invocations. In this respect, it is worth mentioning that we ignore method *overloading* for AAPI methods. In a nutshell, method overloading in Java is the possibility to have different methods in a class having the same name, as long as their arguments list is different. In our extraction, we consider overloaded AAPI invocations as semantically equivalent. This means that we consider $i \in I$ as the concatenation of the class and the method name even if there are multiple entries with the same method name in the class.

A. Experimental Results

In this section, we discuss the results achieved by applying the techniques proposed in Section III to the problem described in Section II, based on the data described in Section IV.

In particular two approaches have been compared:

- LIN: the linear learning algorithm proposed in Problem (2);
- KER: the non linear learning algorithm proposed in Problem (4);

For what concerns LIN, we set $\mathcal{H} = \{\lambda, C\}$ and $\mathfrak{H} = \{10^{-4.0}, 10^{-3.8}, \cdots, 10^{0}\} \times \{10^{-4.0}, 10^{-3.8}, \cdots, 10^{3.0}\}$ while for KER we set $\mathcal{H} = \{\lambda, C, \gamma\}$ and $\mathfrak{H} = \{10^{-4.0}, 10^{-3.5}, \cdots, 10^{0}\} \times \{10^{-4.0}, 10^{-3.5}, \cdots, 10^{3.0}\} \times \{10^{-4.0}, 10^{-3.5}, \cdots, 10^{3.0}\}$ and $n_o = 10^3$. For what concerns (γ, β), the best solution is searched on the following grid $\{\pm 10^{-6.0}, \pm 10^{-5.9}, \cdots, 10^{6}\} \times \{\pm 10^{-6.0}, \pm 10^{-5.9}, \cdots, 10^{6}\}$.

Moreover, the three scenarios discussed in Sec. IV have been investigated, namely:

- PER: where a classifier is built just based on the features related to the required permissions (i.e., $\{P_A\}$);
- INV: where a classifier is built just based on the features related to the AAPI invocations (i.e., $\{I_A\}$);
- PERINV: where a classifier is built based both on the features related to the declared permissions and the ones related to the AAPI invocations.

We split the $s = 14988$ samples in $\mathcal{D}_n$ and $\mathcal{T}_m$ such that $n + m = s$ and $\mathcal{D}_n \cap \mathcal{T}_m = \emptyset$ and $n \in \{750, 1500, 3000, 6000, 12000\}$. Experiments have been repeated 30 times in order to obtain statistically relevant results.

In Table II we reported the $\widehat{L}^{\mathcal{T}_m}(\mathscr{A}_{\mathcal{H}^*}(\mathcal{D}_n))$ of LIN and KER for problem PER, INV and PERINV when varying n. Based on the results reported in Table II, it is possible derive some observation.

The first one is that the larger is the training set (the more app we use for training the model) the more effective the resulting model is. Moreover, in general, the KER is more powerful than LIN. As expected, the more information we provide to the learning algorithm the more effective the resulting model will be. In particular, the AAPI invocations have more predictive power with respect to the permissions and together they have even more predictive performance. Surprisingly, the difference of the two best performing models, LIN and KER with PERINV, is not statistically relevant (the two distributions of the errors cannot be distinguished with a t-test). Therefore, we chose the LIN model that is more suitable to be deployed on a smartphone device as it requires less computational resources in comparison to KER.

In Table III we reported the confusion matrices (in $\%$) of the best performing models, namely LIN and KER for PERINV when $n = 12000$. From Table III it is possible to observe that the false positive and false negative rate is quite balanced in both models, thereby indicating that the models have a high quality. Furthermore, for the sake of completeness we also provided in Table IV the scores associated to some of the most common performance indexes.

In Table V the confusion matrices (in $\%$) of LIN and KER for PERINV and $n = 12000$ are reported. These matrices take into account also a *warning* class that represents the case when an app is classified as malware with a probability between 30% and 70%. In this case, the decision is left to the user; such alternative allows to remarkably reduce the number of false positives and false negatives at the expenses of letting the user decide in critical cases.

In Table VI the *Top* 20 permissions and AAPI invocations, together with their raw importance (see Section III), of LIN are reported. We consider only LIN as it is the only one that can provide such information, for PER, INV, and PERINV with $n = 12000$. From Table VI it is possible to observe that a small amount of permissions and AAPI invocations have high importance for predicting the presence of a malware (strongly positive raw importance). Contrarily, a large amount of them have small importance for predicting the absence of a malware (weakly negative raw importance). This is reasonable since some permissions and AAPI invocations are

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2017.2774184, IEEE Transactions on Sustainable Computing

a sort of strong indicator for a malware while the presence of many other permissions and AAPI invocations show that the app is performing common legal tasks. This underlines that innocuous permissions and AAPI invocations have high importance for predicting the absence of a malware with data-driven techniques while conventional approaches just search for malware behaviors. In general, the analysis of results suggests that the main goal of malware is to collect as much information as possible about the user and the phone, as well as getting access to the SMS service (i.e., to force the user to subscribe to some payment services).

V. Concluding Remarks

Sustainability in the field of computing must not interfere with security, hence, it is important that security systems and related measures are designed from the very start to be sustainable and compatible with the resource constraints of the target platform. This is important in the perspective of greening computing and networking, but is paramount in the world of mobile devices and IoT, where resources in general and energy in particular represent very hard constraints. In this paper we have presented a machine learning-based technique that focuses on the identification of malware in resource constrained devices such as Android smartphones. Our technique has a very low resource footprint and does not rely on resources outside the protected device. The technique is at the basis of BAdDroIds, an Android app focused on early identification of a malware, more in details, directly at installation time, without heavily impacting the usability and the battery life of the mobile device. We adopted a data-driven approach capable of achieving a high level of accuracy in malware identification on the basis of a set of features easily inferable from apps through static analysis techniques. To validate our methodology we have implemented BAdDroIds (see Fig. 1), which has been released on the Google Play Store[1], and we have tested it on almost fifteen thousand different apps half of which were malware (Fig. 2 shows an example of malware and non-malware analysis results). BAdDroIds showed an accuracy level equal to 98.9%, more

[1] https://play.google.com/store/apps/details?id=it.unige.dibris.baddroids

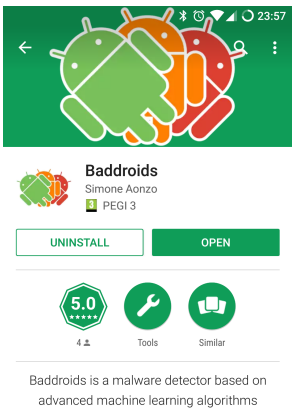

Figure 1: BAdDroIds on the Google Play Store

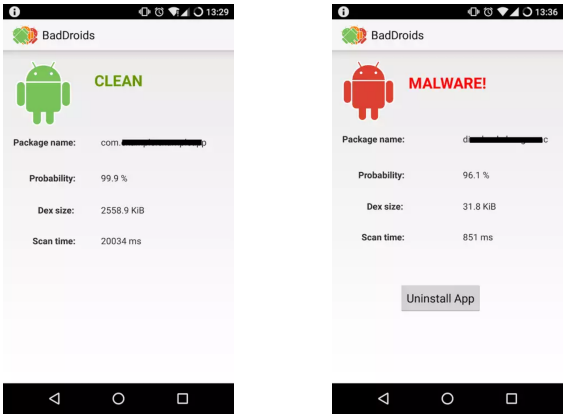

Figure 2: BAdDroIds: malware and non-malware classification

in details 0.6% false positives and 0.5% false negatives. The complete dataset as well as further information on BAdDroIds are available at `http://baddroids.smartlab.ws`. The dataset has been also submitted to UCI [67].

Furthermore, to ensure that the usage of the tool on a mobile device was neither disruptive to the user experience, nor incompatible with less powerful devices, we tested it on a dated device, namely an LG Nexus 5. This device was released in 2013, runs Android 6.0.1, has a Qualcomm MSM8974 Snapdragon 800 CPU running at 2.3GHz, and 2GB of RAM.

We randomly chose 1000 APKs from our dataset and, since BAdDroIds starts whenever an app is installed or updated, we have installed them on our device logging the size of the DEX file and the time needed for the analysis. We obtained that the average DEX file size is 5539 KiB and the average time for analyzing an APK is 64474 ms.

It is worth noting that the specifications of our test device can be considered comparable with a mid-range mobile device of the current generation, thus BAdDroIds does not need a top-notch device to be actually used and on such a configuration and it requires a minute, on average, to analyze an app.

While the results in terms of accuracy are remarkable, the time required to perform the complete analysis is still clearly perceivable by the user, hence, in future work, we need to optimize the process to a further extent. Moreover, while the feature set adopted in this work has shown very good properties in terms of accuracy of the prediction, we need to verify its resilience to obsolescence and we also need to explore the possibility to adopt more sophisticated properties of the apps as independent features, like the interaction with other apps through intents, as well as the usage of Reflection and JNI.

References

[1] Q. Li and M. Zhou, "The survey and future evolution of green computing," in *Proceedings of the 2011 IEEE/ACM International Conference on Green Computing and Communications*, ser. GREENCOM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 230–233. [Online]. Available: http://dx.doi.org/10.1109/GreenCom.2011.47

[2] A. P. Bianzino, C. Chaudet, D. Rossi, and J. L. Rougier, "A survey of green networking research," *IEEE Communications Surveys Tutorials*, vol. 14, no. 1, pp. 3–20, First 2012.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2017.2774184, IEEE Transactions on Sustainable Computing

Table II: $\widehat{L}^{\mathcal{T}_m}(\mathscr{A}_{\mathcal{H}^*}(\mathcal{D}_n))$ of LIN and KER for problem PER, INV and PERINV when varying n.

n	PER		INV		PERINV	
	LIN	KER	LIN	KER	LIN	KER
750	12.6 ± 0.9	11.9 ± 0.8	5.3 ± 0.3	5.4 ± 0.3	5.1 ± 0.2	5.2 ± 0.2
1500	12.4 ± 0.8	10.5 ± 0.8	4.0 ± 0.3	4.1 ± 0.3	4.0 ± 0.2	4.0 ± 0.2
3000	12.3 ± 0.8	10.9 ± 0.8	3.4 ± 0.3	3.4 ± 0.3	2.9 ± 0.2	3.0 ± 0.2
6000	12.0 ± 0.8	10.2 ± 0.7	3.2 ± 0.2	2.9 ± 0.2	2.2 ± 0.1	1.7 ± 0.2
12000	11.7 ± 0.7	9.2 ± 0.6	2.5 ± 0.1	2.2 ± 0.2	1.1 ± 0.1	1.0 ± 0.2

Table III: Confusion matrices (in $\%$) of LIN and KER for PERINV when n = 12000.

		Truth	
	LIN	Legal	Malware
Prediction	Legal	49.4 ± 0.1	0.5 ± 0.1
	Malware	0.6 ± 0.1	49.5 ± 0.1

		Truth	
	KER	Legal	Malware
Prediction	Legal	49.5 ± 0.2	0.5 ± 0.1
	Malware	0.5 ± 0.1	49.5 ± 0.2

Table IV: Performance indexes values (in $\%$) of LIN and KER for PERINV when n = 12000.

Performance Index	LIN	KER
sensitivity or true positive rate	0.988 ± 0.001	0.990 ± 0.001
specificity or true negative rate	0.990 ± 0.001	0.990 ± 0.001
precision or positive predictive value	0.990 ± 0.001	0.990 ± 0.001
negative predictive value	0.988 ± 0.001	0.988 ± 0.001
false negative rate	0.012 ± 0.001	0.012 ± 0.001
fall-out or false positive rate	0.010 ± 0.001	0.010 ± 0.001
false discovery rate	0.010 ± 0.001	0.010 ± 0.001
false omission rate	0.012 ± 0.001	0.012 ± 0.001
accuracy	0.989 ± 0.001	0.990 ± 0.001
F1 score	0.989 ± 0.001	0.990 ± 0.001
Matthews correlation coefficient	0.978 ± 0.001	0.978 ± 0.001
informedness	0.978 ± 0.001	0.980 ± 0.001
markedness	0.978 ± 0.001	0.978 ± 0.001

Table V: Confusion matrices (in $\%$) of LIN and KER for PERINV when n = 12000 when the Warning class is introduced (apps classified with probability of being a Malware greater than 30% and less then 70%).

		Truth	
	LIN	Legal	Malware
Prediction	Legal	49.0 ± 0.1	0.2 ± 0.1
	Warning	0.3 ± 0.1	0.3 ± 0.1
	Malware	0.3 ± 0.1	49.1 ± 0.1

		Truth	
	KER	Legal	Malware
Prediction	Legal	49.1 ± 0.1	0.3 ± 0.1
	Warning	0.2 ± 0.1	0.2 ± 0.1
	Malware	0.3 ± 0.1	49.1 ± 0.1

[3] A. Merlo, M. Migliardi, and L. Caviglione, "A survey on energy-aware security mechanisms," *Pervasive and Mobile Computing*, vol. 24, pp. 77 – 90, 2015, special Issue on Secure Ubiquitous Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574119215000929

[4] M. Migliardi and A. Merlo, "Improving energy efficiency in distributed intrusion detection systems," *J. High Speed Netw.*, vol. 19, no. 3, pp. 251–264, Jul. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2595805.2595811

[5] L. Caviglione and A. Merlo, "The energy impact of security mechanisms in modern mobile devices," *Network Security*, vol. 2012, no. 2, pp. 11 – 14, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1353485812700156

[6] F. Palmieri, S. Ricciardi, and U. Fiore, "Evaluating network-based dos attacks under the energy consumption perspective: new security issues in the coming green ict area," in *Broadband and Wireless Computing, Communication and Applications (BWCCA), 2011 International Conference on*. IEEE, 2011, pp. 374–379.

[7] U. Fiore, A. Castiglione, A. De Santis, and F. Palmieri, "Exploiting battery-drain vulnerabilities in mobile smart devices," *IEEE Transactions on Sustainable Computing*, vol. 2, no. 2, pp. 90–99, 2017.

[8] T. Verge, "The entire history of iphone vs. android summed up in two charts," http://www.theverge.com/2016/6/1/11836816/iphone-vs-android-history-charts, accessed October 19, 2017.

[9] A. Merlo, M. Migliardi, and P. Fontanelli, "Measuring and estimating power consumption in android to support energy-based intrusion detection," vol. 23, no. 5, pp. 611–637, 2015.

[10] "Dangerous permissions," https://developer.android.com/guide/topics/permissions/requesting.html#normal-dangerous, accessed October 19, 2017.

[11] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the Android permission specification," in *ACM conference on Computer and communications security*, 2012.

[12] "Android reflection," https://developer.android.com/reference/java/lang/reflect/package-summary.html, accessed October 19, 2017.

[13] "Java native interface on android," https://developer.android.com/training/articles/perf-jni.html, accessed October 19, 2017.

[14] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 2012, pp. 62–69.

[15] Z. Aung and W. Zaw, "Permission-based android malware detection," *International Journal of Scientific & Technology Research*, vol. 2, no. 3, pp. 228–234, 2013.

[16] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez, "PUMA: Permission Usage to detect Malware in Android," in *International Joint Conference CISIS'12-ICEUTE´ 12-SOCO´ 12 Special Sessions*, 2013.

[17] X. Liu and J. Liu, "A two-layered permission-based android malware detection scheme," in *Mobile cloud computing, services, and engineering (mobilecloud), 2014 2nd ieee international conference on*. IEEE, 2014, pp. 142–148.

[18] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck,

Table VI: Top 20 permission and retrieved API invocations of LIN for PER, INV, and PERINV with n = 12000.

PER	
Raw Importance	Permission
1.00	android.permission.SEND_SMS
0.89	android.permission.READ_PHONE_STATE
−0.80	android.permission.ACCESS_NETWORK_STATE
0.75	com.android.launcher.permission.UNINSTALL_SHORTCUT
0.73	android.permission.CHANGE_WIFI_STATE
0.61	android.permission.READ_SMS
0.59	android.permission.WRITE_APN_SETTINGS
0.58	android.permission.DELETE_PACKAGES
−0.53	android.permission.READ_CALL_LOG
−0.51	android.permission.MODIFY_AUDIO_SETTINGS
0.51	android.permission.ACCESS_LOCATION_EXTRA_COMMANDS
0.42	android.permission.WRITE_CALENDAR
−0.40	android.permission.READ_EXTERNAL_STORAGE
0.39	com.android.launcher.permission.INSTALL_SHORTCUT
0.38	android.permission.READ_LOGS
−0.38	android.permission.PACKAGE_USAGE_STATS
0.37	android.permission.RECEIVE_BOOT_COMPLETED
−0.37	android.permission.GET_ACCOUNTS
−0.35	android.permission.DISABLE_KEYGUARD
0.33	android.permission.STATUS_BAR
INV	
Raw Importance	Retrieved AAPI invocation
1.00	android.telephony.SmsManager->getDefault
0.56	android.content.BroadcastReceiver-><init>
0.51	android.app.admin.DeviceAdminReceiver-><init>
0.49	android.telephony.TelephonyManager->getDeviceId
0.45	android.telephony.TelephonyManager->getLine1Number
0.42	android.telephony.gsm.SmsManager->getDefault
0.42	java.lang.String-><init>
0.39	java.io.InputStreamReader-><init>
0.39	java.lang.reflect.Field->get
0.37	android.app.admin.DevicePolicyManager->isAdminActive
−0.36	android.content.Context->getPackageName
0.36	android.app.Application->attachBaseContext
0.36	android.app.ActivityManager->getRunningServices
0.35	android.app.PendingIntent->getBroadcast
−0.35	android.content.Intent-><init>
−0.35	java.lang.String->format
0.35	java.lang.String->valueOf
−0.33	android.content.Context->getSystemService
0.32	android.content.Context->getDir
0.32	android.os.Bundle->get
PERINV	
Raw Importance	Permission or Retrieved API invocation
1.00	android.permission.SEND_SMS
0.46	android.telephony.SmsManager->getDefault
0.44	android.content.BroadcastReceiver-><init>
0.42	android.app.Application->attachBaseContext
0.40	android.app.admin.DeviceAdminReceiver-><init>
−0.39	android.permission.ACCESS_NETWORK_STATE
0.37	android.telephony.TelephonyManager->getDeviceId
0.37	java.io.InputStreamReader-><init>
0.34	android.telephony.TelephonyManager->getLine1Number
−0.32	android.content.Context->getPackageName
0.30	java.lang.String-><init>
0.28	java.lang.String->valueOf
0.28	java.lang.reflect.Field->get
−0.28	java.lang.String->format
0.28	java.io.FileOutputStream->write
0.28	android.app.admin.DevicePolicyManager->isAdminActive
−0.28	android.content.Context->getSystemService
0.27	android.webkit.WebView->setDownloadListener
0.27	android.permission.RECEIVE_SMS
−0.27	java.util.Iterator->next

"DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *NDSS*, 2014.

[19] T. Ban, T. Takahashi, S. Guo, D. Inoue, and K. Nakao, "Integration of multi-modal features for android malware detection using linear svm," in *Information Security (AsiaJCIS), 2016 11th Asia Joint Conference on*. IEEE, 2016, pp. 141–146.

[20] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan, "Pindroid: A novel android malware detection system using ensemble learning methods," *Computers & Security*, vol. 68, pp. 36–46, 2017.

[21] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 235–245.

[22] V. N. Vapnik, *Statistical learning theory*. Wiley New York, 1998.

[23] J. Shawe-Taylor and N. Cristianini, *Kernel methods for pattern analysis*. Cambridge university press, 2004.

[24] L. Swersky, H. O. Marques, J. Sander, R. J. Campello, and A. Zimek, "On the evaluation of outlier detection and one-class classification methods," in *IEEE International Conference on Data Science and Advanced Analytics*, 2016.

[25] C. M. Bishop, *Neural networks for pattern recognition*. Oxford university press, 1995.

[26] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[27] G. B. Huang, D. H. Wang, and Y. Lan, "Extreme learning machines: a survey," *International Journal of Machine Learning and Cybernetics*, vol. 2, no. 2, pp. 107–122, 2011.

[28] C. Zhang and Y. Ma, *Ensemble machine learning*. Springer, 2012.

[29] E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[30] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.

[31] L. Oneto, S. Ridella, and D. Anguita, "Learning hardware-friendly classifiers through algorithmic stability," *ACM Transaction on Embedded Computing*, vol. 15, no. 2, pp. 23:1–23:29, 2016.

[32] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "Human activity recognition on smartphones using a multiclass hardware-friendly support vector machine," in *International Workshop on Ambient Assisted Living*, 2012.

[33] J. Tang, C. Deng, and G. B. Huang, "Extreme learning machine for multilayer perceptron," *IEEE transactions on neural networks and learning systems*, vol. 27, no. 4, pp. 809–821, 2016.

[34] J. S., "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[35] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[36] C. E. Rasmussen and C. K. I. Williams, *Gaussian processes for machine learning*. MIT press Cambridge, 2006.

[37] A. Bakushinskiy and A. Goncharsky, *Ill-posed problems: theory and applications*. Springer Science & Business Media, 2012.

[38] L. Rosasco, E. De Vito, A. Caponnetto, M. Piana, and A. Verri, "Are loss functions all the same?" *Neural Computation*, vol. 16, no. 5, pp. 1063–1076, 2004.

[39] A. N. Tikhonov and V. I. A. Arsenin, *Solutions of ill-posed problems*. Halsted Press, New York, 1977.

[40] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[41] F. Schöpfer, A. K. Louis, and T. Schuster, "Nonlinear iterative methods for linear ill-posed problems in banach spaces," *Inverse Problems*, vol. 22, no. 1, p. 311, 2006.

[42] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 67, no. 2, pp. 301–320, 2005.

[43] D. Anguita, A. Ghio, L. Oneto, J. L. Reyes-Ortiz, and S. Ridella, "A novel procedure for training l1-l2 support vector machine classifiers," in *International Conference on Artificial Neural Networks*, 2013.

[44] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of machine learning research*, vol. 3, pp. 1157–1182, 2003.

[45] B. Schölkopf, R. Herbrich, and A. J. Smola, "A generalized representer theorem," in *International Conference on Computational Learning Theory*, 2001.

[46] B. Scholkopf, "The kernel trick for distances," *Advances in neural information processing systems*, 2001.

[47] S. S. Keerthi and C. J. Lin, "Asymptotic behaviors of support vector machines with gaussian kernel," *Neural computation*, vol. 15, no. 7, pp. 1667–1689, 2003.

[48] D. Anguita, A. Ghio, L. Oneto, and S. Ridella, "In-sample and out-of-sample model selection and error estimation for support vector machines," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 9, pp. 1390–1406, 2012.

[49] S. Arlot and A. Celisse, "A survey of cross-validation procedures for model selection," *Statistics surveys*, vol. 4, pp. 40–79, 2010.

[50] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap*. CRC press, 1994.

[51] P. L. Bartlett and S. Mendelson, "Rademacher and gaussian complexities: Risk bounds and structural results," *Journal of Machine Learning Research*, vol. 3, pp. 463–482, 2002.

[52] P. L. Bartlett, O. Bousquet, and S. Mendelson, "Local rademacher complexities," *Annals of Statistics*, vol. 33, no. 4, pp. 1497–1537, 2005.

[53] S. Floyd and M. Warmuth, "Sample compression, learnability, and the vapnik-chervonenkis dimension," *Machine learning*, vol. 21, no. 3, pp. 269–304, 1995.

[54] J. Langford and D. McAllester, "Computable shell decomposition bounds," *Journal of Machine Learning Research*, vol. 5, pp. 529–547, 2004.

[55] O. Bousquet and A. Elisseeff, "Stability and generalization," *Journal of Machine Learning Research*, vol. 2, pp. 499–526, 2002.

[56] G. Lever, F. Laviolette, and F. Shawe-Taylor, "Tighter pac-bayes bounds through distribution-dependent priors," *Theoretical Computer Science*, vol. 473, pp. 4–28, 2013.

[57] P. Germain, A. Lacasse, M. Laviolette, A. ahd Marchand, and R. J. F., "Risk bounds for the majority vote: From a pac-bayesian analysis to a learning algorithm," *Journal of Machine Learning Research*, vol. 16, no. 4, pp. 787–860, 2015.
[58] C. Dwork, V. Feldman, M. Hardt, T. Pitassi, O. Reingold, and A. Roth, "Preserving statistical validity in adaptive data analysis," in *Annual ACM Symposium on Theory of Computing*, 2015.
[59] ——, "The reusable holdout: Preserving validity in adaptive data analysis," *Science*, vol. 349, no. 6248, pp. 636–638, 2015.
[60] J. Platt, "Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods," *Advances in large margin classifiers*, vol. 10, no. 3, pp. 61–74, 1999.
[61] "Google play store," https://play.google.com/, accessed October 19, 2017.
[62] "Virus total," https://www.virustotal.com/, accessed October 19, 2017.
[63] "The custom permission problem," https://github.com/commonsguy/cwac-security/blob/master/PERMS.md, accessed October 19, 2017.
[64] "Android custom permissions leak user data," [http://blog.trendmicro.com/trendlabs-security-intelligence/android-custom-permissions-leak-user-data/, accessed October 19, 2017.
[65] "Android permissions," https://developer.android.com/reference/android/Manifest.permission.html, accessed October 19, 2017.
[66] "Dexlib2," https://github.com/JesusFreke/smali/tree/master/dexlib2, accessed October 19, 2017.
[67] M. Lichman, "UCI machine learning repository," http://archive.ics.uci.edu/ml", accessed October 19, 2017.

Simone Aonzo Simone Aonzo is a PhD student in Computer Science and Systems Engineering at DIBRIS, University of Genoa. His research topics are: mobile security, binary/malware analysis and exploitation techniques. Before starting the PhD he worked as security software developer and pentester for an IT security company.

Alessio Merlo Alessio Merlo got a Ph.D. in Computer Science from University of Genova (Italy) in 2010 where he worked on performance and access control issues related to Grid Computing. He is currently serving as an Assistant Professor at the University of Genoa, Italy. His research interests focus on performance and security issues related to Web, distributed and mobile systems.

Mauro Migliardi Mauro Migliardi is currently Associate Professor at the University of Padua and Adjunct Professor at the University of Genoa. His main research interest is the engineering of secure, energy aware distributed systems. He tutored more than 80 among Bachelor, Master and PhD students at the Universities of Genoa, Padua and Emory, and he authored or co-authored more than 130 scientific papers.

Luca Oneto Luca Oneto was born in Rapallo, Italy in 1986. He received his BSc and MSc in Electronic Engineering at the University of Genoa, Italy respectively in 2008 and 2010. In 2014 he received his PhD from the same university in School of Sciences and Technologies for Knowledge and Information Retrieval with the thesis "Learning Based On Empirical Data". In 2017 he obtained the Italian National Scientific Qualification for the role of Associate Professor in Computer Engineering. He is currently an Assistant Professor at University of Genoa with particular interests in Statistical Learning Theory, Machine Learning, and Data Mining.

Francesco Palmieri is an associate professor at the Computer Science Department of the Salerno University. He received his M.S. Degree and Ph.D. in Computer Science from the Salerno University. His research interests concern Advanced Networking Protocols and Architectures and Network Security. He authored more that 150 scientific papers in reputed journals and conferences, serves as the Editor-in-Chief of an international journal and participates to the Editorial Board of other ones.