RiskInDroid: Machine Learning-based Risk Analysis on Android

Alessio Merlo[1], Gabriel Claudiu Georgiu[2]

[1] DIBRIS, University of Genoa, Italy.
Email: alessio@dibris.unige.it
[2] Talos Security, s.r.l.s., Savona, Italy.
Email: gabriel.georgiu@talos-sec.com

Abstract. Risk analysis on Android is aimed at providing metrics to users for evaluating the trustworthiness of the apps they are going to install. Most of current proposals calculate a risk value according to the permissions required by the app through probabilistic functions that often provide unreliable risk values. To overcome such limitations, this paper presents RiskInDroid, a tool for risk analysis of Android apps based on machine learning techniques. Extensive empirical assessments carried out on more than 112K apps and 6K malware samples indicate that RiskInDroid outperforms probabilistic methods in terms of precision and reliability.

Keywords: Risk Analysis, Android Security, Static Analysis, Machine Learning

1 Introduction

Android is still the most widespread mobile operating system in the world, as more than 300 millions Android-enabled smartphones have been sold only in the third trimester of 2016 [1]. Therefore, it remains a sensitive target for malware that aim at exploiting its diffusion to reach a high number of potential victims. Since users have access to a high number of apps through public markets and external web sites, they need reliable tools to rate the trustworthiness of apps they are going to install. App rating is empirically calculated according to different risk analysis techniques. Currently, most of them calculate a *risk index value* (hereafter, `RIV`) through probabilistic methods applied to the set of permissions required by the app. We argue that such approaches suffer from intrinsic limitations in terms of both methodology and setup. To prove this, we apply some optimizations to existing techniques at the state of the art, and we evaluate them through an extensive empirical assessment on a dataset made by 112.425 apps and 6.707 malware samples. Then, we propose a novel approach based on machine learning techniques that we implemented in an open source tool, i.e., `RiskInDroid`[3] (Risk Index for Android). Finally, we evaluate the performance of RiskInDroid on the same dataset, thereby proving that the proposed methodology outperforms probabilistic approaches.

[3] Freely available at: `http://www.csec.it/riskindroid`

Structure of the paper. The rest of the paper is organized as follows: Section 2 briefly introduces the Android architecture and the permission system, while Section 3 summarizes the related work and introduces probabilistic approaches. Section 4 discusses some optimization for probabilistic methods and proves their reliability through an extensive experimental assessment. Section 5 proposes our machine learning-based methodology while Section 6 summarizes its empirical evaluation. Finally, Section 7 concludes the paper and points out some future work.

2 Android in a Nutshell

Android is made by a layered architecture (see Fig. 1) where the top layer hosts both system and user apps. System apps come with the Android distribution itself and provide basic functionality (e.g., calendar, email, ...), while user apps are packed into compressed archives (i.e., the `APKs`) and made available to users on different external sources (e.g., app markets or web sites). Below the app layer lies the Application Framework that provides a set of modular components that apps can use to access system and device resources.

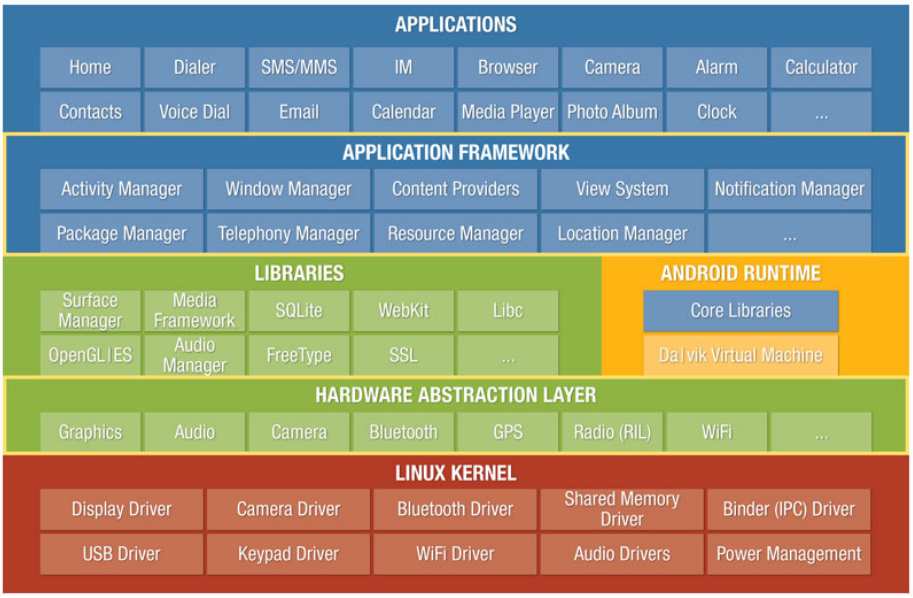


Fig. 1: The Android OS architecture.

Android also contains a set of C/C++ native libraries granting optimized core services (e.g., DBMS, 2D/3D graphics, Codecs, ...). The Android Runtime provides virtual machines to execute the apps bytecode. The Hardware Abstraction Layer (HAL) is a set of libraries allowing the Application Framework to access the actual hardware. The Linux Kernel is at the bottom of the architecture and grants basic OS functionality as Interprocess Communication (IPC), memory and process management.

Security and Permissions. Android assigns a unique Linux user ID at Kernel layer to each app upon installation, thereby *sandboxing* the execution of apps in separate Linux users. Android authorizes apps to access core system resources through *Android Permissions* (hereafter, `APs`) that are required by the app and granted by the user upon installation or at runtime[4]. APs are declared in an XML file, i.e., the *Android Manifest*, contained in the APK. To get services from core Android APIs, the app should have the corresponding AP. There currently exist more than 130 APs[5], divided into four categories, namely, 1) *Normal*, i.e., basic authorizations that are automatically provided by the system upon installation, 2) *Dangerous*, required for accessing core APIs, they are granted by the user, 3) *Signature*, granted to apps signed by the same developer, and 4) *SignatureOrSystem*, automatically granted to system apps. We refer to the whole set of Android permissions as `APSet`. Apps are expected to require the least set of permissions sufficient to work properly, albeit they are often overprivileged [2]. Apps can also be underprivileged, but in this case they are expected to fail during execution.

3 Related Work

The scientific literature related to risk analysis of Android apps is rather limited and mostly focused on APs, so we also take into account works regarding malware classification because we expect to see some relationships between malware and high risk apps. Currently available proposals are probabilistic, i.e., the RIV indicates the probability that an app can be a malware, according to statistical analysis carried out on datasets containing both apps (that are expected to be mostly benign) and well-known malware samples. In [3], authors propose a method for detecting *risk signals* according to the frequency of security-sensitive APs. The RIV is calculated according to bayesian probabilistic models that compare the APs required by each app with those requested by other apps in the same category (that must be known a priori). Furthermore, authors define three properties that should be granted by any probabilistic function calculating a RIV for apps, namely, i) *monotonicity* (i.e., removing an AP should lower the RIV), ii) *coherence* (i.e., malware should have higher RIVs than apps), and iii)

[4] It depends on the Android version. Older Android versions (< v. 6) require all permissions to be granted at install time, while newer versions allow the user to grant them dynamically at runtime.

[5] `https://developer.android.com/guide/topics/manifest/permission-element.html`

ease of understanding (i.e., the RIV of an app should be clearly understandable to the user, and it should allow straightforward comparison among values).

Also [4] proposes a methodology for calculating a RIV for apps according to their category. More specifically, for each category, the kind and number of required APs are empirically inferred, thereby identifying permission patterns belonging to apps in each category. Then, the RIV is calculated by measuring a *distance* between the set of APs required by the app and the permission patterns of its category. Notwithstanding the encouraging empirical results obtained on a dataset made by 7.737 apps and 1.260 malware samples, the main limitation of the approach is in the need to know in advance the category of the app. Such information can be often unreliable as categories are manually chosen by developers[6]. Maetroid [5] evaluates app risk according to both APs and metadata information related to the developer's reputation and the source app market. The risk is calculated according to declared APs only, and by assigning static weights to each AP. Maetroid does not provide a quantitative RIV, but assigns each app in one (out of three) risk category. A *framework* for app risk analysis is discussed in [6]. It is made by three layers carrying out static, dynamic and behavioral analysis, respectively. The framework combines the results from each layer and builds up the RIV. Unluckily, the framework is purely theoretical and lacks of any empirical evaluation, thereby making difficult to assess the viability of the approach. DroidRisk [7] is a quantitative method for calculating a RIV. DroidRisk is trained on a set of 27.274 apps and 1.260 malware samples, whereby it calculates the distribution of declared APs (i.e., those contained in the *Android Manifest* file). Then, DroidRisk applies a probabilistic function that calculates a RIV according to the kind and the potential impact of APs required by the app. More specifically, DroidRisk calculates a RIV for an app A according to two values for each AP p_i, namely the probability and the impact. Slightly extending the original notation, given a set of APs S, the probability $L(p_i, S)$ is the probability that $p_i \in S$ is required in the dataset, i.e., the number of apps requiring p_i on the total set of apps in the dataset; the impact $I(p_i, S)$ is a weight statically applied to each $p_i \in S$ according to its category (i.e., $I(p_i, N) = 1$, $I(p_i, D) = 1.5$, where N stands for the set of *Normal* APs and D for the set of *Dangerous* ones). Then, the RIV R_A for an app A is calculated as $\sum_{p_i \in \{N \uplus D\}} (L(p_i, \{N \uplus D\}) * I(p_i, \{N \uplus D\}))$, where $\uplus$ indicates the *disjoint union* between N and D.

Discussion. We argue that probabilistic methods suffer from some limitations.

1. They are unable to recognize as dangerous the malware that require a limited set of APs; conversely, they averagely provide high RIVs for apps requiring many APs.
2. Current proposals deal with declared APs only, without deepening, for instance, which APs are actually exploited by the app. Due to the monotonicity

[6] `https://support.google.com/googleplay/android-developer/answer/113475?hl=en`

of probabilistic risk indexes, relying only on declared permissions can impact the reliability, as apps are often overprivileged by their developers [2] and can therefore obtain too high RIVs.
3. Probabilistic methods statically define the impact of APs, that is, all APs belonging to the same category (e.g., *Normal*, *Dangerous*, *Signature*, *Signature-OrSystem*) equally impact the estimation of the RIV. This choice does not allow to provide different impacts to APs, e.g., according to their distribution on the set of malware.
4. The validity of RIV is strictly dependent with the chosen dataset, as well as the ratio between apps and malware samples; therefore, the dataset should be large enough - w.r.t. the set of available apps and malware samples - to be statistically significant to calculate a reliable RIV.

We argue that more reliable RIVs can be obtained through a machine learning approach based on

- four sets of permissions for each app A, namely
  1. *Declared* Permissions (DAP_A), i.e., declared in the Android Manifest file;
  2. *Exploited* permissions (EAP_A), i.e., APs that are actually exploited in the app code;
  3. *Ghost* permissions (GAP_A), i.e., APs that the app tries to exploit in the code, but they are not declared in the Android Manifest file;
  4. *Useless* permissions (UAP_A), i.e., declared APs that are not exploited in the app code.
- a statistically significant dataset. Our dataset is made by 112.425 apps and 6.707 malware samples from different sources. In details, apps comes from the Google Play Store[7] (98.162 apps), Aptoide[8] (7.516 apps), and Uptodown[9] (6.747 apps). Malware samples have been mostly taken from the DREBIN dataset [8] (5.560 samples); the remaining samples come from publicly available repositories, namely the Contagio dataset [9], the Husted's dataset [10] and the Bhatia's dataset [11].
- *dynamic impact* for each AP, calculated on the basis of its distribution on the whole dataset. The aim is to weigh APs according to their statistical distribution over malware samples and apps.

4 Reliability of Probabilistic Risk Indexes

In this section, we empirically evaluate the reliability of RIVs calculated through probabilistic methods. To this aim, we extend the methodology proposed in [7] by introducing the notion of `dynamic impacts`. Dynamic impacts allow to take into account the characteristics of a statistically significant dataset in the calculation

[7] `http://play.google.com/store/apps`
[8] `http://www.aptoide.com/page/apps`
[9] `http://en.uptodown.com/android`

of a probabilistic RIV. It is worth noting that all current proposals adopt static impacts, i.e., defined according to some heuristics but independently from the characteristics of the dataset. In order to apply dynamic impacts, an extensive statistical analysis on the dataset must be carried out in advance.

4.1 Statistical Analysis on APs

We took into account the dataset described in the previous section, i.e., made by 112.425 apps and 6.707 malware samples. We systematically extracted information on the four sets of permissions from each app in the dataset. We built a Permission Checker tool that given an app A (i.e., an APK file) in input, it provides back statistics on each AP set. DAP_A is straightforwardly retrievable from the *Android Manifest* file, while EAP_A, GAP_A and UAP_A are inferred through static analysis. More in details, the Permission Checker carries out reverse engineering on the APK to retrieve the app bytecode. Then, for each method invocation in the bytecode, the Permission Checker analyzes the APs required to execute the method. In the end, the Permission Checker builds a set PS_A containing all APs exploited in the bytecode. Remaining sets are built as follows:

- $EAP_A = \{p_i | p_i \in DAP_A \wedge p_i \in PS_A\}$;
- $GAP_A = \{p_i | p_i \notin DAP_A \wedge p_i \in PS_A\}$;
- $UAP_A = \{p_i | p_i \in DAP_A \wedge p_i \notin PS_A\}$;

We indicate with DAP, EAP, GAP, and UAP, the disjoint union of single app permissions sets, for all apps in the dataset, namely: $DAP = \biguplus_A DAP_A$, $EAP = \biguplus_A EAP_A$, $GAP = \biguplus_A GAP_A$, and $UAP = \biguplus_A UAP_A$.

Tab. 1: Statistics on APs on the dataset

	MALWARE			**APPS**		
AP Set	**MAX AP**	**AVG AP**	**Std. dev.**	**MAX AP**	**AVG AP**	**Std. dev.**
DAP	87	10.67	5.76	96	5.84	4.39
EAP	15	4.25	3.19	24	3.81	2.40
GAP	9	1.15	1.26	23	2.9	2.11
UAP	84	6.42	4.58	91	2.03	2.78

Discussion. Tab. 1 summarizes global statistics on the four AP sets. Such values indicate that malware declare more APs than apps on average (i.e., 10.67 vs. 5.84) but they exploit very few of them (i.e., 4.25). Furthermore, malware seldom try to exploit undeclared APs ($AVG_{GAP} = 1.15$) in comparison to apps ($AVG_{GAP} = 2.9$). Fig. 2 and Fig. 3 show the distribution of the top ten APs for malware and apps, respectively. For each AP, the y-axis shows the percentage of malware/apps having the AP.

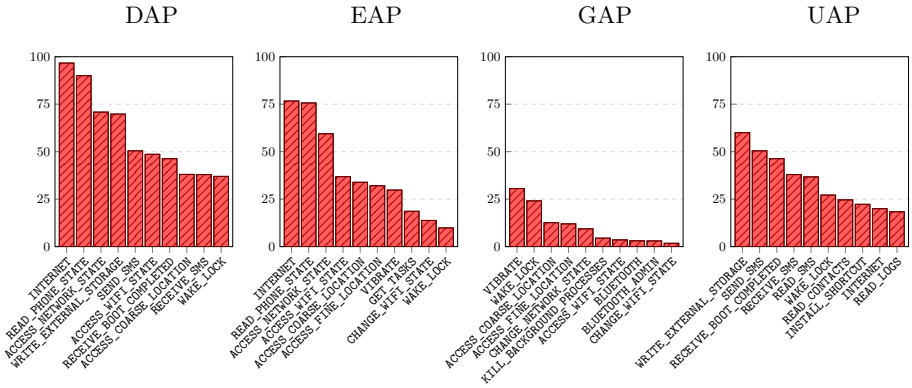


Fig. 2: Top 10 APs for malware

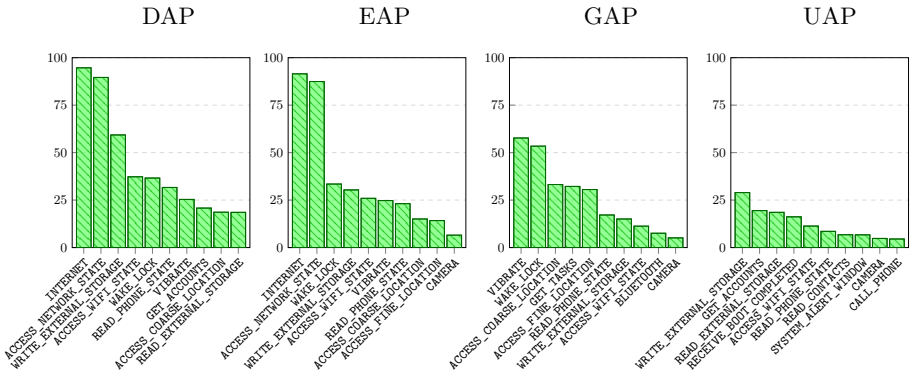


Fig. 3: Top 10 APs for apps

Some APs related to networking are equally divided between malware and apps, e.g., `INTERNET`, `ACCESS_NETWORK_STATE` and `ACCESS_WIFI_STATE`; since apps often require to connect to Internet, it is difficult to evaluate the RIV according to these APs. Other APs are required more frequently by malware than apps; for instance, a comparison between DAP plots in Fig. 2 and Fig. 3 suggests that an app requiring the `READ_PHONE_STATE`, the `RECEIVE_BOOT_COMPLETED` and the `READ_CONTACTS` could be potentially dangerous. The biggest gap between malware and apps is related to SMS APs; in fact, as shown in the DAP plot of Fig. 2, 2 out of 10 APs deal with SMS (i.e., `SEND_SMS` and `RECEIVE_SMS`), while no SMS-related APs appear in the DAP plot of Fig. 3. It is worth noticing that albeit almost 50% of malware require `SEND_SMS`, and more than 40% require `RECEIVE_SMS`, they seldom exploit them (as shown by the absence of such APs in the corresponding EAP set)[10].

[10] Complete statistics are available at: `http://www.csec.it/riskindroid`.

4.2 Dynamic Impacts

We argue that calculating a RIV according to the distribution of APs on malware and apps in the dataset may improve the accuracy of current probabilistic risk indexes. To empirically assess this thesis, we apply the probabilistic method proposed in DroidRisk [7] on our dataset using both static and dynamic impacts. We consider static impacts as defined in the original DroidRisk paper, namely, $I(p_i, N) = 1$ for p_i being a *Normal* AP, and $I(p_i, D) = 1.5$ for p_i being a *Dangerous* one. We define a dynamic impact as follows:

$$I(p_i, S) = \frac{P(p_i|M, S)}{P(p_i|A, S)} \tag{1}$$

being $P(p_i|M, S)$ the probability that a malware requires p_i in the set S, and $P(p_i|A, S)$ the probability that an app requires p_i in the same set S. In this way, the impact value increases as APs are more often required by malware than apps, and vice versa. Note that also in this case, the probability for p_i is calculated as the number of malware/apps requiring p_i on the total number of malware/apps in the dataset. It is also worth noting that the value of dynamic impacts is independent from the AP category. Since DroidRisk takes into consideration only declared permissions, we calculated dynamic impacts only for declared APs, i.e., $I(p_i, DAP)$. Tab. 2 shows an excerpt of dynamic impacts w.r.t. the dataset.

Tab. 2: Dynamic impacts for DAP on the full dataset.

AP values in DAP	$P(p_i\|M, DAP)$	$P(p_i\|A, DAP)$	**Dyn. Imp.**	AP values in DAP	$P(p_i\|M, DAP)$	$P(p_i\|A, DAP)$	**Dyn. Imp.**
INTERNET	96.66	94.83	1.02	READ_PHONE_STATE	90.04	30.97	2.91
ACCESS_NETWORK_STATE	70.84	89.32	0.79	WRITE_EXTERNAL_STORAGE	69.79	58.49	1.19
SEND_SMS	50.45	1.83	27.51	ACCESS_WIFI_STATE	48.61	36.02	1.35
RECEIVE_BOOT_COMPLETED	46.31	14.55	3.18	ACCESS_COARSE_LOCATION	38.03	18.26	2.08
RECEIVE_SMS	37.93	1.80	21.02	READ_SMS	36.72	1.48	24.88
ACCESS_FINE_LOCATION	36.29	16.77	2.16	READ_EXTERNAL_STORAGE	6.38	17.38	0.37
READ_CONTACTS	24.66	5.73	4.31	READ_CALL_LOG	0.46	0.91	0.51
READ_SYNC_SETTINGS	0.60	0.73	0.82	WRITE_SYNC_SETTINGS	0.72	0.78	0.92
RECORD_AUDIO	12.26	5.40	2.27	READ_CALL_LOG	0.46	0.91	0.51
READ_CALENDAR	10.97	1.16	9.44	NFC	0.03	0.61	0.05

4.3 Evaluating Probabilistic Methods

We carry out an empirical assessment aimed at evaluating i) if the usage of dynamic impacts could improve the quality of probabilistic RIV, ii) to which extent probabilistic methods are reliable, and iii) understand potential improvements towards more reliable RIVs.

Discussion. Our analysis indicates that the average RIV for apps is slightly lower with static impacts (i.e., 52.87 vs. 58.43); on malware, this gap is wider (i.e., 71.29 vs. 86.10). Fig. 4 shows the RIV distribution for both malware and apps in the dataset based on static and dynamic impacts. We consider 20 classes of RIVs, each comprising all apps having a RIV between $5i$ and $5i + 5\%$, where

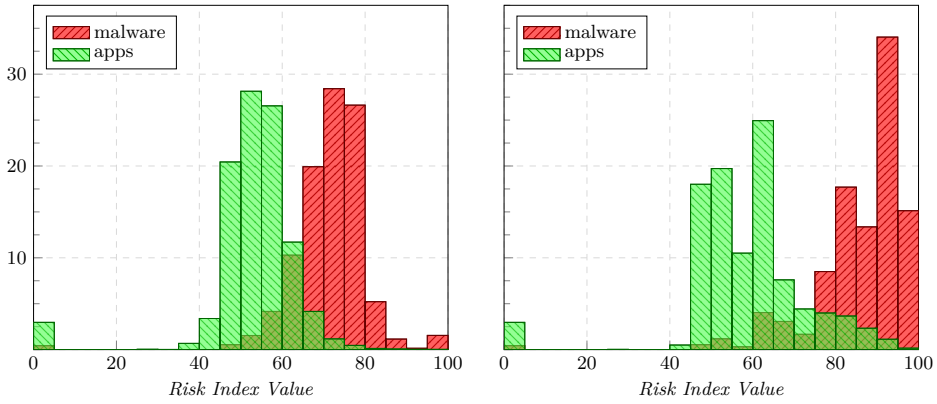


Fig. 4: Risk Index Values with static (left) and dynamic (right) impacts.

$i \in \{0, \ldots, 19\}$. The x-axis of each plot indicates the RIV, while the y-axis indicates the number of RIVs in each class. It is worth noting that in both cases malware have higher RIV on average, thereby suggesting that probabilistic methods are reliable in principle. However, our results also bring out their limitations. First, malware and apps histograms in Fig. 4 often overlap, thereby indicating that probabilistic methods may sometimes provide similar RIVs for malware and apps. In this case, the reliability of RIV depends on the gap between the overlapping histograms. For instance, let us consider the 60%-65% class for static impacts, where both histograms are almost equal; this indicates that each app having a RIV in this interval have rather the same chance to be a malware or not: this would be acceptable for RIVs around 50% only. Dynamic impacts allow to keep the gap in each class wider, at the cost of widening the overlap interval (i.e., histograms overlap from 50% and 80% with static impacts, and from 40% to 95% for dynamic impacts). Furthermore, RIV is averagely high for apps ($> 40\%$) and it does not span on the whole value interval (i.e., from 0% to 100%). Finally, as previously conjectured, probabilistic methods are unable to recognize as risky the malware that declare few or none APs (consider the overlap on class 0%-5% in both plots).

5 RiskInDroid: a Machine Learning-based Risk Index

We argue that the intrinsic limitations of probabilistic methods applied to APs can be overcome by machine learning techniques able to build up more reliable RIVs. In this section we present the methodology at the basis of RiskInDroid, then we provide an extensive empirical assessment of the tool.

5.1 Methodology

Machine learning techniques are used for classifying elements, i.e., given a set of classes, they evaluate each element and assign a class to it. Therefore, they are particularly suitable for binary classification of malware. However, some techniques also provide a probability value related to the prediction. We leverage machine learning techniques to classify apps into two classes, i.e., *malware* and *non malware*, and we use the classification probability to build up a RIV. For our purpose, we adopt the *scikit-learn* library [12], that implements a set of machine learning techniques and provides a probability function for some of them.

Machine learning techniques require feature vectors to compare and classify elements. In our context, elements are apps, and features are APs. We define feature vectors as follows: given $APSet$ the set of APs, for each app A we define four feature vectors FV_S^A, with $S \in \{DAP_A, EAP_A, GAP_A, UAP_A\}$. Each FV is a binary vector of cardinality |APSet|, where $FV_S^A[i] = 1$ if $p_i \in S$, and $FV_S^A[i] = 0$ otherwise. We adopt a supervised learning approach. Supervised learning requires classifiers to be trained on a training set before being applied to classify new elements. We train a set of supervised classifiers on a subset of the dataset and then we use them to classify the remaining APKs.

5.2 Selection of classifiers

The scikit-learn library implements 15 *supervised* classifiers with probability estimation, which means that they adopt proper techniques to provide a probability value for each classification result (also for algorithms that do not natively provide a probability on classification like, e.g., SVM and Decision Trees). In order to choose the more reliable ones, we empirically evaluated them on three sets randomly extracted from the dataset and containing the same number of apps and malware samples each (i.e., 6.707 malware samples and 6.707 apps), considering only DAP as permission set. We select classifiers according to three empirical rules:

1. **Accuracy > 90%**, in order to discard the less reliable classifiers.
2. **4% < AVG Score < 95%**, to avoid binary classifiers, i.e., that tend to provide scores around 100% for malware and 0% for apps.
3. **5% < Std. Dev.** to exclude classifiers that distribute in a little subset of the whole interval.

We evaluated the classifiers (using the default parameters provided by scikit-learn) by applying the `K-fold cross validation` [13] with $K = 10$. In a nutshell, the K-fold cross validation (see Fig. 5 for an example with $K = 4$) is an iterative statistical method where the dataset is divided into K independent sets (i.e., $folds$), each with approximately the same number of elements. At each iteration on K, the i^{th} fold acts as the *testing set*, while the remaining $k-1$ folds form the training set. The testing set is used to validate the model built through the training set. The accuracy value is calculated according to the number of

samples in the testing set whose class have been predicted correctly. The advantage of K-fold cross validation is that all samples are used both to train and to test the model, thereby reducing the *overfitting* problem that occurs when a model classifies correctly in the training set but not in the testing one.

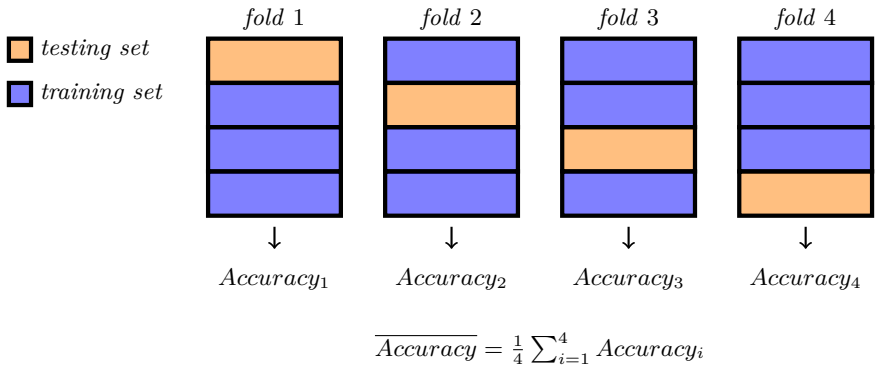


Fig. 5: Example of 4–*fold cross validation*

At each iteration, a classifier is trained on a training set of about 1342 elements (i.e., 671 apps and 671 malware samples) and tested with the remaining 9 sets, assuming that a score (i.e., the probability associated with the classification) $\geq 50\%$ implies recognizing the element as malware, while a score $< 50\%$ implies that the element is not malware. By comparing the nature of the element with its classification, we are able to recognize the correctness of the evaluation. The accuracy value is calculated as the ratio between the number of correct classifications on the total number of classified elements. The average score and standard deviation (i.e., σ) statistics are calculated on the classification probabilities returned by classifiers in the testing phase. Results are reported in Tab. 3. Since all classifiers had a very similar behavior on all three sets, we report the average value for each metric.

Discussion. GNB, BNB and QDA grant low accuracy, while DT, RF, SGD, LR-CV, K-NN, LDA and MPNN have too high average score for apps. Finally, AB has a low standard deviation and provides similar scores for malware and apps (i.e., from 47% to 53% in both cases). Only four classifiers meet all requirements, namely, SVM, MNB, GB and LR. Therefore, we chose to adopt them in RiskInDroid.

6 Experimental Results

RiskInDroid has been developed in Python and implements the selected four classifiers. For each app A, RiskInDroid calculates the RIV on all four APs sets

Tab. 3: Empirical evaluation of classifiers in the *scikit-learn* library

Classifier	**AVG Accuracy**	**Malware**		**Apps**	
		AVG Score	σ	**AVG Score**	σ
Support Vector Machines (SVM)	94.89	94.83	7.42	4.73	8.34
Gaussian Naive Bayes (GNB)	84.64	99.87	1.82	0.05	1.11
Multinomial Naive Bayes (MNB)	90.69	94.88	7.65	4.89	6.29
Bernoulli Naive Bayes (BNB)	89.97	99.07	4.87	0.69	4.19
Decision Tree (DT)	95.68	99.68	3.29	0.73	3.62
Random Forest (RF)	96.73	97.31	8.19	4.09	8.87
AdaBoost (AB)	94.19	52.83	1.45	47.48	1.44
Gradient Boosting (GB)	95.11	94.28	8.99	6.88	10.26
Stochastic Gradient Descent (SGD)	93.62	97.61	6.89	4.80	9.30
Logistic Regression (LR)	94.96	93.36	8.23	4.85	9.38
Logistic Regression CV (LR-CV)	94.93	96.41	8.21	4.71	9.21
K-Nearest Neighbors (K-NN)	94.29	98.69	6.22	4.82	11.34
Linear Discriminant Analysis (LDA)	93.88	98.11	6.42	1.93	6.18
Quadratic Discriminant Analysis (QDA)	78.18	100	0.31	0.06	1.32
Multilayer Perceptron Neural Network (MPNN)	97.06	99.12	4.31	1.68	5.51

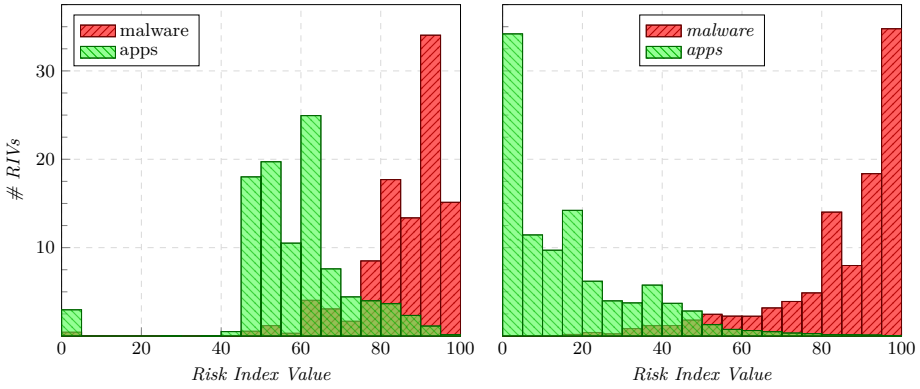
(i.e., DAP_A, EAP_A, GAP_A, and UAP_A), by combining the corresponding feature vectors in a unique one, i.e., $FV^A_{all} = FV^A_{DAP_A} \| FV^A_{EAP_A} \| FV^A_{GAP_A} \| FV^A_{UAP_A}$. The RIV is calculated as the average score value of all four classifiers. To train each classifier in RiskInDroid, we applied the 10-fold cross validation on one of the three sets used to evaluate the classifiers. We also used the same set to empirically assess whether applying all four APs sets may improve the accuracy. To this aim, our tests returned the following average accuracy values: 92.93% for DAP, 88.36% for EAP, 79.12% for GAP, 91.09% for UAP, and 94.87% for all sets. Therefore, we chose to consider all sets.

Tab. 4: Average RIV calculated by probabilistic methods and RiskInDroid.

APK Category	**Static Impacts**	**Dynamic Impacts**	**RiskInDroid**
Malware	71.29	86.10	84.34
Apps	52.87	58.43	16.89

Discussion. Tab. 4 shows the average RIV calculated by RiskInDroid, w.r.t. probabilistic methods in the previously discussed configurations. RiskInDroid substantially lowers the average RIV for apps. Fig. 6 compares the distribution of RIVs with probabilistic methods based on dynamic impacts and RiskInDroid.

The latter distributes RIVs on the whole risk interval, and restricts the histogram overlapping in the center of the interval. This is reasonable as the median value implies the maximum uncertainty (i.e., $RIV = 50\%$ means that the APK has the same probability to be malware or not).



(a) Dynamic impacts (b) RiskInDroid

Fig. 6: Distribution of RIV: probabilistic methods vs RiskInDroid.

Tab. 5: App analysis with VirusTotal: Experimental Results.

APK source	**# apps with RIV >75%**	**% of apps with Flags >X**				
		Flags >1	Flags >3	Flags >5	Flags >10	Flags >15
Google Play	635	14.6%	9.4%	8.7%	7.8%	7.1%
Aptoide	125	26.5%	12.6%	123%	12%	8.5%
Uptodown	86	24.4%	15.6%	8.9%	4.4%	2.2%

RiskInDroid and Malware Detection. We further evaluated the reliability of RIVs by assessing the relationship between apps with high RIVs and malware. More in detail, we selected all apps having RIV>75%, and we analyzed them through VirusTotal[11], a free suite hosting more than 50 online antivirus. Such antivirus are *signature-based*, i.e., they compare the app with a set of known malware footprints. For each analysis, VirusTotal also provides the number of antivirus (i.e., *flags*) recognizing the submitted APK as malware.

Tab. 5 summarizes the results. They indicate that the methodology at the basis of RiskInDroid is promising and the corresponding RIVs are reliable, since

[11] `http://www.virustotal.com`

some apps having high RIV are also recognized as malware by VirusTotal. However, it is worth pointing out that high RIV does not necessarily imply that an app is malware. For instance, social network apps require a lot of dangerous permissions and manage user data; such apps are risky for the security and privacy of the end user, but they are not malware. Finally, the experiments with VirusTotal indicate that apps from Google Play are less likely to be malware w.r.t. those provided by Aptoide and Uptodown: this is an expected result as Google Play carries out security assessments on its apps[12].

Tab. 6: Performance of RiskInDroid on a set of 13.414 APKs (6.707 apps and 6.707 malware samples)

Classifiers	**Training Phase**				**Testing Phase**			
	DAP only		All sets		DAP only		All sets	
	AVG T [ms]	σ [ms]	AVG T [ms]	σ	AVG T [ms]	σ [ms]	AVG T	σ [ms]
SVM	43460	60	97170	870	15	4	18	5
MNB	32	4	53	11	6	3	7	3
GB	5620	52	21806	403	9	3	11	5
LR	81	9	188	11	4	2	5	2
Total	**49193**	**67**	**119220**	**890**	**34**	**12**	**41**	**15**

Performance of RiskInDroid. The performance of RiskInDroid has been evaluated on a general purpose desktop PC equipped with an Intel i7-3635QM @ 3.40 GHz, and 16GB RAM. Tab. 6 summarizes the results. Performance of classifiers is evaluated in terms of average time and standard deviation, during the training and the testing phase. Using all sets decreases the average performance up to 240% during the training phase. However, it is worth noticing that this phase is executed once at the beginning. Instead, the testing phase is very quick and lasts in few millisecs both with one and all sets, thereby suggesting to adopt all four sets to obtain a higher accuracy.

7 Conclusion and Future Work

In this paper we empirically assessed the reliability of probabilistic risk index approaches for Android apps, and we proposed a novel methodology based on machine learning aimed at overcoming the shortcomings of the probabilistic solutions. We implemented the methodology in a tool, RiskInDroid, that we empirically evaluated. Future development of this research includes extending the feature set beyond APs, by taking into account suspicious API calls and URLs, both recognizable in the bytecode through the static analysis technique we adopted to build the permission sets.

[12] `http://googlemobile.blogspot.it/2012/02/android-and-security.html`

References

1. Gartner, "Gartner Says Five of Top 10 Worldwide Mobile Phone Vendors Increased Sales in Second Quarter of 2016."
2. A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), pp. 627–638, ACM, 2011.
3. C. S. Gates, N. Li, H. Peng, B. Sarma, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Generating Summary Risk Scores for Mobile Applications," *IEEE Transactions on dependable and secure computing*, vol. 11, no. 3, pp. 238–251, 2014.
4. H. Hao, Z. Li, and H. Yu, "An Effective Approach to Measuring and Assessing the Risk of Android Application," in *Theoretical Aspects of Software Engineering (TASE), 2015 International Symposium on*, pp. 31–38, IEEE, 2015.
5. G. Dini, F. Martinelli, I. Matteucci, M. Petrocchi, A. Saracino, and D. Sgandurra, "Risk analysis of android applications: A user-centric solution," *Future Generation Computer Systems*, pp. –, 2016.
6. S. Li, T. Tryfonas, G. Russell, and P. Andriotis, "Risk Assessment for Mobile Systems Through a Multilayered Hierarchical Bayesian Network," 2016.
7. Y. Wang, J. Zheng, C. Sun, and S. Mukkamala, "Quantitative Security Risk Assessment of Android Permissions and Applications," in *IFIP Annual Conference on Data and Applications Security and Privacy*, pp. 226–241, Springer, 2013.
8. D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *NDSS*, 2014.
9. "Contagio mobile malware mini dump, Available: `http://contagiominidump.blogspot.com/`; [Accessed: March 27, 2017]."
10. N. Husted, "Android malware dataset," 2011.
11. A. Bhatia, "Collection of android malware samples."
12. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
13. G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R.* Springer Publishing Company, Incorporated, 2014.