DroidAutoML: A Microservice Architecture to Automate the Evaluation of Android Machine Learning Detection Systems

Yérom-David Bromberg(✉) and Louison Gitzinger(✉)

Université de Rennes 1, Rennes, France
{david.bromberg,louison.gitzinger}@irisa.fr

Abstract. The mobile ecosystem is witnessing an unprecedented increase in the number of malware in the wild. To fight this threat, actors from both research and industry are constantly innovating to bring concrete solutions to improve security and malware protection. Traditional solutions such as signature-based anti viruses have shown their limits in front of massive proliferation of new malware, which are most often only variants specifically designed to bypass signature-based detection. Accordingly, it paves the way to the emergence of new approaches based on Machine Learning (ML) technics to boost the detection of unknown malware variants. Unfortunately, these solutions are most often underexploited due to the time and resource costs required to adequately fine tune machine learning algorithms. In reality, in the Android community, state-of-the-art studies do not focus on model training, and most often go through an empirical study with a manual process to choose the learning strategy, and/or use default values as parameters to configure ML algorithms. However, in the ML domain, it is well known admitted that to solve efficiently a ML problem, the tunability of hyper-parameters is of the utmost importance. Nevertheless, as soon as the targeted ML problem involves a massive amount of data, there is a strong tension between feasibility of exploring all combinations and accuracy. This tension imposes to automate the search for optimal hyper-parameters applied to ML algorithms, that is not anymore possible to achieve manually. To this end, we propose a generic and scalable solution to automatically both configure and evaluate ML algorithms to efficiently detect Android malware detection systems. Our approach is based on devOps principles and a microservice architecture deployed over a set of nodes to scale and exhaustively test a large number of ML algorithms and hyper-parameters combinations. With our approach, we are able to systematically find the best fit to increase up to 11% the accuracy of two state-of-the-art Android malware detection systems.

Keywords: Machine learning · Android · Malware · AutoML

Published by Springer Nature Switzerland AG 2020
A. Remke and V. Schiavoni (Eds.): DAIS 2020, LNCS 12135, pp. 148–165, 2020.
https://doi.org/10.1007/978-3-030-50323-9_10

1 Introduction

Smartphones are currently generating more than half of the global internet traffic [1], and its related market surpasses by far computer sales. The Android operating system is one of the most important market players. It owns 70% of market shares, and accounts for 2.5 billion active devices worldwide [6]. Unfortunately, this boundless adoption opens a lucrative business for attackers and ill-intentioned people. The number of Android malware peaks in 2020 [10]. Malicious applications spread across the Android ecosystem at an alarming rate [3,4,7]. Attackers leverage on various techniques such as *dynamic code loading, reflection* and/or *encryption* to design ever more complex malwares [22,48] that systematically bypass existing scanners. To counter this phenomenon, Android security actors, from both research and industry, massively adopt machine learning techniques to improve malware detection accuracy [11,15,16,39]. Although it is a first step towards improving detection, unfortunately, most of related studies neglect the search for fine tuned learning algorithms.

We argue that there are still rooms for improvements unexplored. In particular, it is commonly admitted in the machine learning domain, that performances of trained machine learning models depend strongly on several key aspects: (i) training datasets [25], (ii) learning algorithms [20], and (iii) parameters (i.e. hyper-parameters) used to tune learning algorithms [19,21,23,46]. Accordingly, the key underlying problem, usually referred as *Automated Machine Learning* (AutoML) [29], is how to automate the process of finding the best suitable configuration to solve a given machine learning problem. As far as our knowledge, no attempts have been done towards improving Android malware detection systems based on machine learning algorithms. Whether one [11,16] or several algorithms [17,39,50] are evaluated, the evaluations are always carried out empirically, implying a manual process with few and/or default hyper-parameter combinations. Testing various algorithms along with a large set of hyper-parameters is a daunting task that costs a lot both in terms of time and resources [36].

In this paper, we present DroidAutoML, a new approach that automatically performs an extensive and exhaustive search by training various learning algorithms with thousand of hyper-parameter combinations to find the highest possible malware detection rate given the incoming dataset. DroidAutoML is both generic and scalable. Its genericity comes from its ability to be agnostic to underlying machine learning algorithms used, and its scalability comes from its ability to scale infinitely horizontally by adding as much as machines as required to speed up the processing. To achieve this aim, and leveraging our expertise in the field of Android malware detection, we have defined and deployed a dedicated microservices architecture.

Our contributions are as follow:

- We propose the very first *AutoML* approach, named DroidAutoML, to improve the accuracy of technics based on machine learning algorithms to detect malware on Android. With DroidAutoML, there is no need anymore to manually perform empirical study to configure machine learning algorithms.

– We provide a dedicated microservices architecture specifically designed to fulfill the needs for genericity and flexibility as required by the Android malware detection domain.
– We thoroughly evaluate our approach, and applied it to the state of the art solutions such as Drebin [16] and MaMaDroid [39]. We demonstrated that DroidAutoML enables to improve significantly their performances: detection accuracy has been increased up to 11% for Drebin and 10% for MaMaDroid.

The remainder of this paper is organized as follows: Sect. 2 explains the context of the study. Section 3 presents in details our microservices architecture, and Sect. 4 details our thorough evaluation. We make a review of the state of the art in Sect. 5, and finally conclude the paper in Sect. 6.

2 Background

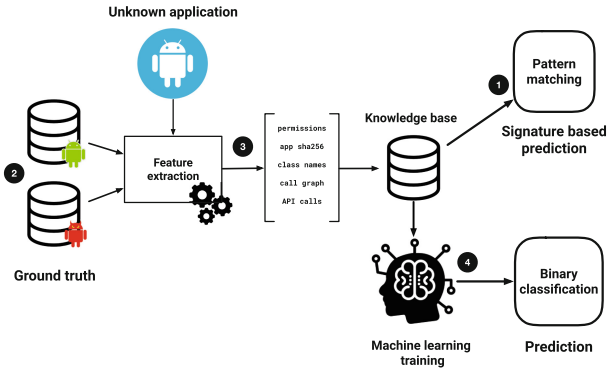

Fig. 1. Overview of the malware detection process on Android

2.1 Emergence of Machine Learning Algorithms to Detect Android Malware

Traditional anti-viruses heavily rely upon signatures to identify malware [8,9] (see Fig. 1, ❶). As soon as new malware are discovered in the wild, antivirus software companies put their hands on, and compute their related signatures. The latter are added to the ground truth database (see Fig. 1, ❷) of the antivirus software. In this way, anti-viruses calculate the signatures of files to be analyzed (see Fig. 1, ❸), and compare them with previously stored signatures to perform the detection of malware. While signature based detection is efficient to catch old and already seen malware, they struggle to deal with new malware generations [22,45]. Indeed, malware authors use various techniques, such as *polymorphism* [31], to generate malware variants that inherently have unforeseen signatures to bypass anti-viruses. These new attacks emphases the need for

more intelligent systems to detect proactively unseen malware variants, commonly known as 0-day threats.

In this aforementioned perspective, the last decade, strong efforts have been achieved to generalize the problem, and to develop new approaches based on machine learning (ML) [11,16,17,39,50] (see Fig. 1, ❹). In contrast to signature based anti-viruses, ML anti-viruses rely on sets of meticulously chosen heuristics, or features (see Fig. 1, ❸) to train learning models from a ground truth (see Fig. 1, ❷) dataset that includes both benign and malicious applications. Once trained, models are thereafter able to make predictions on unseen files. They give either a confidence score [32] or take a binary decision to decide whether a file is benign or potentially harmful.

2.2 Importance of Features in ML Malware Classification Problem

Feature extraction, selection and encoding are essential steps towards building an efficient classification system based on ML. Features must be chosen in such a way that they help ML algorithms to generalize the classification problem and help them to adequately classify them. When badly chosen, algorithms may be unable to generalize the problem or suffer from *overfitting*. More importantly, the number of used features can drastically slow down model training time.

Table 1. Examples of basic and behavioral features that can be extracted from an Android application

Type	Features	Static analysis	Dynamic analysis	Location
Basic	Permissions	✓	✗	manifest
	Intent-filters	✓	✗	manifest
	Components	✓	✗	manifest
	File signatures	✓	✗	apk level
	Protected method calls	✓	✓	bytecode
	Suspicious method calls	✓	✓	bytecode
Behavioural	Call graph	✓	✓	bytecode
	Dynamic code loading	✗	✓	apk level
	Network traffic	✗	✓	OS level
	Intent messages	✗	✓	OS level

Feature Selection. Feature selection is a widely studied area regarding ML malware detection in the Android ecosystem [11,16,39]. Mainly, we distinguish two types of features: (i) basic, and (ii) behavioral features, as illustrated in Table 1. Basic features qualify information inherent in an application, but by themselves do not directly encode its corresponding behavior [16]. It is the correlation of the features altogether that allows machine learning models to differentiate a benign application from a malicious one. For instance, the basic *permission* feature `READ_CONTACTS` can be used by both benign and malicious

applications. As such, it does not give any information about intentions of the application. However, when correlated with the basic *method call* feature `url.openConnection()`, it may highlight the intentions of a malicious application to steal user's contacts. Contrariwise, behavioral features are information about an application that allows to extract both intentions and actions of an application [39]. These features can be extracted by statically extracting the call graph of an application or by monitoring the application during its execution.

Feature Extraction. Mainly two approaches can be used to extract features from an Android application: (i) static, and (ii) dynamic analysis. A static analysis allows to quickly analyze binaries of Android applications without having to execute it. In the Android ecosystem, static analysis is a widely used approach [28,35,38,47] as they allow to analyze applications at scale without having an impact on resources, and in particular on the reactiveness of applications being scanned. However, static analysis are limited to the analysis of the visible part of the application's bytecode. Malware authors may use advanced obfuscations techniques such as *dynamic code loading* or *encryption* to try to defeat static analysis.

Due to the weaknesses of static analysis, dynamic analysis are often explored as an alternative in Android malware detection systems [27,49,51]. Dynamic analysis consists of executing malware to monitor their behaviors at runtime. Most often, to make it scales and for isolation purposes, such analyses are typically executed in *sandboxed* environments. However, malware may implement *evasion* techniques such as *logic bombs*, and *time bombs*, which allow them to bypass runtime surveillance. A *logic bomb* is the ability of a malware to detect its runtime environment (i.e. a sandboxed environment such as a virtual machine), and to prevent itself to trigger its own malicious behavior in such conditions [44]. A *time bomb* enables malware to trigger their malicious actions only after a certain amount of time or periodically, at specific hours. Accordingly, dynamic analysis suffer from scalability issues, and are rarely used due to their inherent strong requirements in terms of both time and resources.

Feature Encoding. Feature vectors are used to represent the characteristics of studied items in a numerical way to simplify their analysis and comparison. Most of ML classification algorithms such as neural networks, nearest neighbor classification or statistical classification use feature vectors as input to train their model. While it is easy to use pixels of an image as a numerical feature vector, it is harder to numerically encode more complex features such as basic or behavioral features of Android applications. For that reason, many studies [11,16,26,39] provide new alternatives for feature encoding. Authors of Drebin [16] embed *basic* extracted features into a feature vector using *one-hot encoding* to code the presence, or the absence of a given feature. Contrariwise, MaMaDroid [39] encodes *behavioral* extracted features using a Markov chain algorithm, which calculates the probability for a method to be called by another in a call graph.

2.3 Choosing and Training the Classification Algorithm

While feature selection and encoding remain important in machine learning based malware detection, the problem of training an accurate model also needs to be addressed. On one side, Android application vendors must ensure that no malicious applications bypass their security barriers. On the other side, discarding too many applications, to stay conservative, may lead to profit losses, as many benign applications can be flagged as false positive. Hence, training a binary classifier with good performances in terms of precision and recall is essential. While features selection can be very helpful to solve this, choosing the best classification algorithm with the best training parameters can greatly improve classification.

Classification Algorithms. In machine learning, and especially on binary classification problems, it is admitted that choosing the right learning algorithm depends on many factors such as the available resources, the algorithm complexity or the input data. As there is no silver bullet to always find the best algorithm, researchers often go through an empirical process to find a good fit. Regarding the Android ecosystem, various algorithms to train models, mostly Random Forest (RF), Support Vector Machine (SVM), and k-nearest neighbors (KNN), have already been tested depending on type of data extracted from applications, and the number of applications used for training ML models [16,33,39,50]. Although all these studies show good evaluation performances, all of them have been empirically evaluated with a manual trial and error strategy. As it is a very time consuming task, it is a safe bet to say that these studies did not found the best learning algorithm to solve the classification problem. Therefore, we claim that automating such a task would be a great help for the research community.

Hyperparameter Optimization. Another important aspect are parameters used to train chosen learning algorithms (most often set to default values). Usually, the number of *hyper-parameters* for a given algorithm is small ($\leq$5), but may take both continuous and discrete values leading to a very high number of different values and so of combinations. For instance, common hyper-parameters include the *learning rate* for a neural network, the C and *sigma* for SVM, or the K parameter for KNN algorithms. The choice of hyper-parameters can have a strong impact on performances, learning time and resource consumption. As a result, *Automated hyper-parameter search* is a trending topic in the machine learning community [37,52]. Currently, grid search and brute force approaches remain a widely used strategy for hyper-parameter optimization [18] but can require time and computational resources to test all possibilities. To deal with this issue, several frameworks are able to efficiently parallelize grid-searching tasks on a single machine, but this does not scale with the ever growing search space [12,41].

3 A Microservice Architecture for ML

DroidAutoML relies on a microservice architecture that separates concerns between data processing (feature *selection, extraction* and *encoding*) and training optimization ML models. Such a design enables DroidAutoML to scale and stay agnostic to the evaluated scanner.

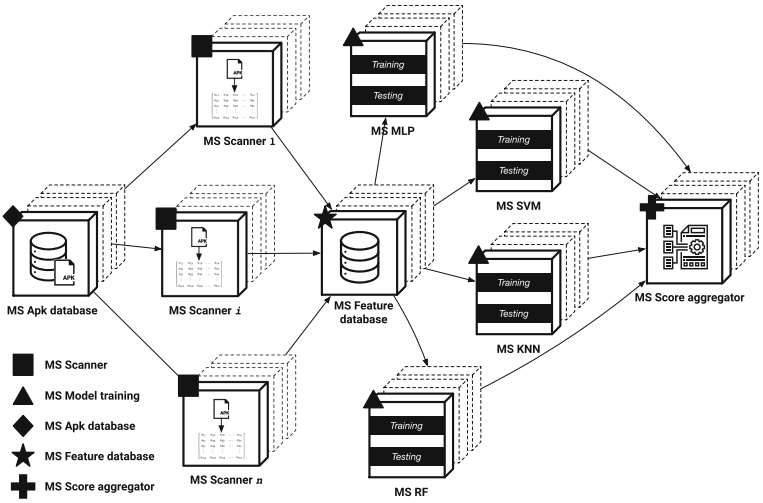

Fig. 2. Overview of DroidAutoML

Microservices Dedicated to Features Operations. Feature extraction and *encoding* are both operations specifics to each scanner. As such, each scanner has its own dedicated microservice for performing these operations (Fig. 2, ■). We define k as the number of applications to process for a given dataset. For n different scanners, $n * k$ instances of $\blacksquare_{(i,j)}$ microservices with $i \in \{1..n\} \wedge j \in \{1..k\}$ will be deployed. Each $\blacksquare_{(i,j)}$ instance takes as input an *apk* to generate its corresponding features vector, interpretable by any machine learning algorithms. The generated *feature vector* is then stored into the *feature database* microservice (See Fig. 2,★).

Microservices Dedicated to Model Training. ML model training operations are specific to a classification algorithm and the set of *hyper-parameter* used to parametrize it. Therefore, each algorithm has its own dedicated microservice to perform the training and testing of a model for one *hyper-parameter combination* (see Fig. 2, ▲). For l different algorithms, l different kinds of m instances of $\blacktriangle_{(i,j)}$ with $i \in \{1..l\} \wedge j \in \{1..m\}$ will be deployed where m is equals to the number of *hyper-parameter combinations* to test for a given algorithm. This allows to scale horizontally by spreading the workload across the available nodes in the cluster. A ▲ microservice takes two inputs: (i) a feature vector matrix from the feature

database ★, and (ii) a set of hyper-parameter values. ▲ microservices leverage *Scikit-learn* to perform both training, and testing steps. Afterwards, each ▲ instance parametrizes its ML algorithm according to the input hyper-parameter combination. All ML models are trained with a 10-cross fold validation process to avoid *overfitting* problems. The input data is split according to machine learning ratio standards: 60% of the data is used to fit the model and 40% to test it. Performances of each model are assessed in terms of *accuracy* and *F1 score*. Finally, trained models are stored within the database along with the configured hyper-parameter settings so that they can further be used by the end-user. The obtained results on the testing set are then communicated to *score aggregator* microservices (see Fig. 2, ✚).

Microservices Dedicated to Score Aggregation. A third set of microservices are the ones dedicated to the collecting of results from ▲ mircroservices to identify the pair {*algorithm,hyper-parameters*} that gives the best performances for a given scanner. Each score aggregator microservice is dedicated to a couple {*scanner, algorithm*} so that it collects only results related to it for all hyper-parameter combinations tested. Accordingly, for n scanners and l algorithms, there will be at least $n * l$ instances of aggregators. Once the best predictive model have been found for a given scanner, the corresponding algorithm and hyper-parameters are communicated to the end-user.

Efficient Microservice Scheduling. DroidAutoML is a system designed to run on top of a cluster of hardware machines. To optimize resources and efficiently schedule tasks on such a cluster, DroidAutoML leverages on a bin packing algorithm [24]. As such, by splitting scanner benchmarking operations into smaller tasks, DroidAutoML can capitalize on properties offered by microservice architectures. Firstly, DroidAutoML fully takes advantage of multi node clusters as each microservice can be scheduled independently on any node in the cluster. Secondly, as scanner benchmarks are parallelized, ▲ microservices can run side by side with ■ microservices as long as they do not work for the same scanner. Thirdly, if a microservice fails during its execution, only its workload is lost and it can be automatically rescheduled.

4 Evaluation

4.1 Implementation

DroidAutoML is built on *Nomad*, an open-source workload orchestrator developed by *HashiCorp* [5], which provides a flexible environment to deploy applications on top of an abstracted infrastructure. More precisely our Nomad instance federates a cluster of 6 nodes (see Fig. 3, ❶) that accounts for 600 GB of RAM and 124 cores at 3.0 Ghz. We use the bin packing algorithm implemented in *Nomad* to schedule (see Fig. 3, ❷) DroidAutoML microservices instances across available nodes in the cluster as schematized in Fig. 3. Each microservice instance is represented as a *job* managed by the *Nomad* scheduler. Hardware resources

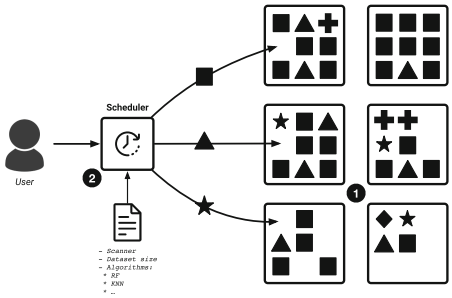

Fig. 3. Overview of DroidAutoML implementation

allocated to each microservice depend on its type: scanner specific instances take 2 cores and 4 GB of RAM each, model training instances take 1 core and 2 GB of RAM, and score aggregator instances take 1 CPU and 1 GB or RAM. The time required for a scanner instance to build a feature vector depends on the size of the input apk as well as its operating time. It ranges from 6 s for a 2 MB application to 61 s for a 100 MB application on average. The apk database of DroidAutoML is currently composed of 11561 applications, 5285 malware and 6276 benign applications and the average size of an application is 20.25 MB with a standard deviation of 21.48.

Given the resources required for one instance, our infrastructure can run 61 ■ microservice instances in parallel, therefore the entire apk database can be processed in 24 min with our current cluster. The time required to train and test a ML model depends on the algorithm, the set *hyper-parameters* used, and the size of the input vector matrix. We provide in Table 4 the minimum, average and maximum time required to train and test a model according to an algorithm. As we use a grid-search approach to perform *hyper-parameter tuning*, the number of ML models train to evaluate a scanner depends on the number of hyper-parameter combinations to test. The Table 4 summarizes the values tested for each *hyper-parameter* according to an algorithm as well as the number of combinations to test them all. For example, given the resource constraints of a ML model microservice, our cluster can run 123 ▲ microservice instances in parallel, thus testing all 3120 hyper-parameter combinations for the Random Forest takes on average 9 min for an input feature vector matrix of 11561 items.

4.2 Evaluation of Two State of the Art Scanners

To evaluate our approach, we propose to apply our microservice architecture to two state-of-the-art machine learning based malware detection systems in order to improve learning algorithm selection and training. More precisely, we conduct our experiments on approaches proposed by Drebin [16] and MaMaDroid [39]. We benchmark our approach against the ground truth of the related work by reusing the same ML algorithms used by the two approaches: Support Vector

Machine (SVM) for Drebin and Random Forest, SVM and K-Nearest Neighbors for MaMaDroid.

We build a dataset of 11561 applications composed of 5285 benign and 6276 malware samples. Malicious samples are collected from three malware datasets: the Drebin dataset [16], the `Contagio` dataset [2] and a dataset of 200 verified ransomware from `Androzoo` [13]. Concerning benign applications, we collect samples from the top 200 of most downloaded applications for each app category in the Google Play Store. To ensure that collected samples are really benigns, we upload them to VirusTotal, an online platform that makes it possible to have a file analyzed by more than 60 commercial antivirus products. According to the literature [40], applications can be safely labeled as benign if less than 5 antivirus detect it as malware, as several antivirus consider *adwares* as potentially dangerous. Among the 6276 applications downloaded, $95,04\%$ (5965 samples) have not been detected as malware at all and $99,18\%$ (6225 samples) by less than 5 antivirus. To guarantee the overall dataset quality, we remove all samples with a detection rate over this threshold.

Table 2. Baseline results for Drebin and MaMaDroid models trained with original *hyper-parameters* settings.

Scanner	Algorithm	Accuracy	F1-Score	Precision	Recall	TP	TN	FP	FN
Drebin	**SVM**	88.91	88.23	84.43	92.39	1833	2087	338	151
MaMaDroid	**KNN**	82.35	81.76	83.25	80.33	1744	1887	427	351
	Random Forest	80.54	83.08	72.65	97.01	2106	1445	65	793
	SVM	79.22	81.97	71.57	95.90	2082	1411	89	827

Ground Truth Results. As original experiments by Drebin and MaMaDroid authors were made on older data, both approaches may suffer from *temporal bias* [14,43]. *Temporal bias* refers to inconsistent machine learning evaluations with data that do not correctly represent the reality over time. To take this bias into account, we start our experiment by measuring ground truth results for both Drebin and MaMaDroid approaches using our own dataset. These results will serve as a baseline to evaluate DroidAutoML performances and compare further results against it. Authors from Drebin use a SVM algorithm to perform the binary classification of malware and benign applications. As the original source code of their approach is not available, we develop our own implementation of their solution using available information in the original paper. While our implementation of Drebin may slightly differ from the original one, the approach and the algorithm used (SVM) remain conceptually the same. As no details are given about *hyper-parameters* used to parametrize the algorithm, we take common default values suggested by machine learning frameworks to train the algorithm. Regarding MaMaDroid, authors tested three learning algorithms: Random Forest, SVM and KNN. We calculate the baseline by using the MaMadroid's approach source code, and the same *hyper-parameters* set by the authors.

The Table 3 reports the grid of *hyper-parameter* values used to train and test each learning models for both approaches. The Table 2 reports the baseline results for each trained model. We observe that the accuracy and F1 scores for both approaches decrease compared to the original results. The accuracy score for the Drebin SVM drops by 5.09% from 94% to 88.91%. Considering MaMaDroid, F1-Scores are below 84% for all studied algorithms, with a false-positive rate over 5%, which is more than 15% lower than best results presented originally in terms of F1-Score. As samples in our dataset are more recent than those used in original experiments, these results confirm that both Drebin and MaMaDroid approaches are suffering from *temporal bias*.

Table 3. Default hyper-parameters used to parametrize evaluated algorithms

	Parameters	Mamadroid	Drebin
Random Forest	n_estimators	101	
	max_depth	32	
	min_samples_split	2	
	min_samples_leaf	1	
	max_features	auto	
SVM	C	1	1
	kernel	rbf	linear
	degree	3	3
	gamma	auto	auto
KNN	n_neighbors	[1, 3]	
	weights	uniform	
	leaf_size	30	
	p	2	

Model Evaluation with DroidAutoML. In the following of this experiment, we aim at answering the following questions:

- **RQ1**: Is DroidAutoML able to find a learning algorithm that performs better than default algorithms used for studied scanners?
- **RQ2**: Can DroidAutoML improve the prediction results of studied scanners by finding an optimal set of *hyper-parameters*?

We answer these questions by running DroidAutoML for each studied scanner with a large grid of *hyper-parameters* (see Table 4) and 4 different learning algorithms for each scanner: Random Forest, SVM, KNN, and a multi layer perceptron (Neural Network).

Table 4. Grid hyper-parameters used to train models with DroidAutoML

	Parameters	Hyperparameters	# of combinations to test	Time for a single run (in seconds for 11 238 apks)		
				min	**avg**	**max**
Random Forest	n_estimators	[200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]	10 * 13 * 3 * 4 * 2 = 3120	15	21	35
	max_depth	[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 500, 1000,None]				
	min_samples_split	[2, 4, 6, 10]				
	min_samples_leaf	[2,5,10,20]				
	max_features	[auto,sqrt]				
SVM	C	[0.0001,0.001,0.01,0.1, 1,10,100,1000,10000]	9* 4* 7 = 252	23	25	31
	kernel	[linear,rbf,sigmoid,poly]				
	gamma	[0.0001,0.001,0.01,0.1, 1,auto,scale]				
KNN	n_neighbors	[1,3,4,5,6,7,8,9,10]	9 *2 *8 *2 = 288	23	42	56
	weights	uniform,distance				
	leaf_size	[1,3,5,10,20,30,50,100]				
	p	[1,2]				
MLP	hidden_layer_sizes	[(50, 50, 50), (50, 100, 50), (100,)]	3 *2 *3* 2* 2 = 72	123	164	250
	activation	[tanh, relu]				
	solver	[sgd, adam, lbfgs]				
	alpha	[0.0001, 0.05]				
	learning_rate	[constant, adaptative]				

Table 5. Best results after model training on DroidAutoML

Scanner	Algorithm	Accuracy	F1-Score	Precision	Recall	TP	TN	FP	FN
Drebin	**KNN**	98.82	98.82	99.91	97.75	2169	2188	2	50
	Random Forest	98.57	98.56	99.63	97.52	2163	2183	8	55
	SVM	99.50	99.50	99.86	99.13	2168	2219	3	19
	MLP	99.61	99.60	99.68	99.54	2164	2228	7	10
MaMaDroid	**KNN**	85.48	86.41	93.69	80.17	2034	1735	137	503
	Random Forest	87.93	88.57	94.98	82.98	2062	1815	109	423
	SVM	88.97	88.49	86.09	91.03	1869	2054	302	184
	MLP	84.71	85.36	90.55	80.73	1966	1769	205	469

The Table 5 reports the best results obtained for both Drebin and MaMaDroid. For the Drebin approach, accuracy and F1 scores of the model trained with SVM increase by 10.59% and 11.27% respectively compared to the baseline. Moreover, we observe that the multi layer perceptron algorithm performs slightly better than the SVM algorithm with +0.11% in accuracy and +0.10% in F1-Score, thus reducing the number of false negative from 19 to 10. DroidAutoML also succeeds to improve MaMaDroid baseline results for all three studied algorithms. In details, DroidAutoML increases MaMaDroid's SVM

Table 6. Hyper-parameters found for best case performance

	Parameters	Mamadroid	Drebin
Random Forest	n_estimators	1600	1500
	max_depth	50	30
	min_samples_split	2	4
	min_samples_leaf	2	10
	max_features	sqrt	auto
SVM	C	1000	1000
	kernel	linear	rbf
	degree	3	3
	gamma	auto	scale
KNN	n_neighbors	3	5
	weights	uniform	uniform
	leaf_size	30	20
	p	2	2
MLP	hidden_layer_sizes	100	50, 100, 50
	activation	tanh	tanh
	solver	adam	lbfgs
	alpha	0.05	0, 0001
	learning_rate	adaptative	constant

baseline accuracy by 9.75%, KNN by 3.13% and RF by 7.39%. These accuracy improvements are accompanied by a significant increase of F1-scores for all algorithms. It represents a significant decrease of the number of false positives and false negatives. In their paper, MaMaDroid's authors discard the SVM algorithm due to poor performance compared to other algorithms tested. We show here that SVM is actually better than other algorithms tested by authors when it is parametrized with the adequate *hyper-parameter* values as shown in Table 6. Notice that in machine learning, optimal *hyper-parameters* values depends on the problem to solve [23]. Therefore, as the feature vectors are encoded differently for Drebin and MaMaDroid, optimal *hyper-parameter* values may slighlty differ from one approach to the other.

We answer **RQ1** by showing that DroidAutoML has been able to find a ML algorithm that performs better than those tested empirically with studied scanners. More precisely, the Multi Layer Perceptron outperforms the SVM algorithm used by Drebin originally and the MaMaDroid SVM originally discarded by the authors due to poor results performs better than other algorithms initially retained (i.e. RF and KNN).

Furthermore, we answer **RQ2** by showing that DroidAutoML has been able to find a combination of *hyper-parameters* in a reasonable amount of time (less than 30 min) that enables to significantly improve prediction results for all machine learning models trained for studied scanners.

5 Related Work

Machine Learning Based Malware Detection on Android. As of today, many studies [11,16,33,39,42,50,53] use machine learning to improve malware detection in the Android ecosystem. Over time, trained models become more and more accurate thanks to the heavy work on feature extraction and feature selection. Among studies published on the subject, several of them [11,16,42] use basic semantic feature to model application's behavior. Particularly, in 2014, authors of DREBIN [16] use various features such as permissions, application components, calls to hardware components, intent filters, etc. to train a support vector machine on more than 5000 malware and 123 000 benign applications. Other studies [39,53] model and encode the application control flow to increase the robustness against adversarial attacks [22,48] that modify the application's byte code without touching its behavior. Unfortunately, the great majority of these studies do not focus on model training, and most often go through a manual process to choose the learning strategy. Only a few studies [39,50] are actually testing more than one learning algorithms. However, the process is still done manually and *hyper-parameters* are empirically chosen or left by default.

Automated Machine Learning Frameworks. Several works already studied *automated machine learning* as a research problem [30,34]. These works have mainly paved the way to make machine learning available to non-experts from the domain. Frameworks such as *Auto-Sklearn* and *Auto-WEKA* related to these studies are actually responding to a demand for machine learning methods that automatically works without expert knowledge. With *Auto-Sklearn* [30], authors leverage on Bayesian optimization and past performance on similar datasets to automate classifier selection and increase trained model *efficiency*. However as stated before, the quality of a model training depends on the input data. While an *AutoML* framework may find an acceptable solution for a given problem, it is not sufficient in many expert domains, especially Android security and malware detection where the best possible efficiency is required. It is a big assumption to trust an *AutoML* framework to choose the best fit for the problem to solve. Especially in Android malware detection, input data can vary a lot depending on the feature selection and encoding approach. Moreover, while *Auto-Sklearn* can efficiently parallelize on a single machine, it is not designed to horizontally scale on a multi-node cluster. For that reason, we consider frameworks such as *Auto-Sklearn* as another option to test along with others classical classifiers such as Random Forest or SVM in DroidAutoML.

6 Conclusion

We have identified that machine learning solutions are underexploited in the Android ecosystem and proposed a novel approach to address this issue. We have built DroidAutoML, a microservice architecture to test malware detection scanners on a large number of machine learning algorithms and hyper-parameter combinations. We have shown that DroidAutoML can significantly improve scanners detection rate while optimizing used resources. DroidAutoML becomes a cornerstone to correctly benchmark both existing and novel ML approaches on existing ML algorithms.

As a future work we plan to integrate new machine learning algorithms in our framework and potentially more efficient approaches to speed up the hyper-parameter optimization process such as *Bayesian optimization*. We also plan to release DroidAutoML as an open-source framework, as the Android security community could greatly benefit from it.

References

1. Cisco visual networking index: Global mobile data traffic forecast update, 2017–2022. https://s3.amazonaws.com/media.mediapost.com/uploads/CiscoForecast.pdf. Accessed 20 042020
2. Contagio dataset. http://contagiodump.blogspot.com/. Accessed 12 Sept 2019
3. Cyber attacks on android devices on the rise. https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise. Accessed 30 Jan 2020
4. Mcafee mobile threat report q1 (2018). https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2018.pdf. Accessed 30 Jan 2020
5. Nomad by hashicorp. https://www.nomadproject.io/. Accessed 10 Feb 2020
6. There are now 2.5 billion active android devices - the verge. https://www.theverge.com/2019/5/7/18528297/google-io-2019-android-devices-play-store-total-number-statistic-keynote. Accessed 30 Jan 2020
7. Threat intelligence report 2019. https://networks.nokia.com/solutions/threat-intelligence/infographic. Accessed 30 Jan 2020
8. Virustotal. https://www.virustotal.com/gui/home/upload. Accessed 04 Feb 2020
9. Yara. https://virustotal.github.io/yara/. Accessed 04 Feb 2020
10. Malware statistics & trends report — av-test (2020). https://www.av-test.org/en/statistics/malware/. Accessed 20 Apr 2020
11. Aafer, Y., Du, W., Yin, H.: Droidapiminer: mining API-level features for robust malware detection in android. In: Security and Privacy in Communication Networks (SecureCom) (2013)
12. Abadi, M., et al.: Tensorflow: a system for large-scale machine learning. In: Symposium on Operating Systems Design and Implementation (OSDI) (2016)
13. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: collecting millions of android apps for the research community. In: Working Conference on Mining Software Repositories (MSR)
14. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Are your training datasets yet relevant? In: International Symposium on Engineering Secure Software and Systems (2015)

15. Amos, B., Turner, H., White, J.: Applying machine learning classifiers to dynamic android malware detection at scale. In: International Wireless Communications and Mobile Computing Conference (IWCMC) (2013)
16. Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K.: Drebin: Effective and explainable detection of android malware in your pocket. In: Annual Network and Distributed System Security Symposium (NDSS) (2014)
17. Bedford, A., et al.: Andrana: Quick and Accurate Malware Detection for Android (2017)
18. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. J. Mach. Learn. Res. **13**, 281–305 (2012)
19. Bergstra, J.S., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization (2011)
20. Bottou, L., Bousquet, O.: The tradeoffs of large scale learning (2008)
21. Chapelle, O., Vapnik, V., Bousquet, O., Mukherjee, S.: Choosing multiple parameters for support vector machines. arXiv preprint arXiv:1502.02127 (2002)
22. Chen, X., et al.: Android HIV: a study of repackaging malware for evading machine-learning detection. Trans. Inf. Forens. Secur. (TIFS) **15**, 987–1001 (2019)
23. Claesen, M., De Moor, B.: Hyperparameter search in machine learning. arXiv preprint arXiv:1502.02127 (2015)
24. Coffman Jr., E.G., Garey, M.R., Johnson, D.S.: An application of bin-packing to multiprocessor scheduling. SIAM J. Comput. **7**(1), 1–17 (1978)
25. Cortes, C., Jackel, L.D., Chiang, W.P.: Limits on learning machine accuracy imposed by data quality. In: Advances in Neural Information Processing Systems (NIPS) (1995)
26. Dai, S., Tongaonkar, A., Wang, X., Nucci, A., Song, D.: Networkprofiler: towards automatic fingerprinting of android apps. In: 2013 Proceedings IEEE INFOCOM (2013)
27. Enck, W., et al.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. Trans. Comput. Syst. (TOCS) **32**(2), 5 (2014)
28. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: semantics-based detection of android malware through static analysis. In: International Symposium on Foundations of Software Engineering (FSE) (2014)
29. Feurer, M., Eggensperger, K., Falkner, S., Lindauer, M., Hutter, F.: Practical automated machine learning for the AUTOML challenge 2018. In: International Workshop on Automatic Machine Learning at ICML (2018)
30. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J.T., Blum, M., Hutter, F.: Auto-sklearn: efficient and robust automated machine learning. In: Hutter, F., Kotthoff, L., Vanschoren, J. (eds.) Automated Machine Learning. TSSCML, pp. 113–134. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05318-5_6
31. Fogla, P., Sharif, M.I., Perdisci, R., Kolesnikov, O.M., Lee, W.: Polymorphic blending attacks. In: USENIX Security Symposium (2006)
32. Grace, M., Zhou, Y., Zhang, Q., Zou, H., Jiang, X.: Riskranker: scalable and accurate zero-day android malware detection. In: International Conference on Mobile Systems, Applications, and Services (MobiSys) (2012)
33. Hou, S., Saas, A., Chen, L., Ye, Y.: Deep4maldroid: a deep learning framework for android malware detection based on Linux kernel system call graphs. In: International Conference on Web Intelligence Workshops (WIW) (2016)
34. Kotthoff, L., Thornton, C., Hoos, H.H., Hutter, F., Leyton-Brown, K.: Auto-weka 2.0: automatic model selection and hyperparameter optimization in Weka. J. Mach. Learn. Res. **18**, 826–830 (2017)

35. Li, L., et al.: Static analysis of android apps: a systematic literature review. Inf. Software Technol. **88**, 67–95 (2017)
36. Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: a novel bandit-based approach to hyperparameter optimization. J. Mach. Learn. Res. **18**(1), 6765–6816 (2017)
37. Lin, S.W., Ying, K.C., Chen, S.C., Lee, Z.J.: Particle swarm optimization for parameter determination and feature selection of support vector machines. Expert Syst. Appl. **35**(4), 1817–1824 (2008)
38. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: Conference on Computer and Communications Security (CCS) (2012)
39. Mariconti, E., Onwuzurike, L., Andriotis, P., De Cristofaro, E., Ross, G., Stringhini, G.: Mamadroid: detecting android malware by building Markov chains of behavioral models. In: Annual Network and Distributed System Security Symposium (NDSS) (2017)
40. Miller, B., et al.: Reviewer integration and performance measurement for malware detection. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 122–141. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40667-1_7
41. Pedregosa, F., et al.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
42. Peiravian, N., Zhu, X.: Machine learning for android malware detection using permission and API calls. In: International Conference on Tools with Artificial Intelligence (2013)
43. Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L.: Tesseract: eliminating experimental bias in malware classification across space and time. In: USENIX Security Symposium (2019)
44. Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the Seventh European Workshop on System Security (2014)
45. Preda, M.D., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. ACM Trans. Program. Lang. Syst. (TOPLAS) **30**(5), 1–54 (2008)
46. Probst, P., Bischl, B., Boulesteix, A.L.: Tunability: importance of hyperparameters of machine learning algorithms. arXiv preprint arXiv:1802.09596 (2018)
47. Rasthofer, S., Arzt, S., Lovat, E., Bodden, E.: Droidforce: enforcing complex, data-centric, system-wide policies in android. In: International Conference on Availability, Reliability and Security (2014)
48. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: evaluating android anti-malware against transformation attacks. In: Symposium on Information, Computer and Communications Security - ASIA CCS 2013 (2013)
49. Saracino, A., Sgandurra, D., Dini, G., Martinelli, F.: Madam: effective and efficient behavior-based android malware detection and prevention. Trans. Depend. Secur. Comput. **15**(1), 83–97 (2016)
50. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.: "andromaly": a behavioral malware detection framework for android devices. J. Intell. Inf. Syst. **38**(1), 161–190 (2012)
51. Sun, M., Wei, T., Lui, J.C.: Taintart: a practical multi-level information-flow tracking system for android runtime. In: Conference on Computer and Communications Security (CCS) (2016)

52. Tsai, J.T., Chou, J.H., Liu, T.K.: Tuning the structure and parameters of a neural network by using hybrid Taguchi-genetic algorithm. Trans. Neural Netw. **17**(1), 69–80 (2006)
53. Zhang, M., Duan, Y., Yin, H., Zhao, Z.: Semantics-aware android malware classification using weighted contextual API dependency graphs. In: Conference on Computer and Communications Security (SIGSAC) (2014)