Automatically Combining Static Malware Detection Techniques

David De Lille, Bart Coppens
Computer Systems Lab
Ghent University, Belgium
bart.coppens@elis.ugent.be

Daan Raman
NVISO CVBA, Belgium
draman@nviso.be

Bjorn De Sutter
Computer Systems Lab
Ghent University, Belgium
bjorn.desutter@elis.ugent.be

Abstract

Malware detection techniques come in many different flavors, and cover different effectiveness and efficiency trade-offs. This paper evaluates a number of machine learning techniques to combine multiple static Android malware detection techniques using automatically constructed decision trees. We identify the best methods to construct the trees. We demonstrate that those trees classify sample apps better and faster than individual techniques alone.

1 Introduction

After having been a plague on mostly Windows-based computers for decades, malware is now also massively targeting Android-based devices. Consumers wanting to protect themselves against Android malware can to some extent rely on Google's malware detection as applied to all apps in the Google Play Store, i.e., the official Android app store. However, no such automated detection exists for non-official stores such as the Chinese stores Baidu App Store and Tencent App Gem. Furthermore, even users that use only the Google Play Store might want to have a second opinion on whether an app contains malware or not. It is therefore not surprising that a lot of anti-virus companies such as AVG, AVAST, Kaspersky, McAfee, BitDefender, etc., all provide malware scanners on the Google Play Android Store, and that a lot of malware detection techniques have been published in scientific literature [1, 6, 7, 10, 14].

To scan apps en masse, scanners need to be fast. Yet they also have to be accurate, and higher accuracy most often comes at the price of longer scanning times. As a result, the area of malware detection techniques is still very active. In turn, malware writers also deploy evolving detection evasion techniques. In this ever changing landscape, a recurring problem is to decide which malware detection techniques to use. In terms of effectiveness, a scanner should deploy most if not all of the available techniques, but in terms of efficiency, that would be detrimental. So scanner designers and users have to make a choice, based on a trade-off between effectiveness and efficiency. Apart from the raw effectiveness with which they classify samples into goodware and malware and their efficiency in terms of running times, different malware detection techniques have different strengths, in the sense that they identify some types of malware better than others. Moreover, some detection techniques are easier to evade than others.

This paper assesses a range of machine learning (ML) methods to automate this decision problem. We focus on automated methods to build decision trees (DTs) in which each path consists of a specific sequence of existing malware detection techniques. At every node, an existing scanning technique is applied to a sample, of which the outcome determines either the final classification of the sample as goodware or malware or the next technique to be applied.

Our design goals for generating the DTs are that they should be fast, and that they should be able to identify new kinds of malware. To keep the system fast, we include only static analysis techniques. In order to focus on detecting previously unknown malware variants, our system will *not* include any technique relying on fingerprints of known malware. However, we do use techniques that try to detect unknown variants of known malware. Finally, we only implemented publicly known techniques that we were able to re-use as is, or that we could easily reimplement.

Our contributions are the the following:

1. We evaluate five methods to construct DTs of static malware detection techniques, and identify a clear winner to supporting a user or designer in trading off effectiveness vs. efficiency.

2. We compare the effectiveness of those DTs to the effectiveness of a range of other supervised ML techniques that do not allow to trade off effectiveness for efficiency.

3. We demonstrate that combinations of detection techniques provide better and faster results than individual techniques alone.

4. We demonstrate that trivial heuristics that lack the required accuracy to be used as stand-alone scanners, are in fact very useful when combined with existing techniques to minimize the average scanning time.
5. We propose adaptations to some existing techniques to make them suitable for inclusion in DTs.

Section 2 describes background about DTs and ML. Next, Section 3 describes our experimental set-up, including the used data sets and detection algorithms. Section 4 then presents results, and Section 5 draws conclusions.

2 Decision Trees and Machine Learning

We want to combine existing malware detection techniques. Our ultimate goal is to have better classification than what is achievable by the individual techniques, at a lower cost than applying all techniques together. Specifically, we want to use ML techniques to generate an optimized decision process that selects and applies a subset of the existing techniques in an order that optimizes both the efficiency, i.e, the computation time, and the effectiveness, i.e., the classification result, of the detection when applied to new samples.

For the classification decision process, we will use DTs, in which each leaf node corresponds to a final label with which a tested sample is labeled. In this paper, the samples are apps, and the labels are "malware" and "goodware".

The internal (non-leaf) nodes of DTs model a flow-chart process, in which each node corresponds to an attribute being evaluated and each branch represents the outcome of an evaluation. The paths from the root to the leafs represent the decision process, which in this case corresponds to the set of classification rules. In the context of this paper, the sample attributes evaluated in the internal nodes are the samples' classification into goodware or malware by the existing, individual malware detection techniques. Each sample evaluation in an internal node has an associated cost: the computation time required by the malware detection technique to analyze the app. Both the efficiency and the effectiveness of a DT hence depend on its internal structure, i.e., on how the existing techniques are combined.

To generate an optimal DT structure with which we can classify new samples efficiently and effectively, we evaluated multiple ML approaches based on supervised learning. These approaches start from a so-called training set of samples that are already labeled and of which all attributes are already computed. In other words, in our context they will start from a set of apps labeled as goodware or malware, and to which all the existing techniques have been applied during the so-called data collection phase of the approach.

In the training phase, the DT is then constructed based on the collected data. Initially, one attribute is chosen for the whole training data set, using some metric. The training set is then partitioned into subsets based on the value of this attribute. On each subset, the same partitioning algorithm is then applied recursively, obviously considering other attributes for each sub step.

The algorithm stops when all the samples in a subset belong to the same class, in which case the subset as a whole can be labeled accordingly. Alternatively, the iterative partitioning into subsets stops when all the samples in a set have identical values for all attributes. In that case, the set of considered attributes has proven to be inconclusive. Such a final subset is then assigned the class label that occurs most often among the samples in the subset. This serves the goal of minimizing the number of incorrect classifications following from the inconclusiveness.

The approaches proposed in literature and evaluated in our research differ in the metric they use for choosing an attribute for each partitioning step. For example, the most popular DT algorithm is called ID3 [15]. The selected attribute for each partition is chosen to be the attribute that contributes the most information about the class when it is examined. It does so by calculating the *information gain* for each attribute, which is based on the concept of entropy:

$$\text{Frequency of class } y \text{ in data set } T : p(y) = \frac{|(T|y)|}{|T|}$$

$$\text{Entropy of data set } T : H(T) = -\sum_{y \in Y} p(y) \cdot log_2(p(y))$$

$$\text{Subset after choosing attribute } A : T = \bigcup_{s \in S} s$$

$$\text{Information Gain: } IG(A,T) = H(T) - \sum_{s \in S} H(s) \cdot \frac{|s|}{|T|}$$

Other metrics for generating DTs sometimes include a cost for using an attribute. With this cost, we can make a trade-off between the information gain and the cost of evaluating an attribute. The alternative proposals from literature as listed below differ in how they combine information gain and cost into a metric for choosing the best attribute:

ID3 [15]:	*information gain*
EG2 [13]:	$(2^{(\textit{information gain})} - 1)/(\textit{attribute cost} + 1)^{\omega}$
CS-ID3 [17]:	$(\textit{information gain})^2/\textit{attribute cost}$
IDX [12]:	$\textit{information gain}/\textit{attribute cost}$
CSGain [5]:	$(N_a/N) \cdot \textit{information gain} - \omega \cdot \textit{attribute cost}$

The parameter ω, used for EG2 and CSGain, can be adjusted by the user to indicate the relative importance of information gain and cost. N is the amount of samples in the entire training set and N_a is the amount of samples in the current data set for which an attribute has to be selected.

Moreover, to avoid that the learning approach overfits the DT onto the training set, we imposed two additional conditions before a data set is partitioned: first of all, the number of samples in the data set has to be high enough, and secondly, a dataset can not be partitioned if the information gain of the chosen attribute is too small. DTs are only one form of classification techniques that can be trained using supervised learning methods. We also evaluated some other techniques. In particular, these are k-nearest neighbor (NN) [4], naive Bayesian networks [9], neural networks [16], support vector machines [3] and linear discriminant analysis [11], all of which are readily available in the ML software package RapidMiner Studio v6.

These additional techniques do not consider cost metrics. This is the case because for classifying a new sample app as malware or goodware once the training has been finalized, these techniques first require the deployment of *all* of the individual detection techniques on the app, whereas the DT approach will only have to apply the techniques on the path followed during the classification process.

3 Experimental Setup

This section discusses the data collection and the malware detection techniques that the ML algorithms will use as input. Seven techniques are taken from literature, and we designed three basic additional techniques. Both our adaptations to the existing techniques, and our additional heuristics are discussed, as is our experimental methodology.

3.1 Data Set

The training data set for this study consists of 1858 Android goodware apps and 1598 Android malware apps. The goodware samples were collected from the top 60 apps in each of the 34 categories of the Google Play store using a service called APK_Downloader. We collected the malware samples from two different sources: from the Android Malware Genome Project [19], and from the Contagio Mobile blog. We excluded 39 samples from our original data set for two reasons. First, we excluded some samples because they could not be handled by all the detection tools we received from their authors. Secondly, we excluded some supposed malware samples because the VirusTotal online malware scanner did not classify them as malware.

For validation purposes, it is important to establish a correct ground truth, i.e., the correct label is assigned to each sample app. This is not a problem for the malware samples, because these apps have already been confirmed to contain malicious code by malware analysts. In theory it is possible, however, for our goodware apps to also contain malware. In practice, this is highly unlikely for two reasons. First, as these apps were in the top-60 of a certain app category, they have been downloaded and tested by a lot of people. Secondly, the Google Play store makes use of the Google Bouncer service that scans all apps for malware and applies dynamic malware detection techniques.

3.2 Existing Detection Techniques

For our research, we selected existing techniques of which we could receive implementations from their authors or about which enough documentation was available to re-implement the technique confidently. Furthermore, we considered the orthogonality of the approaches with respect to the types of features they use for their classification.

Adagio Adagio uses a predictive model based on Support Vector Machines [10]. To build the training data, Adagio first constructs the call graph (CG) of each app. Every node in the CG represents a procedure and gets assigned a hash value that describes the properties of the procedure's neighbors in the CG. Then the number of times each hash value is present in the graph is counted. These values are transformed into a vector that characterizes the app.

The vectors of the training set are used to create a predictive model to classify new apps. Gascon et al. report a precision of 89% and a false positive rate of 10% for this technique as applied on 135792 goodware samples and 12158 malware samples [10]. We obtained the Python implementation of this technique from them.

API-1NN Based on DroidAPIMiner by Aafer et al., API-1NN is a nearest neighbor approach that classifies apps based on the API calls detected in them [1]. For training, the signatures of called methods in the Android API are extracted from the apps. The original DroidAPIMiner numbers all those API calls. A bit vector then represents which API calls are present in a sample. After training, a similar bit vector is computed on a new app to classify, and the nearest neighbor approach is applied to this bit vector.

We did not obtain the original implementation by the original authors, and hence had to re-implement it. Moreover, since one of our requirements is that the creation of DTs can be redone in an evolving landscape of malware techniques and anti-malware techniques, we cannot assume the existence of one fixed mapping of API calls to numbers. Therefore we adapted the method as follows: Each API call is then hashed and the remainder of this hash value mod N, with a given integer N, is computed. A bit vector of size N is then constructed, in which each bit in position n, with $0 \leq n < N$, indicates whether or not an API that was hashed into the value n was present in the app.

For our own implementation, we experimentally determined the appropriate value for N to be 1500. Whereas the original authors reported a precision of 99% for their technique, our re-implementation was only able to achieve 95.5%. This might be due to the different training set, or because of undocumented details of the original implementation that we could not re-implement.

Risk Scoring Schemes Peng et al. described four related techniques that try to quantify the *risk* of Android apps [14]. Rather than presenting the user with a set of permissions requested by an app, they propose a single risk score, that indicates to users the risk involved in installing and using the app. They use probabilistic generative models based on naive Bayes classifiers. As attributes, they consider the category of an app, and the permissions the app requests in its so-called manifest, the idea being that permissions requested by one app, but not typically requested in its category, makes an app suspicious. In order of increasing complexity, the proposed variations proposed by Peng et al. are:

1. Basic Naive Bayes (BNB) only considers permissions.
2. Naive Bayes with Informative Priors (PNB) in addition builds on the a priori knowledge that some permissions are more dangerous than others.
3. Mixture Naive Bayes with Categories (MNBC) considers the category of an app, besides its permissions;
4. Hierarchical Mixture of Naive Bayes (HMNB) considers a uniform partitioning of categories into subclasses.

All four variations performed more or less equally well, with HMNB performing slightly better. We obtained a C++ implementation from the original authors, who reported that the MNBC implementation was problematic. We thus did not use that variation in our experiments.

It is important to note that unlike the previous techniques, these risk scoring schemes do not assign class labels (goodware or malware). Instead, the produced risk scores are numbers that indicate how dangerous the app might be. These scores are converted to the desired binary class output using a threshold. The optimal threshold was determined by first calculating the risk scores of the training samples and finding the threshold that best classified these samples.

DroidLegacy DroidLegacy is a technique introduced by Deshotels et al. It is designed to detect repackaged malware [6]. This is a type of malware that is created by taking a goodware app and adding malicious code to it. When two apps contain very similar malicious code, they are said to belong to the same malware family.

This technique groups the classes of an app into modules that are characterized by their API calls. By comparing training samples of the same malware family, the recurring malicious module can be identified and this will act as a signature for the malware family. Then, every test sample is compared to every signature and a value between 0 and 1 indicates how well the sample fits the signature. The score is then converted to a class prediction using a threshold, similar to the Risk Scoring schemes.

The original authors gave us access to their source code, which required little adaptations for our experimental setup. However, we did discover that in our training set, a lot of false positive results were reported, due to some used signatures being too generic, and hence spanning significantly more than a single malware family. To improve the performance of this technique, we manually searched for the signatures causing the false positives, and excluded them.

Kirin Kirin is one of the oldest techniques for detecting Android malware [7]. Its goal is to inform a user when certain dangerous combinations of permissions are requested by an app. Kirin has seven combinations of permissions that are not allowed. If one of these seven rules is triggered, the sample is considered to be malicious.

While the original code of Kirin is available, we reimplemented it: the original code is designed to prompt a user of an Android device on installing an app, whereas we want our detection techniques to run on a non-Android server.

3.3 Additional Detection Heuristics

In addition to reusing the aforementioned existing techniques, we developed three new heuristics. These new heuristics primarily focus on efficient ways to identify highly likely goodware and highly likely malware, albeit at the cost of many false negatives. As we only intent to combine the heuristics with other techniques, their high false rates are not necessarily problematic.

Permissions Decision Tree Our first heuristic uses a DT to classify goodware and malware based on the permissions requested by an app. In addition to this extremely simple classification, we created an additional *Permissions-DT-Enhanced* technique, that in addition to the app's permissions also takes into account the invocation or incorporation of encryption APIs, dynamic loading, reflection, and native code. We generate the DT based on the training data.

Contains APK In addition to the above permission DTs, we include a simple check for whether or not the Android application package (APK), contains another APK file itself. We did so because there are certain malware apps that don't appear to be dangerous at all, but that do have an APK file embedded in them that contains the malicious code. By itself, this technique has a very low accuracy (only 64.92%), but it has a very strong positive predictive value (95.48%), which is higher than that of all the existing techniques, and it has an extremely low cost. That is why we added it to our suite of possible detection "techniques".

Has Permissions Our final heuristic checks whether an app happens to request no permissions at all. As soon as an app requests a permission, the predicted class will be malware. This provides an extremely fast way to conclude that an app probably is goodware. After all, an app with no permissions is extremely limited in what it can abuse.

3.4 Methodology

By means of k-fold cross-validation, we can prevent the ML algorithms from over-fitting the malware samples in the data set, while at the same time still making use of *all* the malware samples in the data set to train and evaluate our framework [2]. With this technique, the data set is randomly partitioned in k sets. In each of k experiments, one of these sets is used as a validation set, while the other $k-1$ sets are used as training inputs for the ML algorithms. In other words, we repeat the experiments of constructing a DT k times, and evaluate each DT on a validation data set of which no samples were used for the DT's construction. We chose to use 10-fold, and report the average cost and accuracy of the 10 evaluations.

Some of the individual detection techniques first also need to be trained by means of data sets. To train those techniques on each of the training sets for the global 10-fold cross-validation, we used 3-fold cross-validation.

We already mentioned in Section 2 that there are two parameters to consider for generating a DT: the minimal number of samples in a data set before that set can be split, and the minimal information gain for an attribute of a data set before that data set can be split. In addition, some DT algorithms have an additional parameter ω. This parametrization allows us to generate many different DTs using each of the evaluated ML algorithm, with each DT being characterized by its efficiency (i.e., cost) in terms of average execution time, and by its effectiveness in term of achieved classification accuracy. As each of the evaluated algorithms are fast enough to generate thousands of DTs per minute (given our data set sizes), we performed a parameter sweep, and report the cost/accuracy Pareto front.

All our measurements were executed on an 32-bit Intel Core 2 Duo system clocked at 2GHz, with 6GiB of RAM. To speed up our ML experiments, we evaluated all techniques once on each app, recording the results and execution time of the analysis. The average execution time of the technique is used as the cost for generating the DTs. The execution time of a generated DT is then the average time required to analyze an app from the validation set.

We evaluate and rank the different algorithms based on standard metrics for evaluating binary classifications [8, 18]. These metrics assess how well the labels assigned to the samples by a DT match the ground truth. A sample that is correctly identified as malware is a True Positive (TP), a sample that is correctly identified as goodware is a True Negative (TN). A malware sample that is mis-identified as goodware is a False Negative (FN), whereas goodware that is mis-identified as malware is a False Positive (FP). The *sensitivity* is defined as $TP/(TP+FN)$, and is a measure for how well the classification correctly identifies malware. The *specificity* is $TN/(TN+FP)$, and is a measure for how well the classification correctly identifies goodware. Finally, these measures can be combined into a single measure called the *accuracy*, which is defined as $(TP+TN)/(TP+FN+TN+FP)$.

Technique	Cost	Acc.	Sens.	Spec.
Adagio	13.4s	94.8%	97.5%	92.4%
API-1NN	6.7s	95.4%	96.4%	94.5%
Risk Scoring: BNB	0.35s	77.5%	77.1%	77.8%
Risk Scoring: PNB	0.35s	77.2%	79.9%	77.5%
Risk Scoring: HMNB	0.35s	79.3%	78.4%	80.1%
Contains APK	0.44s	64.6%	24.6%	98.9%
DroidLegacy	137s	75.7%	71.8%	79.1%
Kirin	0.35s	68.8%	47.6%	87.5%
Has Permissions	0.35s	47.2%	99.3%	2.4%
Perm.-DT	0.53s	93.1%	91.3%	94.7%
Perm.-DT-Enhanc.	8.2s	94.1%	94.1%	94.2%

Table 1. Results of the individual techniques

4 Results

We start by investigating the results of the individual detection techniques. Table 1 shows the aggregated results. The first thing to note is the vast range of execution time per technique: the fastest techniques take less only about 1/3 of second. These are the techniques that essentially only open the APK file to inspect the permissions, such as the *Kirin* and *Has Permissions* techniques. At the opposite end are the techniques that require extraction of significantly more information from the APK, such as the used API calls in the case of *API-1NN*, and information about the call graphs in the case of *Adagio*.

Furthermore, note that the slow techniques do not necessarily perform better than the faster techniques on our data set. For example, while DroidLegacy is the slowest technique, it has a lower accuracy than many of the techniques that take less than a second to execute. However, in this class of techniques that on average take less than a second to execute, there is still significant variation in the accuracy. For example, the simple *Has Permissions* check performs significantly worse than *Kirin*, even though both techniques are equally fast. However, note that the *Has Permissions* check is especially good at correctly identifying goodware, as can be seen from its sensitivity of over 99%.

Next, we investigated combining these individual techniques by using DTs. Figure 1 shows the Pareto fronts of the different DT construction algorithms. Please note that per parameter setting, 10 DTs are generated for the 10 cross-validation experiments. Points on the Pareto fronts represent average costs and accuracies over those 10 DTs. Given the high accuracy of fast techniques such as *Permissions*

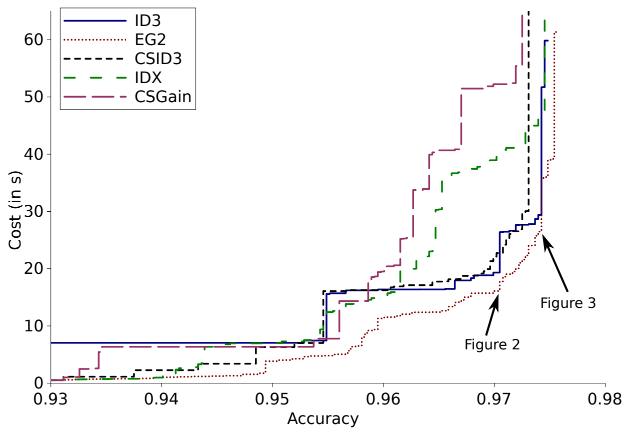

Figure 1. Pareto fronts of cost vs. accuracy of DTs generated with different strategies.

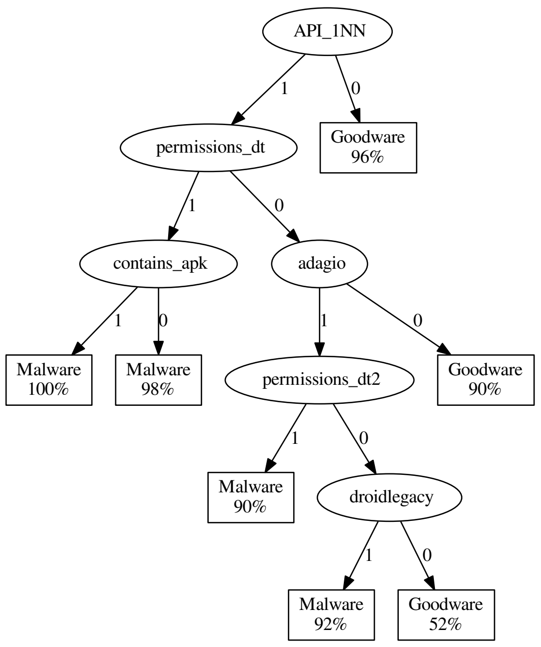

Figure 2. DT with an accuracy of 97.0% and an average execution time of 16.1s.

DT, the graph only show results for DTs that deliver an average accuracy of more than 93%.

Irrespective of the trade-off that one wants to make between cost and accuracy, the EG2 strategy consistently delivers the best DTs. Only when the fastest DTs are considered, at the cost of significantly lower accuracy, do the DTs generated using the IDX algorithm come close to the performance of EG2-generated DTs. This consistent result of EG2 is somewhat surprising, as in other domains such as forensics classification, more recent algorithms parametrizable with the same ω performed much better.

Another interesting result is that the choice of algorithm significantly influences the average analysis time. For example, DTs built with EG2 can achieve an accuracy of at least 97% with an average analysis time of less than 20 seconds, whereas using the CSGain algorithm results in DTs that on average require more than 50 seconds to get the same accuracy. So EG2 not only consistently delivers better DTs, it delivers much better ones.

Whereas the best performance of the individual techniques is 95.3%, using DTs we can achieve a performance of 97.6%, with an execution time of only 61 seconds. In practice, however, we would choose the DTs generated with EG2 that achieve slightly less than 97.5%, but have an execution time of less than 30 seconds, which still have a significantly higher accuracy than the individual techniques.

The simplest DTs correspond to running a single detection technique and presenting that result to the user. For instance, the DT with an accuracy of 93.1% corresponds to a single check of the *Permissions DT* analysis. More interesting behavior shows up once the DTs combine different analysis techniques. Figure 2 shows one of the DTs generated using the EG2 algorithm that correspond to the Pareto-optimal point with an accuracy of of 97.0% and an average execution time of 16.1 seconds. This point is labeled in the Pareto curves of Figure 1. Because we use 10-fold cross-validation, this is only one out of 10 generated DTs for that Pareto point. In each leaf node of the DT, the label given to a simple is depicted, as well as the *confidence* that the label is correct. This confidence equals the number of all training samples that were labeled correctly by the leaf node, divided by the total number of samples labeled by it.

We see that a single negative check of the *API-1NN* technique is sufficient to classify a sample as malware. However, to improve the accuracy, the DT tries to minimize the impact of *API-1NN*'s false positives by running more analyses when *API-1NN* suggests that the sample is malware.

Note that this automatically generated DT can still be improved (manually) when the user is only interested in the final classification, but not in the confidence of it: Regardless of the result of the *Contains APK* test, the produced label is *Malware*. In other words, even though the test requires some execution time, it has no impact on the classification whatsoever. So we could simply remove it to obtain an even faster DT, with the same accuracy.

This test was included in the DT is because of the lack of lookahead in the greedy EG2 construction algorithms: While *Contains APK* was the best technique to split the data set at some point during the iterative construction of the DT, the chosen parameters prevented the remaining sets from being split further. As both sets contain more malware than goodware, both are labeled as malware.

Technique	Accuracy
1-NN	96.66%
2-NN	96.95%
3-NN	97.24%
4-NN	97.05%
5-NN	97.28%
Naive Bayes	95.85%
Neural Network	97.36%
Support Vector Machine	95.59%
Linear Discriminant Analysis	96.66%

Table 2. Results of the other ML techniques

To achieve more accurate decisions, more complex DTs need to be generated. Figure 3 shows one of the DTs for an accuracy of 97.4% and an average execution time of 26.5 seconds. Again, this point is labeled in the Pareto curves of Figure 1. The top part of this DT is relatively balanced in the number of techniques that need to be applied for each sample. Only the bottom right stands out with a rather narrow subtree, of which only a single leaf node represents malware. This subtree furthermore includes a decision based on *DroidLegacy*, the slowest individual technique. In this case, the advantages of DTs are obvious: Even though it is by far the slowest malware detection technique, it is still useful in this case because not only can it usefully contribute to an accurate classification, the induced average overhead is relatively small, due to the fact that this subtree contains only 5% of the samples.

Table 2 shows the accuracies obtained with alternative ML techniques. For the parameterizable techniques, we chose the default values as they are set in the RapidMiner ML framework. Do note that, because these algorithms do not take into account the cost and because they require all of the individual detection techniques to be deployed on a sample in order to classify the sample, the average execution time of all of these techniques is the sum of the average times of *all* individual detection techniques, which is about 170 seconds in our setup. Even though the DTs use only a subset of the detection techniques to base their classification on, they clearly outperform the other ML techniques.

Because of our use of only default parameters, it is possible that a different combination of parameters would allow these alternative classification techniques to get a higher accuracy. Even so, the total analysis time per sample would still be more than 6 times higher than that of the DT shown in Figure 3.

5 Conclusions

We have demonstrated that DTs provide an excellent strategy to combine different static malware detection techniques into a single framework that is optimized for both a high classification accuracy, while keeping the analysis time low. In particular, we have shown that by using the EG2 algorithm to construct the decision trees, malware can be classified significantly better than could be done using the individual detection techniques, without needing to pay the overhead of executing all detection techniques. Furthermore, we show that these DTs can outperform other machine learning techniques that combine *all* individual malware detection techniques.

References

[1] Aafer, Y., Du, W., and Yin, H. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *Security and Privacy in Communication Networks*, vol. 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer International Publishing, 2013, pp. 86–103.

[2] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] Cortes, C., and Vapnik, V. Support-vector networks. *Machine learning 20*, 3 (1995), 273–297.

[4] Cover, T., and Hart, P. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theor. 13*, 1 (Jan. 1967), 21–27.

[5] Davis, J., Ha, J., Rossbach, C., Ramadan, H., and Witchel, E. Cost-sensitive decision tree learning for forensic classification. In *Machine Learning: ECML 2006*, vol. 4212 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 622–629.

[6] Deshotels, L., Notani, V., and Lakhotia, A. Droidlegacy: Automated familial classification of Android malware. In *Proc. of ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)* (2014), pp. 3:1–3:12.

[7] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proc. 16th ACM Conf. on Computer and Communications Security (CCS)* (2009), pp. 235–245.

[8] Fawcett, T. An introduction to ROC analysis. *Pattern Recognition Letters 27*, 8 (June 2006), 861–874.

[9] Friedman, N., and Goldszmidt, M. Building classifiers using bayesian networks. In *Proc. 13th National Conf. on Artificial Intelligence (AAAI)* (1996), pp. 1277–1284.

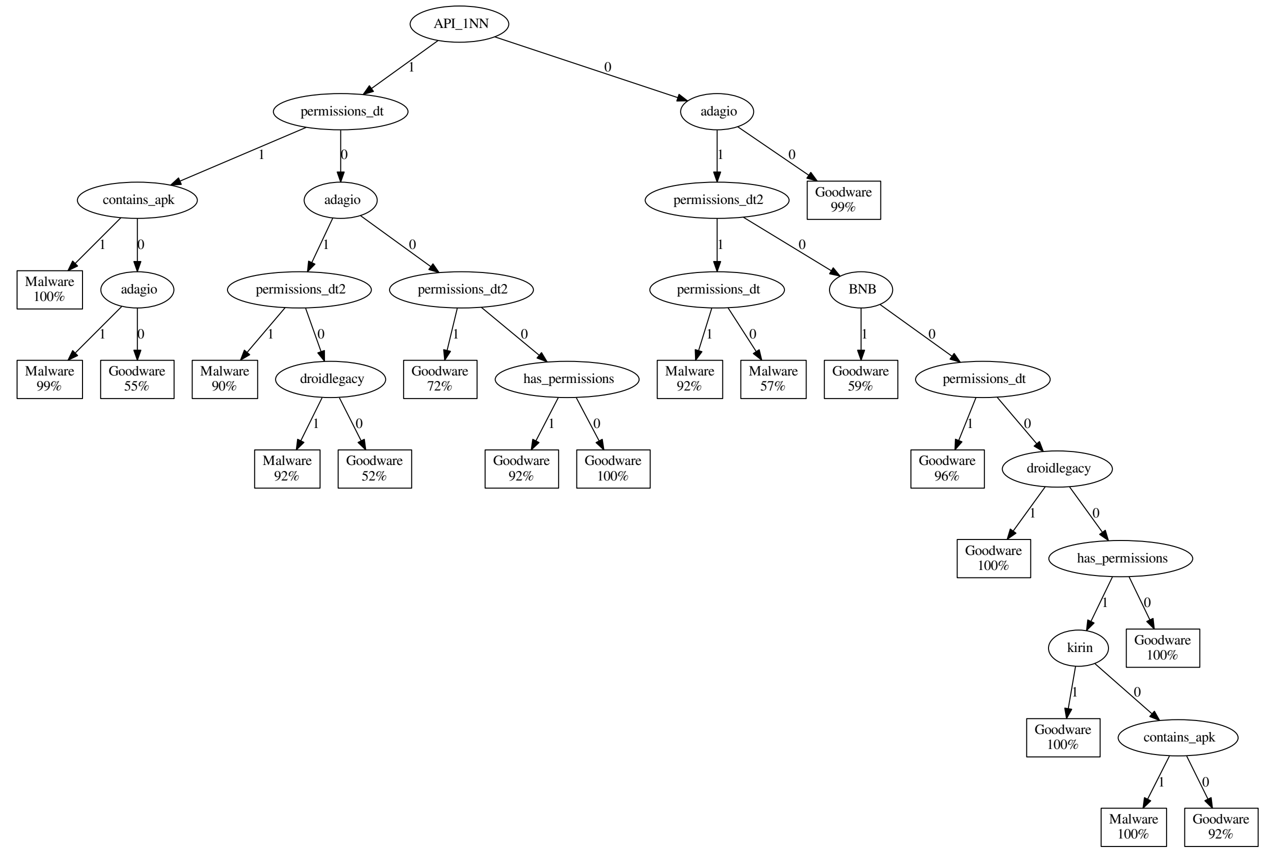

Figure 3. DT with an accuracy of 97.4% and an average execution time of 26.5s.

[10] GASCON, H., YAMAGUCHI, F., ARP, D., AND RIECK, K. Structural detection of Android malware using embedded call graphs. In *Proc. ACM Workshop on Artificial Intelligence and Security (AISec)* (2013), pp. 45–54.

[11] LACHENBRUCH, P. A. *Discriminant analysis*. Wiley Online Library, 1975.

[12] NORTON, S. W. Generating better decision trees. In *Proc. 11th Int'l Joint Conf. on Artificial Intelligence (IJCAI)* (1989), pp. 800–805.

[13] NEZ, M. The use of background knowledge in decision tree induction. *Machine Learning 6*, 3 (1991), 231–250.

[14] PENG, H., GATES, C., SARMA, B., LI, N., QI, Y., POTHARAJU, R., NITA-ROTARU, C., AND MOLLOY, I. Using probabilistic generative models for ranking risks of Android apps. In *Proc. ACM Conf. on Computer and Communications Security (CCS)* (2012), pp. 241–252.

[15] QUINLAN, J. Induction of decision trees. *Machine Learning 1*, 1 (1986), 81–106.

[16] RUMELHART, D. E., HINTON, G. E., AND WILLIAMS, R. J. *Neurocomputing: Foundations of Research*. MIT Press, 1988, ch. Learning Representations by Back-propagating Errors, pp. 696–699.

[17] TAN, M. Cost-sensitive learning of classification knowledge and its applications in robotics. *Machine Learning 13*, 1 (1993), 7–33.

[18] WITTEN, I. H., FRANK, E., AND HALL, M. A. *Data Mining: Practical Machine Learning Tools and Techniques*, third ed. Morgan Kaufmann, 2011.

[19] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *IEEE Symp. on Security and Privacy (S&P)* (2012), pp. 95–109.