applied sciences

Article

Comparing Deep Learning and Shallow Learning Techniques for API Calls Malware Prediction: A Study

Angelo Cannarile [1], Vincenzo Dentamaro [2], Stefano Galantucci [2,*], Andrea Iannacone [1], Donato Impedovo [2] and Giuseppe Pirlo [2]

[1] BVTech SpA, 20123 Milano, Italy; a.cannarile@bv-tech.it (A.C.); a.iannacone@bv-tech.it (A.I.)
[2] Department of Computer Science, University of Bari "Aldo Moro", 70125 Bari, Italy; vincenzo.dentamaro@uniba.it (V.D.); donato.impedovo@uniba.it (D.I.); giuseppe.pirlo@uniba.it (G.P.)
* Correspondence: stefano.galantucci@uniba.it

Abstract: Recognition of malware is critical in cybersecurity as it allows for avoiding execution and the downloading of malware. One of the possible approaches is to analyze the executable's Application Programming Interface (API) calls, which can be done using tools that work in sandboxes, such as Cuckoo or CAPEv2. This chain of calls can then be used to classify if the considered file is benign or malware. This work aims to compare six modern shallow learning and deep learning techniques based on tabular data, using two datasets of API calls containing malware and goodware, where the corresponding chain of API calls is expressed for each instance. The results show the quality of shallow learning approaches based on tree ensembles, such as CatBoost, both in terms of F1-macro score and Area Under the ROC curve (AUC ROC), and training time, making them optimal for making inferences on Edge AI solutions. The results are then analyzed with the explainable AI SHAP technique, identifying the API calls that most influence the process, i.e., those that are particularly afferent to malware and goodware.

Keywords: CAPEv2; cuckoo; machine learning; classification; malware prediction; API calls; explainable AI; SHAP

check for updates

Citation: Cannarile, A.; Dentamaro, V.; Galantucci, S.; Iannacone, A.; Impedovo, D.; Pirlo, G. Comparing Deep Learning and Shallow Learning Techniques for API Calls Malware Prediction: A Study. *Appl. Sci.* **2022**, *12*, 1645. https://doi.org/10.3390/app12031645

Academic Editor: Federico Divina

Received: 30 December 2021
Accepted: 31 January 2022
Published: 4 February 2022

1. Introduction

The protection of a computer system, especially in a smart enterprise, represents a key factor for the company's survival: the damage caused by computer attacks can have significant economic impacts. Among the possible attacks, malware attacks are particularly relevant, as they cause direct damage to the system or intercept relevant information for the company. Protecting the smart enterprise from malware is crucial, so it is necessary to implement techniques that can detect and recognize malware within enterprise networks. Most malware detection systems are based on signature verification techniques, which are effective for all known malicious software, but inadequate for new malware as there are no signatures available. This limitation represents a problem, as it can intercept attacks built with prefabricated elements, but it cannot protect against a specifically constructed attack. A way to overcome the limitations of such approaches is to exploit the sandbox software CAPEv2 [1], the evolution of Cuckoo [2], to perform an analysis of hypothetical threats. The use of machine learning and classification techniques, operating on the information related to the behavior of executables, allows the detection of malware, especially the 0-day ones, i.e., those not yet known.

Edge AI (Edge computing + Artificial intelligence) solutions allow inference to be made at the direct generation of the data, even before it is downloaded to the receiver's machine. Such approaches have the advantage of generally working in a near real-time mode, but, on the other hand, they are constrained by limits of memory availability and computational power, thus making execution time a key factor. For these reasons, it is essential to understand which solutions are best suited to run on edge computing devices and then create Edge AI solutions that capture malicious events with high accuracy while maintaining low complexity and execution time constraints. A possible compromise in the view of such solutions is to extract the calls to Application Programming Interfaces (APIs) made by individual executables, thus obtaining an abstraction concerning the behavior of the programs. This approximation allows a summary evaluation of the behavior, which can be analyzed in terms of machine learning, to estimate if the software is malicious or goodware.

This paper presents a benchmark whose primary purpose is to test various machine learning algorithms, using deep learning and shallow learning techniques, applied on datasets of goodware/malware labeled software API calls.

The following tree-based shallow learning algorithms were considered:

- Random Forest;
- CatBoost;
- XGBoost;
- ExtraTrees;

and also two other deep learning algorithms based on neural networks:

- TabNet;
- NODE (Neural Oblivious Decision Ensembles).

These algorithms were tested on two different datasets: *malware-analysis-datasets-api-call-sequences* [3,4] and *APIMDS* [5] in balanced K-Fold cross-validation mode. Results' reports were then generated and analyzed in detail. The explainable artificial intelligence technique SHAP [6] also allowed for determining the extent to which each API call contributes both in discriminating goodware from malware, and vice versa.

The main contributions of this work are:

- Provide a benchmark showing the trade-off between the proposed algorithms based on accuracy and execution time;
- Use the Explainable AI SHAP technique to determine which API calls are heavily influential in the classification process;
- Show how the use of such approaches can support the results of dynamic analyses, focusing on the importance of individual API calls.

The paper is organized as follows: Section 2 contains the state-of-the-art review; Section 3 contains a description of the datasets used in the experiment; Section 4 describes the algorithms used, the settings of each of them for the experiment, and the techniques applied for preprocessing the dataset; Section 5 provides the results and their discussion. Section 6 presents the reasoning about explainable artificial intelligence using SHAP. Conclusions are presented in Section 7.

2. Related Work

Some of the scientific literature refers to Cuckoo [2], a precursor to CAPEv2; therefore, the principles described in these works are directly applicable.

The Honeynet Project published an open-source project called CuckooML [7] in 2016. According to the state of the art at the time, the researchers built an innovative feature based on anomaly detection techniques so that the modified software could cluster and identify new types of malware. All are available both in the command line and from a simplified web interface. The approach chosen by the authors was an unsupervised, descriptive type, specifically Clustering. One of the advantages, given the nature of the problem, is the lack of need for labeled data.

Darshan et al. [8] propose a new solution towards signature-based malware detection systems, where new malware, also obtained by applying obfuscation techniques on previously known malware, manage to easily evade controls, thus overcoming the defenses of a system. Thus, to detect malware based on their behavior, it starts by using the framework provided by Cuckoo, which offers an isolated environment where malicious software can be freely executed and its actions analyzed. Most sandboxes focus on system calls, i.e., those mechanisms used by a process at the application layer to request services at the operating system layer to perform desired user activities such as reading data, placing packets on the network, writing records to the registry, and so on.

The proposed approach for malware recognition is innovative: starting from executing the hypothetical threat inside Cuckoo, once a report in JSON format containing the system calls is obtained, this report is first converted into a new format, i.e., MIST (Malware instruction set). After that, the system calls contained in MIST, once isolated, are used to create N-grams that is a sub-sequence of N characters where N represents the length of each sequence. Through the metric of information gain, which represents the expected reduction of entropy in the data, the starting dataset (composed of N-grams) is partitioned, selecting the top N sub-sequences of characters as a Final Feature Vector, i.e., a vector of features to be processed in the training phase of the classifier. Unlike the paper presented by The Honeynet Project, in [8], a supervised approach and a classification task are proposed, and hence predictive, with the need to have a labeled dataset. The n-grams thus extracted belong to both malicious and genuine software; moreover, in case of duplicates, these are removed. Once the final features are obtained, an element called a instruction converter takes care of their transformation into the ARFF (Attribute Relation File Format), used by the WEKA framework to operate with different machine learning algorithms for classification. The authors selected, within WEKA, six different classification algorithms, respectively:

- Bayesian-Logistic-Regression;
- SPegasos;
- IB1;
- Bagging;
- Part;
- J48.

At the end of the runs, the best classifier was SPegasos, in terms of accuracy, lowest false positive rate, and highest true positive rate.

Ali et al. [9] used n-grams and TF-IDF for performing dynamic analysis for detecting malware using API calls, and the work reached an accuracy of 98.4% using this kind of preprocessing with Logistic Regression classifier. Kumar et al. [10], based on the anomaly detection technique, propose an innovative malware detection methodology based on clustering and Trend Micro Locality Sensitive Hashing (TLSH) metrics in order to cluster the dataset entries. In summary, starting from the report generated through the Cuckoo sandbox, a hash value is calculated according to the Trend Micro Locality Hashing metric. After that, the specific difference for each generated hash is calculated, and different clusters are generated based on a certain threshold. Next, the feature extraction and feature selection steps are applied, according to which the starting dataset is partitioned and prepared for the training phase. Finally, several classifiers for malware detection were trained through the `scikit-learn` library of Python. In addition, in this work, one of the crucial points lies in the dataset, since it is difficult to find suitable ones in public form and available on the net for this purpose: the authors had to build one specifically in order to evaluate the proposed methodology, which includes both malware and benign software. Although the task is supervised (prediction), in some preliminary training phases, i.e., feature extraction first and feature selection later, an unsupervised approach, clustering, was used to describe and group data according to TLSH metrics. This innovation is intended to overcome traditional feature extraction techniques, which were particularly unsuitable for large volumes of data. In fact, by applying this metric to clustering, a decrease in the training time of machine learning algorithms was observed without affecting the quality of predictions. Therefore, once Cuckoo obtains the report in JSON format, this report is used to calculate the footprint according to the TLSH metric. Based on these imprints, clusters are generated to group malware having similar traits. Several binary classification algorithms, i.e., genuine - malicious form, have been used for malware detection, such as:

- Decision Tree;

- Random Forest;
- Logistic Regression.

At the end of the execution, during the evaluation phase of the proposed model, it was found that the Random Forest algorithm obtained the best performance in terms of accuracy, lower false-positive rate, and higher true-positive rate.

Udayakumar et al. [11] propose a methodology similar to the one discussed in [8], exploiting the same feature extraction and feature selection techniques, but differing mainly on the classification algorithms used and finally on the results obtained. Starting from the reports generated at the end of the analysis performed by the Cuckoo Sandbox in JSON format, the first step consists of converting and extracting the system calls in MIST format. After that, the model follows the heuristic methodology of the decomposition in n-grams, where n represents the length of the sub-sequences of characters to extract, which represent system calls. Exploiting the metric of the Information Gain and ordering the features in a decreasing way, extract the final vector of the characteristics that will be fundamental for all the activities of automatic learning. After the feature selection phase, the final feature vector will be composed respectively selecting the top 200, 400, and 600 features in the form of n-grams, with n equal to 3 and 4. Some of the classifiers chosen by the authors are:

- Adaboost;
- Random Forest;
- Naive Bayes;
- Logistic Regression;
- Random Tree.

The dataset used for the testing phase is composed of 3000 files belonging to the non-malware category and 3100 malware, which belong to different categories. The test shows that the Random Forest classifier obtains the best results both in accuracy and in a lower rate of false positives and a higher rate of true positives. Ndibanje et al. [12] created a framework for malware de-obfuscation and analysis using machine learning algorithms which showed good performances on detecting possible threats. In [13], there is a survey that provides a global overview of how machine learning algorithms can be used in the context of offense-defense and, more in general, cybersecurity. Authors in [14] developed Sisyfos, a modular and extendible platform for malware analysis comprehensive of a web interface. The accuracy is tested using Random Forest Classifier reaching 98%. Kim in [15] presents a combination of static and dynamic analysis of various types of malware using several machine learning algorithms for classification aim. Moreover, the author estimates a malware risk index for using an analytic hierarchy process to detect malware and their probabilities. Choi et al. [16] used KNN combined with vantage-point tree for classification of malware; they reduced the detection time by 67% and increased detection rate by 25%. In [17], several machine learning techniques are used, demonstrating that the decision tree based family outperforms all others. In particular, the decision tree reduced the false alarm rate by 2% and reached an error rate of 99% of F1-score. El-Shafai et al. [18] used a visual encoding of malware in order to use a deep convolutional neural network and perform classification. Malware are converted from binary files into images, then VGG16, AlexNet, DarkNet-53, DenseNet, and ResNet CNNs were trained in transfer learning mode on the proposed dataset reaching as high as 99.97% of accuracy, but without promoting insights about what are the learned patterns and how they correlate to malware. A survey on shallow and deep learning techniques to detect ransomware malware in IoT networks is provided in [19] by Fernando et al. Instead, Ref. [20] provides a survey concerning malware detection in mobile devices (especially Android), categorizing the literature to three dimensions: type of analysis, features, and techniques. A recently adopted approach for Explainable AI is the SHAP library, which determines the most involved features in the classification process. Rao et al. [21] propose using such a library to analyze the result of using an Isolation Forest technique on the NSL-KDD dataset. In detail, the technique is employed to label the data based on the combinations of features judged to be of major significance. A different approach is the following one proposed by Wang et al. [22], who, even though they apply to the same NSL-KDD dataset, propose a method to explain locally and globally the predictions made by an Intrusion Detection System. The innovation brought by this approach is a certain coherence between what is produced by the model realized by the authors and the peculiarities of the specific attacks, allowing the operator using the system to have more punctual information to make more informed decisions. The two works considered provide some initial approaches to using the library in the cybersecurity domain, using the explanations to add information to the prediction made by a classifier. The work proposed offers a similar approach but is oriented towards identifying the APIs that most influence the classification process, thus allowing the individuation of some elements that can be warning signs. This approach is also oriented in the perspective of an optimization of the time for inference. As it is possible to see from this literature review, several techniques are used on different datasets and in different conditions. Thus, it is difficult to draw any conclusion about the best techniques for this kind of problem. Exotic, nonstandard techniques should be re-implemented to be tested on exactly the same conditions, but some work lacks details to re-implement their technique. One of the final goals of this work is to test state-of-the-art tree-based techniques and their counterpart, which make use of deep neural networks on the same datasets in exactly the same conditions. This would allow the creation of a basic testing framework that could be used by other authors and allow comparable results.

3. Dataset Description

In order to build an effective model, as well as for subsequent evaluation, it is necessary to have a large amount of data. This section describes the datasets used for the experiments.

The dataset *malware-analysis-datasets-api-call-sequences* [3,4] contains 42,797 malware API call sequences and 1079 goodware API call sequences. Each API call sequence consists of the first 100 consecutive non-repeating calls associated with the parent process, extracted from the "calls" elements of the reports obtained from Cuckoo Sandbox. The *APIMDS* dataset [5] instead consists of 23,080 malware samples randomly chosen from two other datasets: the Malicia project [23] and Virus Total [24]. A detailed description of the organization of the dataset can be found in Table 1, which shows the multitude of malware types recognizable using the analyzed approaches. However, this study focuses on binary classification.

Table 1. APIMDS dataset structure [5].

Category	Subcategory	Ratio (%)
Backdoor	-	3.37
Worm	Worm	3.32
	Email-Worm	0.55
	Net-Worm	0.79
	P2P-Worm	0.30
Packed	-	5.57
PUP	Adware	13.63
	Downloader	2.94
	WebToolbar	1.22
Trojan	Trojan (Generic)	29.3
	Trojan-Banker	0.14
	Trojan-Clicker	0.12
	Trojan-Downloader	2.29
	Trojan-Dropper	1.91
	Trojan-FakeAV	18.8
	Trojan-GameThief	0.63
	Trojan-PSW	3.79
	Trojan-Ransomware	2.58
	Trojan-Spy	3.12
Misc.	-	5.52

4. Algorithms Used

The aim is to compare deep learning and shallow learning techniques on the two different datasets previously mentioned for the benchmark under consideration. The models in question are respectively subdivided in Shallow learning techniques:

- Random Forest;
- CatBoost;
- XGBoost;
- ExtraTrees;

Deep learning techniques:

- TabNet;
- NODE (Neural Oblivious Decision Ensembles).

4.1. Random Forest

Random Forest [25] is a classifier obtained from the aggregation, through bagging, of decision trees. A decision tree is an acyclic graph of decisions and their possible consequences, mainly used to create an "action plan" aimed at a purpose. The particularity of this model is the clarity in the expression of the information, which is represented as a tree. Therefore, the decision tree is a predictive model in which each internal node represents a variable. The arcs between a parent node and the child nodes are obtained by distinguishing the paths based on the value of one of the data features to be classified; each branching step is the result of a splitting based on equality, majority, or minority condition applied on a variable. The leaf nodes instead represent the predicted class. The classification is obtained by following the expressed conditions, from the root node to a leaf node. Thus, a decision tree is, in fact, a set of decision rules based on the values of the variables. A decision tree is generated starting from a dataset; in the training phase, defining some stopping criteria (halting) is necessary since a much-ramified tree significantly increases the computational complexity against little benefits in classification accuracy. The bagging operation allows merging multiple models of the same type, all derived from the same original dataset, using data obtained from sampling without replacement. Random Forest is then composed of a set of decision trees, all trained from the same dataset, where each decision tree is trained on a random subset of the variables. The resulting classification is the mode (in the case of a classification task) or the mean (in the case of a regression task) of the results obtained by the individual decision tree classifiers.

4.2. XGBoost

Gradient Boosting is a Machine Learning technique developed to solve classification and regression problems. It creates prediction models from an ensemble of smaller prediction models, typically represented by decision trees. Model building is done stepwise, in the same way as other boosting algorithms, but the advantage of this technique lies in the fact that it can generalize by allowing the optimization of an arbitrary differentiable loss function. This approach differs from others in that it is suitable for a wide range of tasks and has excellent portability as it supports a cross-platform between different programming languages and different operating systems.

XGBoost [26] improves the Gradient Boosting Machines (GBM) framework on which it is based through optimizations in the system and improvements in the algorithms. XGBoost takes advantage of cache-aware algorithms by allocating internal buffers in each of the threads to store computed statistics. XGBoost uses parallelization, as the process of tree construction is done in a parallel way, due to the exchange of nested loops used in tree construction. The `max_depth` parameter allows for adjusting the division of each tree, instead of using the basic stop criterion. The features that enable algorithmic improvements are:

- Allows for avoiding the phenomenon of overfitting in the training phase, ensuring regularization;

- Capability of adaptation in the presence of features with missing values, since it automatically selects a value to replace the missing ones, based on the training data;
- Cross-validation at each iteration.

XGBoost models currently represent an excellent solution in classification and regression tasks, both for their performance in terms of the results obtained and computation time compared to other algorithms.

4.3. CatBoost

CatBoost [27] is an open-source software library that defines itself as state of the art for the gradient boosting [28] technique on decision trees. During training, a set of trees are built consecutively, where each new tree is built with a different and reduced loss to the previous. This influences the tree structure greedily. Another important aspect is the capability of making feature values' quantization automatically: it automatically defines the thresholds to use to create disjoint ranges (bins) for the feature values and labels. In addition to providing accuracy, robustness, practicality, and extensibility, all backed by the ease of use, it also offers direct support for categorical format data and has a GPU-computable version. CatBoost can easily be integrated with deep learning frameworks like Google's TensorFlow and Apple's Core ML. It can work with different data types, and it supports explainable artificial intelligence through feature ranking to sort the most important features. In this work, the model for plain supervised classification on a set of features is used.

4.4. Extra Trees

Extremely Randomized Trees [29], also based on decision trees, combines the results from the base models organized in the forest to determine the prediction output. The Extra Trees model creates a large number of decision trees that are unpruned and generated automatically starting from the data. For classification purposes, the predictions are made using majority voting and regression purposes by averaging the result of each tree. It is based on the intuition that, by building trees by randomly picking the feature to split at each step for each tree, the model will not overfit. This makes the ensemble of trees less correlated and increases variance. This increased variance can be faced by increasing the number of trees. The main difference with Random Forest is in the construction of the decision trees within the forest: ExtraTrees does not exploit the bootstrap methodology on learning samples and randomly selects a branching point on each feature instead of targeting the optimal split of Random Forest.

4.5. TabNet

The idea behind TabNet [30] is to build a neural network for processing tabular data. As demonstrated in other application domains, e.g., images, the application of deep learning techniques has brought a significant performance boost as the dataset grows vis-à-vis machine learning techniques. It is designed to learn similar decision tree-based models and have their benefits: interpretability and feature selection. It sequentially uses the multi-headed attention technique to choose from which features to use at each decision step. Feature selection is made on an instance-by-instance basis to differentiate for each input. The model is built in multiple sequential steps by passing the input instances from one step to the other: it is composed of a transformer with several in parallel multi-headed attention using a sparse-matrix to give Sparse feature selection. This aspect increases interpretability by extracting, for each instance, weights (also known as importances) of features. Initially, the dataset is processed without any feature engineering. Then, instances are Batch normalized and passed to the feature transformer, where it passes through different decision steps made up of fully connected layers and different gated linear unit (GLU) activation functions. The output of each activation is embedded (e.g., with a sum operator) with others, and depending on the problem, if regression or classification, a different loss function is used for performing end-to-end training. It is important to state that normalization with 0.5 helps stabilize learning by ensuring that the variance throughout the network does not change dramatically. The model used for this problem is the classification version with automatic feature engineering and feature selection. A significant drawback of this technique is the massive amount of data required for learning, which is, among other things, one of the major limitations of the multi-headed attention model.

4.6. NODE (Neural Oblivious Decision Ensembles)

The NODE [31] algorithm consists of a deep learning architecture designed to work in the presence of tabular data. The basic unit of this architecture is the oblivious decision tree, with the peculiarity of being constrained to use the same feature for the split and the same split threshold for all nodes having the same depth. The network is trained in an end-to-end way through backpropagation. In the deep version, where multiple NODE layers are stacked on top of the other, the connection among these layers is made using the residual connection from the ResNet work. The input features and all other layers' outputs are concatenated before being fed to the next NODE layer. In the end, each layer's output is averaged in the case of regression, and majority voting is used in case of classification, similarly to Extra Trees. The model used for NODE is the deep version for classification purposes.

4.7. Experimental Setup and Preprocessing

Both datasets are unbalanced towards the "malware" class, and this has to be taken into account for all the following steps of the process. The solution is to leverage some sampling methods, which modify the starting data making their distribution more balanced. Tests were carried out in 10-fold stratified cross-validation, where, for each fold, the number of goodware and malware instances are randomly balanced (random undersampling). The values reported are the average values for the 10 folds.

For the dataset *malware-analysis-datasets-api-call-sequences*, all features were already numerical. For the *APIMDS* dataset, instead, all categorical variables were one-hot encoded, numerical variables were used as-is. In addition, only the first 114 features were kept into consideration for the *APIMDS* dataset. This is because the following features were composed of almost all null values (>90% of null values), and thus it has been decided to remove them in order to decrease the complexity of the problem and better face the famous Bellman's "curse of dimensionality" problem. For using standard algorithms such as NODE, TabNet, and so on, datasets were standardized with z-score normalization prior to the training process.

The stratified version ensures that the same proportion of the data is maintained in all training and test sets in the same way as the original data. This is so that no value is over (or under)—represented in the various sets to have a more accurate estimate of the results.

The configuration of the algorithms is as follows: Random Forest and Extra Tree are configured to use 100 trees and the "gini" index. XGBoost is configured as suggested by the algorithm's authors with the max depth pre-pruning parameter equal to 6, and the learning rate equal to 0.3. CatBoost is configured without any adjustment or selection of hyperparameters and default parameters. TabNet attention embedding amplitude and decision accuracy amplitude are equal to 8 as per the official publication. The default settings were used for NODE.

5. Results

Tables 2 and 3 present the results calculated for each algorithm, indicated in the first column, derived from the stratified cross-validation experiment with k equal to 10 and averaged between them. In particular, with regard to the metrics of Precision, Recall, F1-Score, they are to be considered calculated with macro average. The reference dataset for Table 2 is *malware-analysis-datasets-api-call-sequences*, while Table 3 presents the results for dataset *APIMDS*.

Table 2 highlights the differences between the algorithms. In this specific case, the deep learning techniques NODE and TabNet, although showing an average F1-macro score in line with the state of the art, have F1-macro score and Area Under the ROC curves lower when compared to algorithms using trees such as Random Forest, XGBoost, and CatBoost. Finally, CatBoost presents the best performance, which seems to obtain the best trade-off between precision and recall.

Table 2. Metrics averaged over 10 folds—*malware-analysis-datasets-api-call-sequences* dataset.

Algorithm	Precision	Recall	F1 Score	AUC ROC
Random Forest	0.965	0.809	0.869	0.8094
CatBoost	0.962	0.82	0.877	0.8201
XGBoost	0.9575	0.7816	0.8473	0.7816
ExtraTrees	0.97	0.7411	0.8162	0.7411
NODE	0.9532	0.7238	0.7971	0.7238
TabNet	0.8985	0.7674	0.8166	0.7674

Table 3. Metrics averaged over 10 folds—*APIMDS* dataset.

Algorithm	Precision	Recall	F1 Score	AUC ROC
Random Forest	0.9914	0.9215	0.9532	0.9215
CatBoost	0.9955	0.9399	0.9655	0.9399
XGBoost	0.9944	0.9619	0.9779	0.9632
ExtraTrees	0.9949	0.9132	0.9498	0.9132
NODE	0.9948	0.9166	0.9519	0.0022
TabNet	0.9843	0.9264	0.9529	0.9264

The dataset *APIMDS* consists of 300 goodware class examples (labeled with 0) and 23,146 malware class examples (labeled with 1). Again, shallow learning techniques using trees such as Random Forest, Cat Boost and XGBoost perform slightly better than their deep learning counterparts such as TabNet and NODE. Specifically, it can be seen that, as in the case of the previous dataset, XGBoost and CatBoost are the best techniques both from an AUC ROC and F1-macro score point of view. XGBoost is slightly better than CatBoost in this experiment because it has better recall; however, it is also relevant to note how robust CatBoost is to the dataset change, i.e., its ability to perform with accuracies in line with the state of the art regardless of dataset. It is now of interest to understand what features the best performing algorithms, such as CatBoost, deem most important for classifying instances as malware and as goodware using the explainable artificial intelligence SHAP technique. Table 4 illustrates the training and prediction times of the treated algorithms as applied to dataset *malware-analysis-datasets-api-call-sequences*. The time intervals represented in the table are expressed in seconds. To achieve a more accurate estimate, the times reported in the table were averaged on the number of iterations completed for each step by each algorithm, i.e., 100.

Table 4. Training and prediction times measured in seconds—*malware-analysis-datasets-api-call-sequences* dataset. * This value is relative to a single training task. Because it is enormous, it is impossible to average over 100 iterations. Instead, the prediction time was averaged.

Algorithm	Training Time	Prediction Time
Random Forest	12.84419	0.07022
CatBoost	6.70016	0.00566
XGBoost	7.50087	0.01600
TabNet	38.97393	0.16049
ExtraTrees	4.22829	0.07658
NODE	49,774.68524 *	31.14673

Table 5 exposes the training and prediction times of the algorithms on the *APIMDS* dataset. The same considerations made for the *malware-analysis-datasets-api-call-sequences* dataset, expressed in the previous paragraph, apply.

Table 5. Training and prediction times measured in seconds—*APIMDS* dataset. * This value is relative to a single training task. Because it is enormous, it is impossible to average over 100 iterations. Instead, the prediction time was averaged.

Algorithm	Training Time	Prediction Time
Random Forest	4.69358	0.02279
CatBoost	6.93550	0.00476
XGBoost	4.87004	0.00682
TabNet	26.50733	0.09314
ExtraTrees	2.48567	0.03346
NODE	26,426.73316 *	18.67745

As it is possible to observe from Tables 4 and 5 concerning the implementation of these algorithms in an edge AI use case, the clear winners are Extra Tree, CatBoost, and XGBoost algorithms. All algorithms do not use end-to-end backpropagation training and neural networks in practice. This is because backpropagation training requires several epochs to allow the model to converge to a stable solution and provide reliable results. In addition, the nonlinearity brought from activation functions makes the learning phase of the neural network computationally intensive. In addition, from a memory constraint perspective, tree-based algorithms avoid the use and concatenation of big tensors found in deep neural networks (especially when the number of features is very high) and thus less memory greedy.

6. Reasoning

Once the experimental models have been built, and the metrics have been estimated, an explanation of the output values is now provided. The SHAP [6] technique is based on Shapely's value theory, which has its origin in game theory. Each instance of the dataset corresponds to a "player" in a game, in which the forecast represents the payoff. In contrast to game theory, in which the payoff is assigned to the players on the basis of the choices they make, in this case, the payoff obtained is the result of the combination of variables that characterize the dataset. To each variable, then, a weight, or marginal contribution, is assigned, and it represents the value of SHAPley. The marginal contribution of each feature is calculated considering all the possible interactions with the other features present in the model. It is estimated, therefore, how much information is contained in every combination, estimating the added value that every feature brings in the prediction. To every variable, a marginal contribution is associated based on the increase in the accuracy of the forecast. Specifically, many combinations are tested, depending on the number of features to be included in the model. For every feature, all the combinations with the others are calculated attributing in a first phase the previewed weight; subsequently, the same prediction is calculated without considering the variable in question and, on the base of the difference of prediction obtained, the marginal contribution is attributed to that feature. This process is carried out for every feature of the model on all the instances of the database in order to have the average value of the marginal contribution of each feature. Analyzing Shapely values on CatBoost trained model on *malware-analysis-datasets-api-call-sequences* dataset, as depicted in Table 6, it is possible to extract the importance of each feature for classifying the instance as malignant or non-malignant and their relative magnitude of importance.

Therefore, by reducing the feature space, an experiment was carried out in order to study the API calls most present in malware, measuring the frequency of the values and comparing it to the number of examples belonging to the same class. The same was done for the goodware class. Once the results for both categories were obtained, the "distances" were calculated, i.e., the differences between the presence of the API call in malware and the presence of the same API call in goodware were highlighted (values expressed in percentage). Finally, the following analysis shows that APIs *LdrGetProcedureAddress, LdrGetDllHandle, LoadResource, FindResourceExW, LdrLoadDll, GetSystemInfo, LoadStringA, NtProtectVirtualMemory, GetSystemMetrics* and *CryptAcquireContextW* are used most by malware and least by goodware; Instead, API *NtClose, NtCreateFile, RegOpenKeyExW, NtOpenKey, RegCloseKey, GetSystemDirectoryW, RegQueryValueExW, GetSystemWindowsDirectoryW, NtOpenSection* and *SetErrorMode* are more frequently used by goodware and less by malware.

Table 6. Top 10 API calls for malware class and goodware class according to the distance measure. CatBoost on dataset *malware-analysis-datasets-api-call-sequences*.

API Call	Distance Value
Malware class	
LdrGetProcedureAddress	1.16378
LdrGetDllHandle	0.76617
LoadResource	0.59114
FindResourceExW	0.37002
LdrLoadDll	0.32114
GetSystemInfo	0.24794
LoadStringA	0.21848
NtProtectVirtualMemory	0.20139
GetSystemMetrics	0.18984
CryptAcquireContextW	0.18984
Goodware class	
NtClose	0.80112
NtCreateFile	0.74446
RegOpenKeyExW	0.56946
NtOpenKey	0.53424
RegCloseKey	0.48842
GetSystemDirectoryW	0.37624
RegQueryValueExW	0.31458
GetSystemWindowsDirectoryW	0.31099
NtOpenSection	0.27815
SetErrorMode	0.27693

As far as *APIMDS* dataset is concerned, the same experiment was conducted, considering the 114 features of which it is composed. Here, too, the frequency of API call values present in malware was first measured and then related to the number of examples belonging to the same class. It was similarly done for the goodware class. Finally, the distances between the API call in malware and the API call in goodware were calculated (values expressed in percentage) as shown in Table 7. Thus, the 10 API calls most involved in malware and least involved in goodware are: *LoadLibraryExW, LocalAlloc, GetProcAddress, GetSystemMetrics, GetVersionExW, GetModuleHandleW, CreateFileW, RegisterClipboardFormatW, LoadLibraryW, MapViewOfFileEx*;

However, the 10 API calls most involved in goodware and least in malware are respectively: *CoUninitialize, TerminateProcess, MessageBoxW, FormatMessageW, CoCreateInstance, PostQuitMessage, TranslateMessage, GetWindowRect, PostMessageW, FreeEnvironmentStringsA*.

Let us now seek confirmation of what emerges from these data by analyzing examples of correctly classified malware and goodware and evaluating the related API calls. The following entries are selected from the dataset *malware-analysis-datasets-api-call-sequences*: entries n° 42,679, 17,006, and 6173 for the goodware class; and entries n° 41,444, 43,641, 8921 for the malware class. Table 8 shows a chart that highlights the calls made by each entry, focusing on the calls that fall in the top 10 API calls made by malware and goodware (previously reported in Table 6).

Table 7. Top 10 API calls for malware class and goodware class according to the distance measure. CatBoost on dataset *APIMDS*.

API Call	Distance Value
Malware class	
LoadLibraryExW	1.43355
LocalAlloc	1.07286
GetProcAddress	1.04397
GetSystemMetrics	0.94762
GetVersionExW	0.82373
GetModuleHandleW	0.74527
CreateFileW	0.73244
RegisterClipboardFormatW	0.72595
LoadLibraryW	0.69086
MapViewOfFileEx	0.68175
Goodware class	
CoUninitialize	0.07650
TerminateProcess	0.06663
MessageBoxW	0.06323
FormatMessageW	0.06280
CoCreateInstance	0.06216
PostQuitMessage	0.05991
TranslateMessage	0.04931
GetWindowRect	0.03996
PostMessageW	0.03974
FreeEnvironmentStringsA	0.03927

Looking at the first record, 8 out of 10 of the most invoked APIs turn out to belong to the table that summarizes the TOP 10 APIs of the Goodware category, 1 API belongs to those typically invoked by malware, 1 API does not belong to the Top 10 ranking. In addition, the same happens in the second and third records, i.e., they show a prevalence of calls belonging to the TOP 10 of goodware. Next, by evaluating the results for the malware class records, the results for the first malware show 6 out of 10 APIs from the Top 10 ranking in malware, and 2 out of 10 belong to the Top 10 ranking in goodware. The remaining two APIs do not belong to the ranking of the experiment. In addition, in the last two records, a similar situation is presented, where the results show a prevalence of calls belonging to the TOP 10 in malware. Thus, the data show a confirmation of the influence of API calls.

Now, let analogous evaluations be performed on the *APIMDS* dataset. Table 9 shows the relevant API calls for three examples chosen for the goodware class (23,235, 23,386, 23,313) and the malware class (22,042, 15,316, 11,972), with similar classification compared to what was already done for the *malware-analysis-datasets-api-call-sequences* dataset. From Table 7, it can first be seen that, unlike the other dataset, the most frequently used API calls by goodware show rather small distance values. This evaluation follows that they have little power to discriminate in favor of goodware. Focusing on the goodware examples, it appears that, although most of the API calls do not belong to any of the top 10 expressed in the table by category, the presence of at least one of those indicated for the correct category is still relevant. Observing Table 9, it is arguable that these examples have been classified as goodware since the calls identified in the top 10 of malware are missing, and there are few calls recognized pertaining mainly to goodware. The classification, in this case, was not, as in the case of the previous dataset, based on the presence of benign calls but rather on the absence of malicious calls. Conversely, in the malware examples, it is possible to see the power of discrimination carried out by the top 10 malware API calls since the examples reported contain, in many cases, such calls.

Table 8. API calls performed by some examples of the dataset *malware-analysis-datasets-api-call-sequences*.

	Index	API Calls that are in Top 10 Malware Calls	API Calls that are in Top 10 Goodware Calls	Other API Calls
G	42,679	*LdrGetProcedureAddress*	*NtCreateFile* *NtOpenKey* *NtOpenSection* *RegOpenKeyExW* *RegCloseKey* *RegQueryValueExW* *NtClose* *SetErrorMode*	*NtMapViewOfSection*
G	17,006	*LdrGetProcedureAddress* *LdrLoadDll*	*NtCreateFile* *NtOpenKey* *NtOpenSection* *RegCloseKey* *RegOpenKeyExW* *RegQueryValueExW* *NtClose*	*NtMapViewOfSection*
G	6173	*LdrGetProcedureAddress* *LdrLoadDll*	*NtCreateFile* *NtOpenKey* *NtOpenSection* *RegCloseKey* *RegOpenKeyExW* *RegQueryValueExW* *NtClose*	*NtMapViewOfSection*
M	41,444	*LdrGetProcedureAddress* *LdrLoadDll* *LdrGetDllHandle* *GetSystemMetrics* *FindResourceExW* *NtProtectVirtualMemory*	*NtClose* *NtOpenKey*	*NtQueryValueKey* *NtAllocateVirtualMemory*
M	43,641	*LdrGetProcedureAddress* *LdrGetDllHandle* *LdrLoadDll* *CryptAcquireContextW*	*NtClose* *NtCreateFile*	*NtQuerySystemInformation* *NtAllocateVirtualMemory* *NtFreeVirtualMemory* *NtCreateSection*
M	8921	*LdrGetProcedureAddress* *LdrLoadDll* *GetFileType* *GetSystemMetrics*	*NtClose*	*NtAllocateVirtualMemory* *DrawTextExW* *GetSystemTimeAsFileTime* *NtDuplicateObject* *SetUnhandledExceptionFilter*

As discussed above, the results shown in the table for each of the two datasets, including those obtained from these additional experiments, are different. The cause is that the datasets examined are different, with different features, numerosity, and balance. In particular, the cardinality of the examples belonging to the *malware-analysis-datasets-api-call sequences* dataset is higher than the cardinality of the examples from the APIMDS dataset. On the other hand, the former collects a sequence of the first 100 features, while the latter has 114. The other difference there is within the features. In particular, the first dataset expresses 307 distinct API values within it, and the second dataset contains 2264. However, it is essential to note that the results are consistent with each other; all algorithms applied to the same dataset have accuracies that vary ±7% on *malware-analysis-datasets-api-call-sequences* and ±3% on *APIMDS*.

Table 9. API calls performed by some examples of the dataset *APIMDS*.

	Index	API Calls that are in Top 10 Malware Calls	API Calls that are in Top 10 Goodware Calls	Other API Calls
G	23,235		*FormatMessageW*	*−1* *LocalFree* *SetThreadUILanguage* *GetStdHandle* *GetConsoleOutputCP* *GetConsoleMode* *GetModuleHandleA* *WideCharToMultiByte* *WriteFile*
G	23,386		*MessageBoxW*	*−1* *RegCreateKeyExW* *RegCloseKey* *RegSetValueExW* *FreeSid* *RegEnumKeyExW* *AllocateAndInitializeSid* *CheckTokenMembership* *RegOpenKeyExW*
G	23,313		*FormatMessageW* *TerminateProcess*	*−1* *GetLengthSid* *LookupAccountSidW* *GetLastError* *GetStdHandle* *GetFileAttributesW* *VerifyVersionInfoW* *GetCurrentProcess*
M	22,042	*GetModuleHandleW* *GetVersionExW* *GetProcAddress* *LoadLibraryExW* *GetSystemMetrics* *LocalAlloc* *LoadLibraryW* *RegisterClipboardFormatW*		*DisableThreadLibraryCalls* *GetModuleFileNameW*
M	15,316	*GetProcAddress* *GetModuleHandleW* *LoadLibraryExW* *GetVersionExW* *RegisterClipboardFormatW* *LoadLibraryW*		*GetVersionExA* *GetStartupInfoA* *GetACP* *GetModuleFileNameW*
M	11,972	*GetModuleHandleW* *GetProcAddress* *LoadLibraryExW* *GetVersionExW*		*GetModuleHandleA* *GetCurrentThreadId* *GetVersionExA* *CreateEventW* *GetTickCount* *VirtualQueryEx*

7. Conclusions

This work presents a benchmark of most modern shallow learning and deep learning techniques for tabular data in the context of malware detection within API. Two datasets have been tested against six different machine learning algorithms. The accuracy of algorithms trained in stratified 10-fold cross-validation mode proved the quality of the results and the efficacy of those techniques on this specific problem. In particular, it is important to state that shallow learning techniques tend to perform better and to converge faster (less training time) to a suitable solution. Nonetheless, once the model is built, its intrinsic complexity is lighter than the deep learning solutions. This suggests the possibility of using such techniques (e.g., CatBoost) on edge devices to perform inference in a near real-time scenario. This solution opens new research directions, such as the continuous learning for edge artificial intelligence API calls, where new and unknown API calls are then stored, and the model is re-trained with or without human intervention to provide direct feedback. The use of Explainable Artificial Intelligence technique called SHAP based on Shapely values, which in turn, bases its roots on the proven Nash equilibrium and game theory, allowed for extracting what are the most important API calls to catch in order to classify an instance as malignant or what other API calls are safe. These insights were generated automatically and without any human intervention allowing, thus, the development of an automatic visualization tool for suspicious API calls.

Author Contributions: Methodology, software, writing—original draft preparation: A.C., A.I.; Conceptualization, validation: V.D., S.G., D.I., G.P.; Supervision, Project Administration: D.I.; Project Administration, Funding Acquisition: G.P. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the Fondo Europeo di Sviluppo Regionale Puglia Programma Operativo Regionale (POR) Puglia 2014-2020-Axis I-Specific Objective 1a-Action 1.1 (Research and Development)-Project Title: CyberSecurity and Security Operation Center (SOC) Product Suite by BV TECH S.p.A., under Grant CUP/CIG B93G18000040007.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Cuckoo Foundation. Cape Sandbox. Available online: https://capev2.readthedocs.io/en/latest/introduction/what.html (accessed on 13 December 2021).
2. Catak, F.O.; Ahmed, J.; Sahinbas, K.; Khand, Z.H. Data augmentation based malware detection using convolutional neural networks. *PeerJ Comput. Sci.* **2021**, *7*, e346. [CrossRef] [PubMed]
3. Oliveira, A.; Sassi, R. Behavioral malware detection using deep graph convolutional neural networks. *Int. J. Comp. Appl.* **2021**, *174*, 0975–8887.
4. Angelo Oliveira. Malware Analysis Datasets: API Call Sequences. IEEE Dataport. Available online: https://ieee-dataport.org/open-access/malware-analysis-datasets-api-call-sequences (accessed on 13 December 2021).
5. Ki, Y.; Kim, E.; Kim, H.K. A novel approach to detect malware based on API call sequence analysis. *Int. J. Distrib. Sens. Networks* **2015**, *11*, 659101. [CrossRef]
6. Štrumbelj, E.; Kononenko, I. Explaining prediction models and individual predictions with feature contributions. *Knowl. Inf. Syst.* **2014**, *41*, 647–665. [CrossRef]
7. The Honeynet Project. CuckooML: Machine Learning for Cuckoo Sandbox. Available online: https://github.com/honeynet/cuckooml (accessed on 13 December 2021).
8. Darshan, S.S.; Kumara, M.A.; Jaidhar, C. Windows malware detection based on cuckoo sandbox generated report using machine learning algorithm. In Proceedings of the 2016 11th International Conference on Industrial and Information Systems (ICIIS), Roorkee, India, 3–4 December 2016; pp. 534–539.
9. Ali, M.; Shiaeles, S.; Bendiab, G.; Ghita, B. MALGRA: Machine learning and N-gram malware feature extraction and detection system. *Electronics* **2020**, *9*, 1777. [CrossRef]
10. Kumar, R.; Sethi, K.; Prajapati, N.; Rout, R.R.; Bera, P. Machine Learning based Malware Detection in Cloud Environment using Clustering Approach. In Proceedings of the 2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), Kharagpur, India, 1–3 July 2020; pp. 1–7.
11. Udayakumar, N.; Subbulakshmi, T. Classification of Malware with MIST and N-Gram Features Using Machine Learning. *Int. J. Intell. Eng. Syst.* **2021**, *14*, 323–333.
12. Ndibanje, B.; Kim, K.H.; Kang, Y.J.; Kim, H.H.; Kim, T.Y.; Lee, H.J. Cross-method-based analysis and classification of malicious behavior by api calls extraction. *Appl. Sci.* **2019**, *9*, 239. [CrossRef]
13. Truong, T.C.; Diep, Q.B.; Zelinka, I. Artificial intelligence in the cyber domain: Offense and defense. *Symmetry* **2020**, *12*, 410. [CrossRef]
14. Serpanos, D.; Michalopoulos, P.; Xenos, G.; Ieronymakis, V. Sisyfos: A Modular and Extendable Open Malware Analysis Platform. *Appl. Sci.* **2021**, *11*, 2980. [CrossRef]
15. Kim, D. Decision-Making Method for Estimating Malware Risk Index. *Appl. Sci.* **2019**, *9*, 4943. [CrossRef]
16. Choi, S. Combined kNN Classification and hierarchical similarity hash for fast malware detection. *Appl. Sci.* **2020**, *10*, 5173. [CrossRef]
17. Usman, N.; Usman, S.; Khan, F.; Jan, M.A.; Sajid, A.; Alazab, M.; Watters, P. Intelligent dynamic malware detection using machine learning in IP reputation for forensics data analytics. *Future Gener. Comput. Syst.* **2021**, *118*, 124–141. [CrossRef]
18. El-Shafai, W.; Almomani, I.; AlKhayer, A. Visualized malware multi-classification framework using fine-tuned CNN-based transfer learning models. *Appl. Sci.* **2021**, *11*, 6446. [CrossRef]

19. Fernando, D.W.; Komninos, N.; Chen, T. A Study on the Evolution of Ransomware Detection Using Machine Learning and Deep Learning Techniques. *IoT* **2020**, *1*, 551–604. [CrossRef]
20. Alswaina, F.; Elleithy, K. Android malware family classification and analysis: Current status and future directions. *Electronics* **2020**, *9*, 942. [CrossRef]
21. Rao, D.; Mane, S. Zero-shot learning approach to adaptive Cybersecurity using Explainable AI. *arXiv* **2021**, arXiv:2106.14647.
22. Wang, M.; Zheng, K.; Yang, Y.; Wang, X. An explainable machine learning framework for intrusion detection systems. *IEEE Access* **2020**, *8*, 73127–73141. [CrossRef]
23. Nappa, A.; Rafique, M.Z.; Caballero, J. Driving in the cloud: An analysis of drive-by download operations and abuse reporting. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 1–20.
24. Virus Total. Available online: https://www.virustotal.com/ (accessed on 13 December 2021).
25. Breiman, L. Random forests. *Mach. Learn.* **2001**, *45*, 5–32. [CrossRef]
26. Chen, T.; Guestrin, C. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd ACM Sigkdd International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 785–794.
27. Prokhorenkova, L.; Gusev, G.; Vorobev, A.; Dorogush, A.V.; Gulin, A. CatBoost: Unbiased boosting with categorical features. *arXiv* **2017**, arXiv:1706.09516.
28. Friedman, J.H. Stochastic gradient boosting. *Comput. Stat. Data Anal.* **2002**, *38*, 367–378. [CrossRef]
29. Geurts, P.; Ernst, D.; Wehenkel, L. Extremely randomized trees. *Mach. Learn.* **2006**, *63*, 3–42. [CrossRef]
30. Arık, S.O.; Pfister, T. Tabnet: Attentive interpretable tabular learning. *arXiv* **2020**, arXiv:1908.07442.
31. Popov, S.; Morozov, S.; Babenko, A. Neural oblivious decision ensembles for deep learning on tabular data. *arXiv* **2019**, arXiv:1909.06312.