Digital Investigation 28 (2019) S77–S87

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

MalDy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports

ElMouatez Billah Karbab*, Mourad Debbabi

Concordia University, Montreal, Canada

ARTICLE INFO

Article history:

Keywords:
Malware
Android
Win32
Behavioral analysis
Machine learning
NLP

ABSTRACT

In response to the volume and sophistication of malicious software or malware, security investigators rely on dynamic analysis for malware detection to thwart obfuscation and packing issues. Dynamic analysis is the process of executing binary samples to produce reports that summarise their runtime behaviors. The investigator uses these reports to detect malware and attribute threat types leveraging manually chosen features. However, the diversity of malware and the execution environments make manual approaches not scalable because the investigator needs to manually engineer fingerprinting features for new environments. In this paper, we propose, MalDy (mal die), a portable (plug and play) malware detection and family threat attribution framework using supervised machine learning techniques. The key idea of MalDy portability is the modeling of the behavioral reports into a sequence of words, along with advanced natural language processing (NLP) and machine learning (ML) techniques for automatic engineering of relevant security features to detect and attribute malware without the investigator intervention. More precisely, we propose to use *bag-of-words* (BoW) NLP model to formulate the behavioral reports. Afterward, we build ML ensembles on top of BoW features. We extensively evaluate MalDy on various datasets from different platforms (Android and Win32) and execution environments. The evaluation shows the effectiveness and the portability of MalDy across the spectrum of the analyses and settings.

Introduction

Malware investigation is an important and time consuming task for security investigators. The daily volume of malware raises the automation necessity of detection and threat attribution tasks. The diversity of platforms and architectures makes the malware investigation more challenging. The investigator has to deal with variety of malware scenarios from Win32 to Android. Also, nowadays, malware targets all CPU architectures from x86 to ARM and MIPS that heavily influence the binary structure. The diversity of malware adopts the need for portable tools, methods, and techniques in the security investigator's toolbox for malware detection and threat attribution.

Binary code static analysis is a valuable tool to investigate malware in general. It has been used effectively and efficiently in many solutions (Arp et al., 2014), (Hu et al., 2013), (Karbab et al., 2018), (Karbab et al., 2017c), and (Mariconti et al., 2017) in PC and Android realms. However, the use of static analysis could be problematic on heavily obfuscated and custom packed malware. Solutions, such as (Hu et al., 2013), address those issues partially but they are platform/architecture dependent and cover only simple evading techniques. The dynamic analysis solutions, on the other hand, (Willems et al., 2007), (Bayer et al., 2009a), (Wong and Lie, 2016), (Alzaylaee et al., 2016), (SeveriTim Leek and Dolangavitt, 2018), (Karbab and Debbabi, 2018a), (Karbab and Debbabi, 2018b), (Isohara et al., 2011), (Rieck et al., 2011), (Paul and Filiol, 2016) show more robustness to evading techniques such as obfuscation and packing. Dynamic analysis's main drawback is its hungry to computation resources (Wang and Winston Shieh, 2015), (Graziano et al., 2015). Also, it may be blocked by anti-emulation techniques, but this is less common compared to binary obfuscation and packing techniques. For this reason, dynamic (also behavioral) analysis is still the default choice and the first analysis for malware by security companies.

* Corresponding author.
E-mail address: e_karbab@encs.concordia.ca (E.B. Karbab).

https://doi.org/10.1016/j.diin.2019.01.017

Static and behavioral analyses are sources for security features, which the security investigator uses to decide the maliciousness of a given program. Manual inspection of these features is a tedious task and could be automated using machine learning techniques. For this reason, the majority of the state-of-the-art malware detection solutions use machine learning techniques (Mariconti et al., 2017), (Arp et al., 2014). We could classify these solutions' methodologies into supervised and unsupervised. The supervised approach, such as (Nataraj et al., 2011) for Win32 and (Wu and Hung, 2014), (Chen et al., 2016) for Android, is the most used in malware investigation (Martinelli et al., 2016), (Chen et al., 2016). The supervised machine learning process starts by training a classification model on a train-set. Afterward, we use this model on new samples in a production environment. Second, the unsupervised approach, such as (Rieck et al., 2011), (Karbab et al., 2016a,b), (Bayer et al., 2009b), (Karbab et al., 2017a), (Karbab et al., 2017b), in which the authors cluster the malware samples into groups based on their similarity. Unsupervised learning is more common in malware family clustering (Rieck et al., 2011), and it is less common in malware detection (Karbab et al., 2016b).

In this paper, we focus on supervised machine learning techniques along with behavioral (dynamic or runtime) analyses to investigate malicious software. Dynamic and runtime analyses execute binary samples to collect their behavioral reports. The dynamic analysis makes the execution in a sandbox environment (emulation) where malware detection is an off-line task. The runtime analysis is the process of collecting behavioral reports from production machines. The security practitioner aims to obtain these reports to make an online malware checking without disturbing the running system.

Problem statement

The state-of-the-art solutions, such as in (Chen et al., 2016), (Amin et al., 2016), (Sgandurra et al., 2016), rely on manual security features investigation in the detection process. For example, StormDroid (Chen et al., 2016) used *Sendsms* and *Recvnet* dynamic features, which have been chosen base on a statistical analysis, for Android malware detection. Another example, the authors in (Kolbitsch et al., 2009) used explicit features to build behavior's graphs for Win32 malware detection. The security features may change based on the execution environment despite the targeted platform. For instance, the authors (Chen et al., 2016) and (Alzaylaee et al., 2016) used different security features due to the difference between the execution environments. In the context of the security investigation, we are looking for a portable framework for malware detection based on the behavioral reports across a variety of platforms, architectures, and execution environments. The security investigator would rely on this plug and play framework with a minimum effort. We plug the behavioral analysis reports for the training and apply (play) the produced classification model on new reports (same type) without an explicit security feature engineering as in (Chen et al., 2016), (Kolbitsch et al., 2009), (Chen et al., 2017); and this process works virtually on any behavioral reports.

MalDy

We propose, MalDy, a portable and generic framework for malware detection and family threat investigation based on behavioral reports. MalDy aims to be a tool on the security investigator toolbox to leverage existing behavioral reports to build a malware investigation tool without prior knowledge regarding the behavior model, malware platform, architecture, or the execution environment. More precisely, MalDy is portable because of the automatic mining of relevant security features to allow moving MalDy to learn new environments' behavioral reports without a security expert intervention. Formally, MalDy framework is built on top natural language processing (NLP) modeling and supervised machine learning techniques. The main idea is to formalize the behavioral report, in agnostic way to the execution environment, into a bag of words (BoW) where the features are the reports' words. Afterward, we leverage machine learning techniques to automatically discover relevant security features that help differentiate and attribute malware. The result is MalDy, a portable (Section 8.2), effective (Section 8.1), and efficient (Section 8.3) framework for malware investigation.

Result summary

We extensively evaluate MalDy on different datasets, from various platforms, under multiple settings to show our framework portability, effectiveness, efficiency, and its suitability for general purpose malware investigation. First, we experiment on Android malware behavioral reports of MalGenome (Zhou and Jiang, 2012), Drebin (Arp et al., 2014), and Maldozer (Karbab et al., 2018), (Karbab et al., 2017c) datasets along with benign samples from AndroZoo (Kevin et al., 2016) and PlayDrone[1] repositories. The reports were generated using Droidbox (DroidBox, 2016) sandbox. MalDy achieved 99.61%, 99.62%, 93.39% f1-score on the detection task on the previous datasets respectively. Second, we apply MalDy on behavioral reports (20k samples from 15 Win32 malware family) provided by ThreatTrack security[2] (ThreatAnalyzer sandbox). Again, MalDy shows high accuracy on the family attribution task, 94.86% f1-score, under different evaluation settings. Despite the difference between the evaluation datasets, MalDy shows high effectiveness under the same hyper-parameters with minimum overhead during the production, only 0.03 s runtime per one behavioral report on modern machines.

Contributions

- **New Framework:** We propose and explore a data-driven approach on behavioral reports for malware investigation (Section 5). We leverage a word-based security feature engineering (Section 6) instead of the manual specific security features to achieve high portability across different malware platforms and settings.
- **BoW and ML**: We design and implement the proposed framework using the bag of words (BoW) model (Section 5.4) and machine learning (ML) techniques (Section 5.4). The design is inspired from NLP solutions where the word frequency is the key for feature engineering.
- **Application and Evaluation**: We utilise the proposed framework for Android Malware detection using behavioral reports from DroidBox (DroidBox (2016)) sandbox (Section 8). We extensively evaluate the framework on large reference datasets namely, Malgenome (Zhou and Jiang, 2012), Drebin (Arp et al., 2014), and Maldozer (Karbab et al., 2018) (Section 7). To evaluate the portability, we conduct a further evaluation on Win32 Malware reports (Section 8.2) provided by a third-party security company. MalDy shows high accuracy in all the evaluation tasks.

[1] https://archive.org/details/android_apps.
[2] https://www.threattrack.com.

Threat model

We position MalDy as a generic malware investigator tool. MalDy current design considers only behavioral reports. Therefore, MalDy is by design resilient to binary code static analysis issues like packing, compression, and dynamic loading. MalDy performance depends on the quality of the collected reports. The more security information and features are provided about malware samples in the reports the higher MalDy could differentiate malware from benign and attribute to known families. The execution time and the random event generator may have a considerable impact on MalDy because they affect the quality of the behavioral reports. First, the execution time affects the amount of information in the reports. Small execution time may result little information to fingerprint malware. Second, the random event generator may not produce the right events to trigger certain malware behaviors; this loads to false negatives. Anti-emulation techniques, used to evade Dynamic analysis, could be problematic for MalDy framework. However, this issue is related to the choice of the underlying execution environment. First, this problem is less critical for a runtime execution environment because we collect the behavioral reports from real machines (no emulation). This scenario presumes that all the processes are benign and we check for malicious behaviors. Second, the security practitioner could replace the sandbox tool with a resilient alternative since MalDy is agnostic to the underlying execution environment.

Overview

The execution of a binary program (or app) produces textual logs whether in a controlled environment (software sandbox) or a production one. The logs, a sequence of statements, are the result of the app events, this depends on the granularity of the logs. Furthermore, each statement is a sequence of words that give a more granular description on the actual app event. From a security investigation perspective, the app behaviors are summarized in an execution report, which is a sequence of statements and each statement is a sequence of words. We argue that malicious apps have distinguishable behaviors from benign apps and this difference is translated into words in the behavioral report. Also, we argue that similar malicious apps (same malware family) behaviors are translated into similar words.

Nowadays, there are many software sandbox solutions for malware investigations. CWSandbox (2006–2011) is one of the first sandbox solutions for production use. Later, CWSandbox becomes ThreatAnalyzer,[3] owned by ThreatTrack Security. TheatAnalyzer is a sandbox system for Win32 malware, and it produces behavioral reports that cover most of the malware behavior aspects such as a file, network, register access records. Fig. 1 shows a snippet from a behavioral report generated by ThreatAnalyzer. For android malware, we use *DroidBox* (DroidBox (2016)), a well-established sandbox environment based on Android software emulator (Android Emulator, 2016) provided by Google Android SDK (Android SDK, 2016). Running an app may not lead to a sufficient coverage of the executed app. As such, to simulate the user interaction with the apps, we leverage *MonkeyRunner* (MonkeyRunner (2016), which produces random UI actions aiming for a broader execution coverage. However, this makes the app execution non-deterministic since *MonkeyRunner* generates random actions. Fig. 2 shows a snippet from the behavioral report generated using DroidBox.

[3] https://www.threattrack.com/malware-analysis.aspx.

```
<open_key~key="HKEY_LOCAL_MACHINE\Software\Microsoft\
    Windows
NT\CurrentVersion\AppCompatFlags\Layers"/> <open_key
key="HKEY_CURRENT_USER\Software\Microsoft\Windows
NT\CurrentVersion\AppCompatFlags\Layers"/> <open_key
key="HKEY_LOCAL_MACHINE\System\CurrentControlSet\
    Services\
LanmanWorkstation\NetworkProvider"/>
</registry_section> <process_section> <enum_processes
apifunction="Process32First" quantity="84"/> <
    open_process targetpid="308"
desiredaccess="PROCESS_ALL_ACCESS
    PROCESS_CREATE_PROCESS PROCESS_CREATE_THREAD
PROCESS_DUP_HANDLE PROCESS_QUERY_INFORMATION
    PROCESS_SET_INFORMATION
PROCESS_TERMINATE PROCESS_VM_OPERATION
    PROCESS_VM_READ PROCESS_VM_WRITE
PROCESS_SET_SESSIONID PROCESS_SET_QUOTA SYNCHRONIZE"
apifunction="NtOpenProcess" successful="1"/>
```

Fig. 1. Win32 malware behavioral report snippet, ThreatAnalyzer

Notation

- $X = \{X_{build}, X_{test}\}$:X is the global dataset used to build and report MalDy performance in the various tasks. We use build set X_{build} to train and tune the hyper-parameters of MalDy models. The test set X_{test} is use to measure the final performance of MalDy, which is reported in the evaluation section. X is divided randomly and equally to X_{build} (50%) and X_{test} (50%). To build the sub-datasets, we employ the stratified random split on the main dataset X.
- $X_{build} = \{X_{train}, X_{valid}\}$: Build set, X_{build}, is composed of the train set and validation set. It is used to build MalDy ensembles.
- $m_{build} = m_{train} + m_{valid}$: Build size is the total number of reports used to build MalDy. The train set takes 90% of the build set and the rest is used as a validation set.
- $X_{train} = \{(x_0, y_0), (x_1, y_1), .., (x_{m_{train}}, y_{m_{train}})\}$: The train set, X_{train}, is the training dataset of MalDy machine learning models.
- $m_{train} = |X_{train}|$: The size of m_{train} is the number of reports in the train set.
- $X_{valid} = \{((x_0, y_0), (x_1, y_1), .., (x_{m_{valid}}, y_{m_{valid}})\}$: The validation set, X_{valid}, is the dataset used to tune the trained models. We choose

```
"accessedfiles": { "1546331488": "/proc/1006/cmdline
    ","2044518634":
"/data/com.macte.JigsawPuzzle.Romantic/shared_prefs/
    com.apperhand.global.xml",
"296117026":
"/data/com.macte.JigsawPuzzle.Romantic/shared_prefs/
    com.apperhand.global.xml",
"592194838": "/data/data/com.km.installer/
    shared_prefs/TimeInfo.xml",
"956474991": "/proc/992/cmdline"},"apkName": "
    fe3a6f2d4c","closenet":
{},"cryptousage": {},"dataleaks": {},"dexclass": {
    "0.2725639343261719": {
   "path": "/data/app/com.km.installer-1.apk", "type
       ": "dexload"}
```

Fig. 2. Android malware behavioral report snippet (DroidBox (DroidBox, 2016)).

the hyper-parameters that achieve the best scores on the validation set.

- $m_{valid} = |X_{valid}|$: The size of m_{valid} is the number of reports in the validation set.
- (x_i, y_i) : A single record in X is composed of a single report x_i and its label $y_i \in \{+1, -1\}$. The label meaning depends on the investigation task. In the detection task, a positive means malware, and a negative means benign. In the family attribution task, a positive means the sample is part of the current model malware family and a negative is not.
- $X_{test} = \{((x_0, y_0), (x_1, y_1), .., (x_{m_{test}}, y_{m_{test}})\}$: We use X_{test} to compute and report back the final performance results as presented in the evaluation section (Section 8).
- $m_{test} = |X_{test}|$: m_{test} is the size of X_{test} and it represents 50% of the global dataset X.

Methodology

In this section, we present the general approach of MalDy as illustrated in Fig. 3. The section describes the approach based on the chronological order of the building steps.

Behavioral reports generation

MalDy Framework starts from a dataset X of behavioral reports with known labels. We consider two primary sources for such reports based on the collection environment. First, We collect the reports from a software sandbox environment (Willems et al., 2007), in which we execute the binary program, malware or benign, in a controlled system (mostly virtual machines). The main usage of sandboxing in security investigation is to check and analyze the maliciousness of programs. Second, we could collect the behavioral reports from a production system in the form of runtime logs of the running apps. The goal is to investigate the sanity of the apps during the executions i.e. there is no malicious activity. As presented in Section 3, MalDy employs a word-based approach to model the behavioral reports, but it is not clear yet how to use the report's sequence of words in MalDy Framework.

Report vectorization

In this section, we answer the question: how can we model the words in the behavioral report to fit in our classification component? Previous solutions (Chen et al., 2016) (Rieck et al., 2011) select specific features from the behavioral reports by: (i) extract relevant security features (ii) manually inspect and select from these features (Chen et al., 2016). This process involves manual work from the security investigator. Also, it is not scalable since the investigator needs to redo this process manually for each new type of behavioral report. In other words, we are looking for features (words in our case) representation that makes an automatic feature engineering without the intervention of a security expert. For this purpose, MalDy proposes to employ Bag of Word (BoW) NLP model. Specifically, we leverage term frequency-inverse document frequency (TFIDF) (tf-idf, 2016) or feature hashing (trick) (FH) (ShiJames et al., 2009). MalDy has two variants based on the chosen BoW technique whether TFIDF or FH. These techniques generate fixed length vectors from behavioral reports using words' frequencies. TFIDF and FH are presented in more detail in Section 6. At this point, we formulate the reports into features vectors, and we are looking to build classification models.

Build models

MalDy framework utilises supervised machine learning to build its malware investigation models. To this point, MalDy is composed of a set of models, each model has a specific purpose. First, we have the threat detection model that finds out the maliciousness likelihood of a given app from its behavioral report. Afterward, the rest machine learning models aim to investigate individual family threats separately. MalDy uses a model for each possible threat that the investigator is checking for. In our case, we have a malware detection model along with a set of malware family attribution models. In this phase, we build each model separately using X_{build}. All the models are conducting a binary classification to provide the likelihood of a specific threat. In the process of building MalDy models, we evaluate different classification algorithms to compare their performance. Furthermore, we tune up each ML algorithm classification performance under an array of hyper-parameters (different for each ML algorithm). The latter is a completely automatic process; the investigator only needs to provide X_{build}. We train each investigation model on X_{train} and tune its performance on X_{valid} by finding the ML algorithm's optimum hyper-parameters as presented in Algorithm 1. Afterward, we determine the optimum decision threshold for each model using it performance on X_{valid}. At the ends this stage, we have list of optimum models' tuples $Opt = \{< c_0, th_0, params_0 >, < c_1, th_1, params_1 >, .., < c_c, th_c, params_c >\}$, the

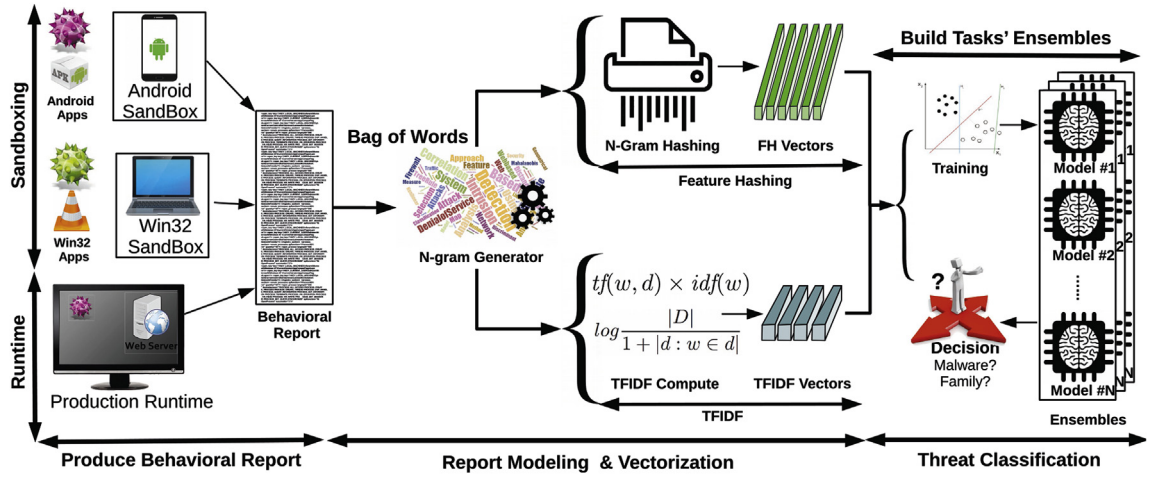


Fig. 3. MalDy methodology overview.

cardinality of list c is number of explored classification algorithms. A tuple $< c_i, th_i, params_i >$ defines the optimum hyper-parameters $params_i$ and decision threshold th_i for ML classification algorithm c_i.

Algorithm 1. Build Models Algorithm.

Input : X_{build}: build set
Output: Opt: optimum models' tuples
$X_{train}, X_{valid} = X_{build}$
for c *in* $MLAlgorithms$ **do**
  score = 0 **for** $params$ *in* $c.params\_array$ **do**
    model = train($alg, X_{train}, params$) ;
    s, th = validate(model, X_{valid}) ;
    if $s > score$ **then**
      ct = $< c, th, params >$;
    end
  end
  Opt.add(ct)
end
return Opt

Ensemble composition

Previously, we discuss the process of building and tuning individual classification model for specific investigation tasks (malware detection, family one threat attribution, family two threat attribution, etc.). In this phase, we construct an ensemble model (outperforms single models) from a set of models generated using the optimum parameters computed previously (Section 5.3). We take each set of optimally trained models $\{(C_1, th_1), (C_2, th_2), .., (C_h, th_h)\}$ for a specific threat investigation task and unify them into an ensemble E. The latter utilises the majority voting mechanism between the individual model's outcomes for a specific investigation task. Equation (1) shows the computation of the final outcome for one ensemble E, where w_i is the weight given for a single model. The current implementation gives equal weights for the ensemble's models. We consider exploring w variations for future work. This phase produces MalDy ensembles, $\{E^1_{Detection}, E^2_{Family1}, E^3_{Family2} \cdots, E^T_{familyJ}\}$, an ensemble for each threat and the outcome is the likelihood this threat to be positive.

$$\widehat{y} = E(x) = sign\left(\sum_{i}^{|E|} w_i C_i(x, th_i)\right)$$
$$\begin{cases} +1 : \sum_i (w_i C_i) \rangle = 0 \\ -1 : \sum_i (w_i C_i) \langle 0 \end{cases} \quad (1)$$

Ensemble prediction process

MalDy prediction process is divided into two phases as depicted in Algorithm 2. First, given a behavioral report, we generate the feature vector x using TFIDF or FH vectorization techniques. Afterward, the detection ensemble $E_{detection}$ checks the maliciousness likelihood of the feature vector x. If the maliciousness detection is positive, we proceed to the family threat attribution. Since the family threat ensembles, $\{E^2_{Family1}, E^3_{Family2} \cdots, E^T_{familyJ}\}$, are independent, we compute the outcomes of each family ensemble E_{family_i}. MalDy flags a malware family threat if and only if the majority voting is above a given voting threshold vth (computed using X_{valid}).

Table 1
Explored machine learning classifiers.

Classifier Category	Classifier Algorithm	Chosen
Tree	CART	✓
	Random Forest	✓
	Extremely Randomized Trees	✓
General	K-Nearest Neighbor (KNN)	✓
	Support Vector Machine (SVM)	✓
	Logistic Regression	✗
	XGBoost	✓

In the case there is no family threat flagged by the family ensembles, MalDy will tag the current sample as an unknown threat. Also, in the case of multiple families are flagged, MalDy will select the family with the highest probability, and provide the security investigator with the flagged families sorted by the likelihood probability. The separation between the family attribution models makes MalDy more flexible to update. Adding a new family threat will need only to train, tune, and calibrate the family model without affecting the rest of the framework ensembles.

Algorithm 2. Prediction Algorithm.

Input : $report$: Report
Output: D: Decision
$E_{detection} = E^1_{Detection}$;
$E_{family} = \{E^2_{Family1}, .., E^T_{familyJ}\}$;
x = Vectorize($report$) ;
detection_result = $E_{detection}(x)$;
if $detection\_result < 0$ **then**
  return detection_result ;
end
for E_{F_i} *in* E_{family} **do**
  family_result = E_{F_i}(x) ;
end
return detection_result, family_result ;

Framework

In this section, we present in more detail the key techniques used in MalDy framework namely, n-grams (Abou-Assaleh et al., 2004), feature hashing (FH), and term frequency inverse document frequency (TFIDF). Furthermore, we present the explored and tuned machine learning algorithms during the models building phase (Section 6.2).

Table 2
Evaluation Datasets. D: DroidBox, T: ThreatAnalyzer

Platform	Dataset	Sandbox	Tag	#Sample/#Family
Android	MalGenome (Zhou and Jiang, 2012)	D	Malware	1k/10
	Drebin (Arp et al., 2014)	D	Malware	5k/10
	Maldozer (Karbab et al., 2018)	D	Malware	20k/20
	AndroZoo (Kevin et al., 2016)	D	Benign	15k/-
	PlayDrone [5]	D	Benign	15k/-
Win32	Malware [6]	T	Malware	20k/15

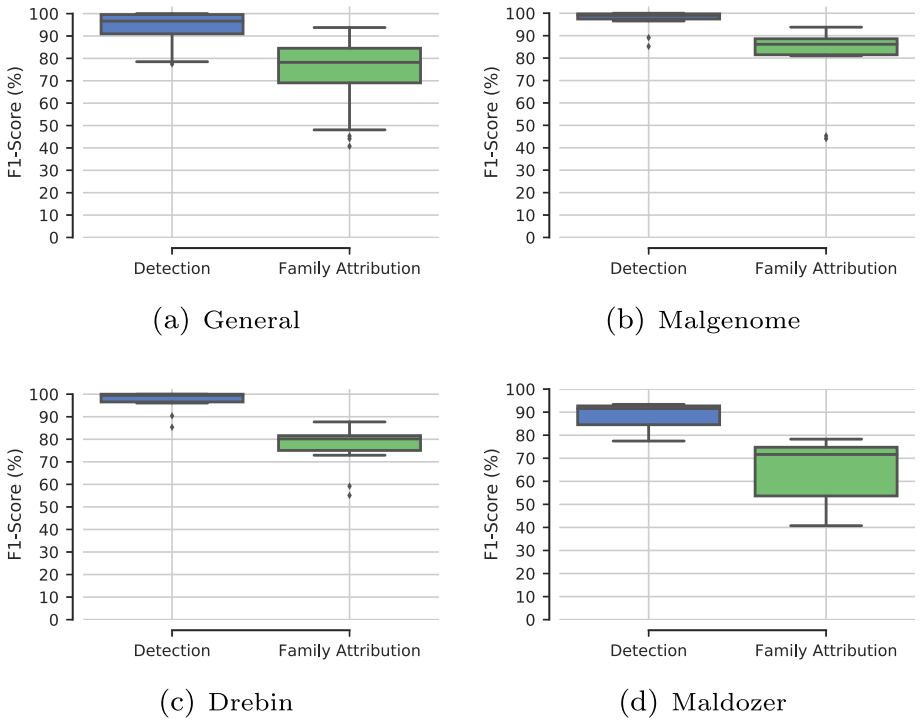


Fig. 4. MalDy effectiveness performance.

Feature engineering

In this section, we describe the components of the MalDy related to the automatic security feature engineering process.

Common N-Gram analysis (CNG)

A key tool in MalDy feature engineering process is the common N-gram analysis (CNG) (Abou-Assaleh et al., 2004) or simply N-gram. N-gram tool has been extensively used in text analyses and natural language processing in general and its applications such as automatic text classification and authorship attribution (Abou-Assaleh et al., 2004). Simply, n-gram computes the contiguous sequences of n items from a large sequence. In the context of MalDy, we compute word n-grams on behavioral reports by counting the word sequences of size n. Notice that the n-grams are extracted using a moving forward window (of size n) by one step and incrementing the counter of the found feature (word sequence in the window) by one. The window size n is a hyper-parameter in MalDy framework. N-gram computation happens simultaneously with the vectorization using FH or TFIDF in the form of a pipeline to prevent computation and memory issues due to the high dimensionality of the n-grams. From a security investigation perspective, n-grams tool can produce distinguishable features between the different variations of an event log compared to single word (1-g) features. The performance of the malware investigation is highly affected by the features generated using n-grams (where $n > 0$). Based on BoW model, MalDy considers the count of unique n-grams as features that will be leveraged by through a pipeline to FH or TFIDF.

Feature hashing

The first approach to vectorize the behavioral reports is to employ feature hashing (FH) (ShiJames et al., 2009) (also called hashing trick) along with n-grams. Feature hashing is a machine learning preprocessing technique to compact an arbitrary number of features into a fixed-length feature vector. The feature hashing algorithm, as described in Algorithm 3, takes as input the report N-grams generator and the target length L of the feature vector. The

Table 3
Tuning effect of tuning of MalDy performance.

	Detection (F1%)			Attribution (F1%)		
	Base	Tuned	Ens	Base	Tuned	Ens
General						
mean	86.06	90.47	94.21	63.42	67.91	73.82
std	6.67	6.71	6.53	15.94	15.92	14.68
min	69.56	73.63	77.48	30.14	34.76	40.75
25%	83.58	88.14	90.97	50.90	55.58	69.07
50%	85.29	89.62	96.63	68.81	73.31	78.21
75%	91.94	96.50	99.58	73.60	78.07	84.52
max	92.81	97.63	100.0	86.09	90.41	93.78
Genome						
mean	88.78	93.23	97.06	71.19	75.67	79.92
std	5.26	5.46	4.80	16.66	16.76	16.81
min	77.46	81.69	85.23	36.10	40.10	44.09
25%	85.21	89.48	97.43	72.36	77.03	81.47
50%	91.82	96.29	99.04	76.66	81.46	86.16
75%	92.13	96.68	99.71	80.72	84.82	88.61
max	92.81	97.63	100.0	86.09	90.41	93.78
Drebin						
mean	88.92	93.34	97.18	65.97	70.37	76.47
std	4.93	4.83	4.65	9.23	9.14	9.82
min	78.36	83.35	85.37	47.75	52.40	55.10
25%	84.95	89.34	96.56	61.67	65.88	75.05
50%	91.60	95.86	99.47	69.62	74.30	80.16
75%	92.25	96.53	100.0	72.68	76.91	81.61
max	92.78	97.55	100.0	76.28	80.54	87.71
Maldozer						
mean	80.48	84.85	88.38	53.11	57.68	65.06
std	6.22	6.20	5.95	16.03	15.99	13.22
min	69.56	73.63	77.48	30.14	34.76	40.75
25%	75.69	80.13	84.56	39.27	43.43	53.65
50%	84.20	88.68	91.58	56.62	61.03	71.65
75%	84.88	89.01	92.72	67.34	71.89	74.78
max	85.68	89.97	93.39	71.17	76.04	78.30

output is a feature vector x_i with a fixed size L. We normalize x_i using the euclidean norm (also called L2 norm). As shown in Formula 2, the euclidean norm is the square root of the sum of the squared vector values.

$$L2Norm(x) = ||x||_2 = \sqrt{x_1^2 + .. + x_n^2} \quad (2)$$

Algorithm 3. Feature Vector Computation.

```
Input  : X_seq: Report Word Sequence,
         L: Feature Vector Length
Output: FH: Feature Hashing Vector
ngrams = Ngram_Generator(X_seq);
FH = new feature_vector[L];
for Item in ngrams do
    H = hash(Item) ;
    feature_index = H mod L ;
    FH[feature_index] += 1 ;
end
// normalization
FH = FH / ||FH||_2 ;
```

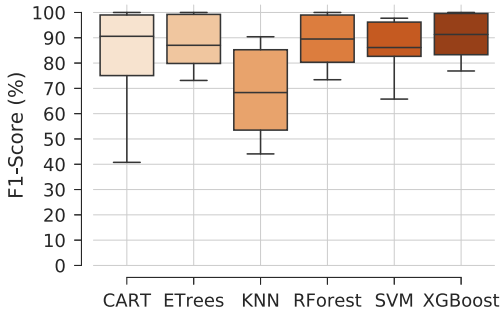
Previous researches (ShiJames et al., 2009; Weinberger et al., 2009) have shown that the hash kernel approximately preserves the vector distance. Also, the computational cost incurred by using the hashing technique for reducing a dimensionality grows logarithmically with the number of samples and groups. Furthermore, it helps to control the length of the compressed vector in an associated feature space. Algorithm 3 illustrates the overall process of computing the compacted feature vector.

Term frequency-inverse document frequency

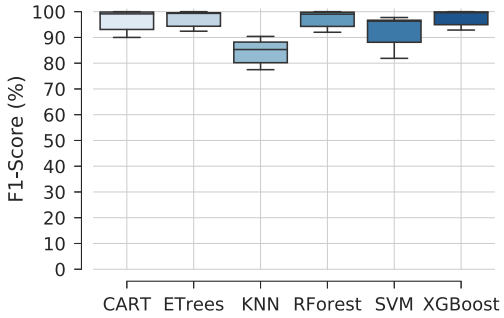
TFIDF (tf-idf, 2016) is the second possible approach for behavioral reports vectorization that also leverages N-grams tool. It is a well-known technique adopted in the fields of *information retrieval* (IR) and *natural language processing* (NLP). It computes feature vectors of input behavioral reports by considering the relative frequency of the n-grams in the individual reports compared to the whole reports dataset. Let $D = \{d_1, d_2, \ldots, d_n\}$ be a set of behavioral documents, where n is the number of reports, and let $d = \{w_1, w_2, \ldots, w_m\}$ be a report, where m is the number of n-grams in d. TFIDF of n-gram w and report d is the product of *term frequency* of w in d and the *inverse document frequency* of w, as shown in Formula 3. The *term frequency* (Formula 4) is the occurrence number of w in d. Finally, the *inverse document frequency* of w (Formula 5) represents the number of documents n divided by the number of documents that contain w in the logarithmic form. Similarly to the feature hashing (Section 6.1.2), we normalize the produced vector using L2

Table 4
Android malware detection.

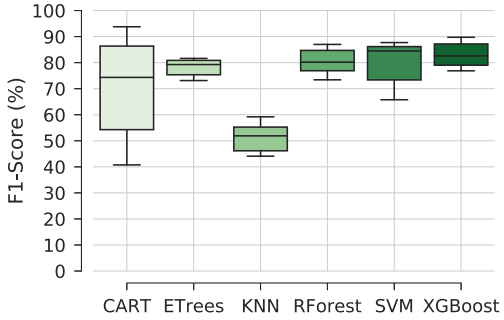
Settings			Attribution F1-Score (%)			Detection F1-Score (%)			
Model	Dataset	Vector	Base	Tuned	Ensemble	Base	Tuned	Ensemble	FPR(%)
CART	Drebin	Hashing	64.93	68.94	72.92	91.55	95.70	99.40	00.64
	Drebin	TFIDF	68.12	72.48	75.76	92.48	96.97	100.0	00.00
	Genome	Hashing	82.59	87.28	89.90	91.79	96.70	98.88	00.68
	Genome	TFIDF	86.09	90.41	93.78	92.25	96.50	100.0	00.00
	Maldozer	Hashing	33.65	38.56	40.75	82.59	87.18	90.00	06.92
	Maldozer	TFIDF	40.14	44.21	48.07	83.92	88.67	91.16	04.91
ETrees	Drebin	Hashing	72.84	77.27	80.41	91.65	95.77	99.54	00.23
	Drebin	TFIDF	71.12	76.12	78.13	92.78	97.55	100.0	00.00
	Genome	Hashing	74.41	79.20	81.63	91.91	96.68	99.14	00.16
	Genome	TFIDF	73.83	78.65	81.02	92.09	96.61	99.57	00.03
	Maldozer	Hashing	65.23	69.34	73.13	84.56	88.70	92.42	06.53
	Maldozer	TFIDF	67.14	71.85	74.42	84.84	88.94	92.74	06.41
KNN	Drebin	Hashing	47.75	52.40	55.10	78.36	83.35	85.37	12.86
	Drebin	TFIDF	51.87	56.53	59.20	82.48	86.57	90.40	05.83
	Genome	Hashing	36.10	40.10	44.09	77.46	81.69	85.23	07.01
	Genome	TFIDF	37.66	42.01	45.31	81.22	85.30	89.13	02.10
	Maldozer	Hashing	41.68	46.67	48.69	69.56	73.63	77.48	26.21
	Maldozer	TFIDF	48.02	52.73	55.31	70.94	75.36	78.51	03.86
RForest	Drebin	Hashing	72.63	76.80	80.46	91.54	95.95	99.12	00.99
	Drebin	TFIDF	72.15	76.40	79.91	92.31	96.62	100.0	00.00
	Genome	Hashing	78.92	83.73	86.12	91.37	95.79	98.95	00.68
	Genome	TFIDF	79.45	83.90	87.00	92.75	97.49	100.0	00.00
	Maldozer	Hashing	66.06	70.72	73.41	84.49	88.96	92.01	07.37
	Maldozer	TFIDF	67.96	72.04	75.89	85.07	89.41	92.72	06.10
SVM	Drebin	Hashing	57.35	61.95	82.92	84.50	89.33	96.08	00.86
	Drebin	TFIDF	63.11	67.19	87.71	85.11	89.35	96.73	01.15
	Genome	Hashing	69.99	74.68	86.08	85.47	89.83	96.54	00.19
	Genome	TFIDF	73.16	77.82	86.20	84.46	88.46	97.73	00.39
	Maldozer	Hashing	30.14	34.76	65.76	72.32	77.12	81.88	15.82
	Maldozer	TFIDF	36.69	41.09	70.18	76.82	81.14	85.46	08.56
XGBoost	Drebin	Hashing	76.28	80.54	**84.01**	92.05	96.50	**99.61**	**00.29**
	Drebin	TFIDF	73.53	77.88	**81.18**	92.23	96.45	**100.0**	**00.00**
	Genome	Hashing	81.80	85.84	**89.75**	91.86	96.09	**99.62**	**00.32**
	Genome	TFIDF	80.36	84.48	**88.24**	92.81	97.63	**100.0**	**00.00**
	Maldozer	Hashing	71.17	76.04	**78.30**	85.68	89.97	**93.39**	**05.86**
	Maldozer	TFIDF	69.51	74.15	**76.87**	85.01	89.16	**92.86**	**06.05**



(a) General



(b) Detection



(c) Attribution

Fig. 5. MalDy effectiveness per machine learning classifier.

norm (see Formula 2. The computation of TFIDF is very scalable, which enhance MalDy efficiency.

$$tf - idf(w,d) = tf(w,d) \times idf(w) \tag{3}$$

$$tf(w,d) = |w_i \in d, d = \{w_1, w_2, \ldots w_n\} : w = w_i| \tag{4}$$

$$idf(w) = log \frac{|D|}{1 + |d : w \in d|} \tag{5}$$

Machine learning algorithms

Table 1 shows the candidate machine learning classification algorithms used in MalDy framework. The candidates represent the most used classification algorithms and come from different learning categories such as tree-based. Also, all these algorithms have efficient public implementations. We chose to exclude the logistic regression from the candidate list due to the superiority of SVM in almost all cases. KNN may consume a lot of memory resources during the production because it needs all the training dataset to be deployed in the production environment. However, we keep KNN in MalDy candidate list because of its unique fast update feature. Updating KNN in a production environment requires only update the train set, and we do not need to retrain the model. This option could be very helpful in certain malware investigation cases. Other ML classifiers is considered as future work.

Evaluation datasets

Table 2 presents the different datasets used to evaluate MalDy framework. We focus on the Android and Win32 platforms to prove the portability of MalDy, and other platforms are considered for a further future research. All the used datasets are publicly available except the Win32 Malware dataset, which is provided by a third-party security vendor. The behavioral reports are generated using DroidBox (DroidBox (2016)) and ThreatAnalayzer[4] for Android and Win32 respectively.

MalDy evaluation

In the section, we evaluate MalDy framework on different datasets and various settings. Specifically, we question the effectiveness of our word-based approach for malware detection and family attribution on Android behavior reports (Section 8.1). We verify the portability and MalDy concept on other platforms (Win32 malware) behavioral reports (Section 8.2). Finally, We measure the efficiency of MalDy under different machine learning classifiers and vectorization techniques (Section 8.3). During the evaluation, we answer some other questions related to the comparison between the vectorization techniques (Section 8.1.2, and the used classifiers in terms of effectiveness and efficiency (Section 8.1.1). Also, we show the effect of train-set's size (Section 8.2.2) and the usage of machine learning ensemble technique (Section 8.1.3) on the framework performance.

Effectiveness

The most important question in this research is: Can MalDy framework detect malware and make family attribution using a word-based model on behavioral reports? In other words, how effective this approach? Fig. 4 shows the detection and the attribution performance under various settings and datasets. The settings are the used classifiers in ML ensembles and their hyper-parameters, as shown in Table 4. Fig. 4(a) depicts the overall performance of MalDy. In the detection, MalDy achieves 90% f1-score (100% maximum and about 80% minimum) in most cases. On the other hand, in the attribution task, MalDy shows over 80% f1-score in the various settings. More granular results for each dataset are shown in Fig. 4(b), (c), and 4(d) for Malgenome (Zhou and Jiang, 2012), Drebin (Arp et al., 2014), and Maldozer (Karbab et al., 2018) datasets respectively. Notice that Fig. 4(a) combines the performance of based (worst), tuned, and ensemble models, and summaries the results in Table 3.

Classifier effect

The results in Fig. 5, Table 3, and the detailed Table 4 confirm the effectiveness of MalDy framework and its word-based approach.

[4] threattrack.com.
[5] https://archive.org/details/android_apps.
[6] https://threattrack.com/.

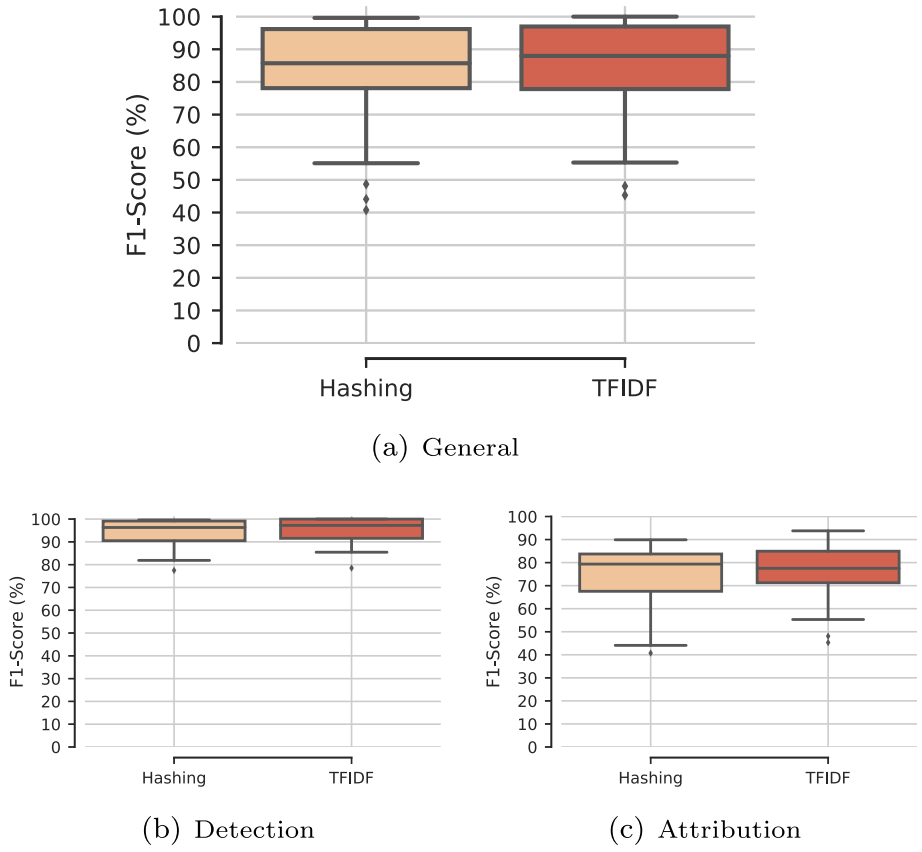


Fig. 6. MalDy effectiveness per vectorization technique.

Fig. 5 presents the effectiveness performance of MalDy using the different classifier for the final ensemble models. Fig. 5(a) shows the combined performance of the detection and attribution in f1-score. All the ensembles achieved a good f1-score, and XGBoost ensemble shows the highest scores. Fig. 5(b) confirms the previous notes for the detection task. Also, Fig. 5(c) presents the malware family attribution scores per ML classifier. More details on the classifiers performance is depicted in Table 4.

Vectorization effect

Fig. 6 shows the effect of vectorization techniques on the detection and the attribution performance. Fig. 6(a) depicts the overall combined performance under the various settings. As depicted in Fig. 6(a), Feature hashing and TFIDF show a very similar performance. In detection task, the vectorization techniques' f1-score is almost identical as presented in Fig. 6(b). We notice a higher overall attribution score using TFIDF compared to FH, as shown in Fig. 6(c). However, we may have cases where FH outperforms TFIDF. For instance, XGBoost achieved a higher attribution score under the feature hashing vectorization, as shown in Table 4.

Tuning effect

Fig. 7 illustrates the effect of tune and ensemble phases on the overall performance of MalDy. In the detection task, as in Fig. 7(a), the ensemble improves the performance by 10% f1-score over the base model. The ensemble is composed of a set of tuned models that already outperform the base model. In the attribution task, the ensemble improves the f1-score by 9%, as shown in Fig. 7(b).

Portability

In this section, we question the portability of the MalDy by

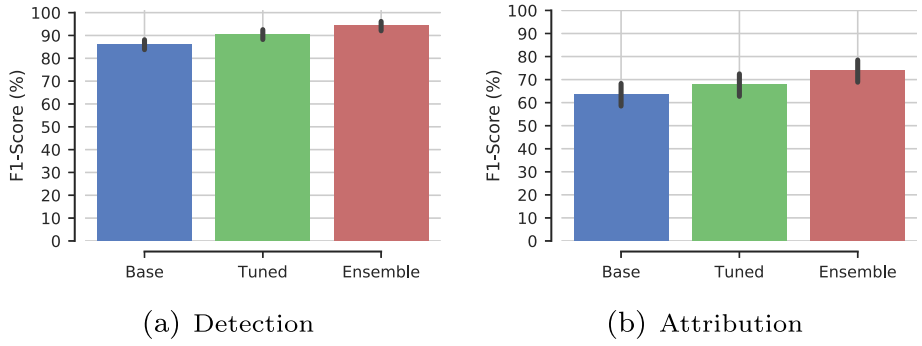


Fig. 7. Effect of MalDy ensemble and tunning on the performance.

applying the framework on a new type of behavioral reports (Section 8.2.1). Also, we investigate the appropriate train-set size for MalDy to achieve a good results (Section 8.2.2). We report only the results of the attribution task on Win32 malware because we lack Win32 benign behavioral reports dataset.

MalDy on Win32 malware

Table 5 presents MalDy attribution performance in terms of f1-score. In contrast with previous results, we trains MalDy models on only 2k (10%) out of 20k reports' dataset (Table 2). The rest of the reports have been used for testing (18k reports, or 80%). Despite that, MalDy achieved high scores that reaches 95%. The results in Table 5 illustrate the portability of MalDy which increases the usefulness of our framework across the different platforms and environments.

MalDy train dataset size

Using Win32 malware dataset (Table 5), we investigate the train-set size hyper-parameter for Maldy to achieve good results. Fig. 8 exhibits the outcome of our analysis for both vectorization techniques and the different classifiers. We notice the high scores of MalDy even with relatively small datasets. The latter is very clear using SVM ensemble, in which it achieved 87% f1-score with only 200 training samples.

Efficiency

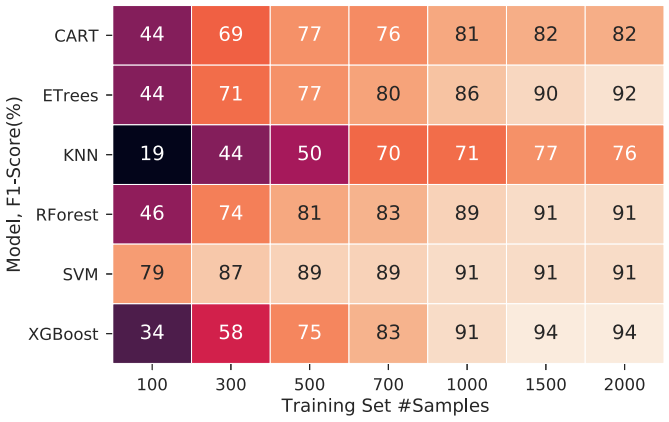
Fig. 9 illustrates the efficiency of MalDy by showing the average runtime require to investigate a behavioral report. The runtime is composed of the preprocessing time and the prediction time. As depicted in Fig. 9, MalDy needs only about 0.03 s for a given report for all the ensembles and the preprocessing settings except for SVM ensemble. The latter requires from 0.2 to 0.5 s (depend on the preprocessing technique) to decide about a given report. Although SVM ensemble needs a small train-set to achieve good results (see Section 8.2.2), it is very expensive in production in terms of runtime. Therefore, the security investigator could customize MalDy framework to suite particular cases priorities. The efficiency experiments have been conducted on Intel(R) Xeon(R) CPU E52630 (128G RAM) machine, where we used only one CPU core.

Conclusion, limitation, and future work

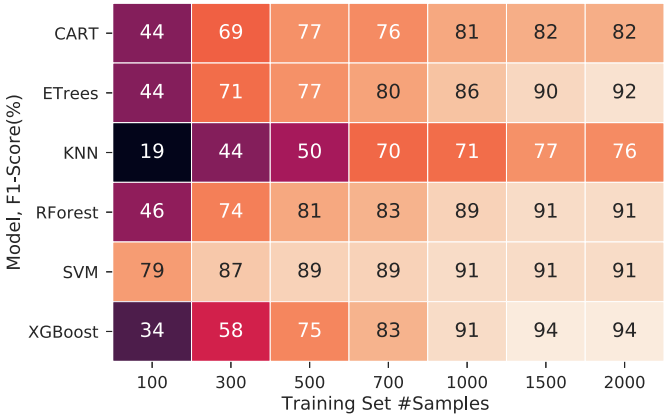
The daily number of malware, that target the well-being of the cyberspace, is increasing exponentially, which overwhelms the security investigators. The diversity of the targeted platforms and architectures compounds the problem by opening new dimensions to the investigation. Behavioral analysis is an important investigation tool to analyze the binary sample and produce behavioral reports. In this work, we propose a portable, effective, and yet efficient investigation framework for malware detection and family attribution. The key concept is to model the behavioral reports using the bag of words model. Afterwards, we leverage advanced NLP and ML techniques to build discriminative machine learning ensembles. MalDy achieves over 94% f1-score in Android detection task on Malgenome, Drebin, and MalDozer datasets and more than 90% in the attribution task. We prove MalDy portability by applying the framework on Win32 malware reports where the framework achieved 94% on the attribution task. MalDy performance depends to the execution environment reporting system; and the quality of the reporting affects its performance. In the current design, MalDy is not able to measure this quality to help the investigator choosing the optimum execution environment. We consider solving this issue for future research.

Table 5
MalDy performance on Win32 malware behavioral report.

Model	Ensemble F1-Score(%)	
	Hashing	TFIDF
CART	82.35	82.74
ETrees	92.62	92.67
KNN	76.48	80.90
RForest	91.90	92.74
SVM	91.97	91.26
XGBoost	**94.86**	**95.43**



(a) Hashing (F1-Score %)



(b) TFIDF (F1-Score %)

Fig. 8. MalDy on Win32 Malware and Effect the training Size of the Performance.

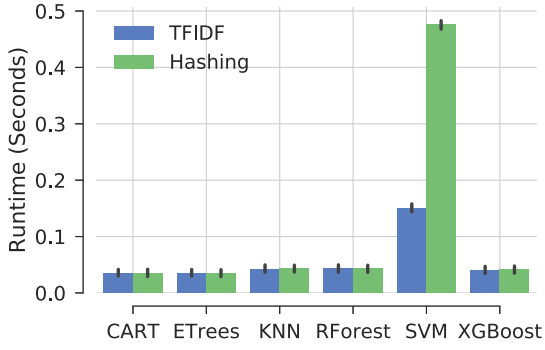


Fig. 9. MalDy efficiency.

Acknowledgements

Many thanks to the anonymous reviewers who provided helpful and insightful comments.

References

Abou-Assaleh, T., Cercone, N., Keselj, V., Sweidan, R., 2004. N-gram-based detection of new malicious code. In: International Computer Software and Applications Conference (COMPSAC).

Alzaylaee, Mohammed K., Yerima, Suleiman Y., Sezer, Sakir, 2016. DynaLog: an automated dynamic analysis framework for characterizing Android applications. CoRR.

Amin, Kharraz, Arshad, Sajjad, Mulliner, Collin, Robertson, William K., Kirda, Engin, 2016. UNVEIL: a large-scale, automated approach to detecting ransomware. In: USENIX Security Symposium.

Android Emulator - https://tinyurl.com/zlngucb, 2016.

Android SDK - https://tinyurl.com/hn8qo9o, 2016.

Arp, Daniel, Spreitzenbarth, Michael, Hubner, Malte, Gascon, Hugo, Riec, Konrad, 2014. DREBIN: effective and explainable detection of android malware in your pocket. In: Symposium on Network and Distributed System Security. NDSS.

Bayer, Ulrich, Comparetti, Paolo Milani, Hlauschek, Clemens, Kruegel, Christopher, Kirda, Engin, 2009a. Scalable, behavior-based malware clustering. In: Symposium on Network and Distributed System Security. NDSS.

Bayer, Ulrich, Comparetti, Paolo Milani, Hlauschek, Clemens, Krügel, Christopher, Kirda, Engin, 2009b. Scalable, behavior-based malware clustering. In: Symposium on Network and Distributed System Security. NDSS.

Chen, Sen, Xue, Minhui, Tang, Zhushou, Xu, Lihua, Zhu, Haojin, 2016. StormDroid: a streaminglized machine learning-based system for detecting android malware. In: ACM Symposium on Information, Computer and Communications Security. ASIACCS.

Chen, Li, Zhang, Mingwei, Yang, Chih-yuan, Sahita, Ravi, 2017. Semi-supervised classification for dynamic android malware detection. In: ACM Conference on Computer and Communications Security. CCS.

DroidBox - https://tinyurl.com/jaruzgr, 2016.

Graziano, Mariano, Canali, Davide, Bilge, Leyla, Lanzi, Andrea, Balzarotti, Davide, 2015. Needles in a haystack: mining information from public dynamic analysis sandboxes for malware intelligence. In: USENIX Security Symposium.

Hu, Xin, Bhatkar, Sandeep, Griffin, Kent, Shin, Kang G., 2013. MutantX-S: scalable malware clustering based on static features. In: USENIX Annual Technical Conference.

Isohara, Takamasa, Takemori, Keisuke, Kubota, Ayumu, 2011. Kernel-based behavior analysis for android malware detection. In: International Conference on Computational Intelligence and Security. CIS.

Karbab, ElMouatez Billah, Debbabi, Mourad, 2018a. Automatic investigation framework for android malware cyber-infrastructures. CoRR.

Karbab, ElMouatez Billah, Debbabi, Mourad, 2018b. ToGather: automatic investigation of android malware cyber-infrastructures. In: International Conference on Availability, Reliability and Security. ARES).

Karbab, ElMouatez Billah, Debbabi, Mourad, Mouheb, Djedjiga, 2016a. Fingerprinting Android packaging: generating DNAs for malware detection. Digit. Invest.

Karbab, ElMouatez Billah, Debbabi, Mourad, Derhab, Abdelouahid, Cypider, Djedjiga Mouheb, 2016b. Building community-based cyber-defense infrastructure for android malware detection. In: ACM Computer Security Applications Conference. ACSAC.

Karbab, ElMouatez Billah, Debbabi, Mourad, Alrabaee, Saed, Mouheb, Djedjiga, 2017a. DySign: dynamic fingerprinting for the automatic detection of android malware. CoRR.

Karbab, ElMouatez Billah, Debbabi, Mourad, Alrabaee, Saed, DySign, Djedjiga Mouheb, 2017b. Dynamic fingerprinting for the automatic detection of android malware. In: International Conference on Malicious and Unwanted Software. MALWARE.

Karbab, ElMouatez Billah, Debbabi, Mourad, Derhab, Abdelouahid, Mouheb, Djedjiga, 2017c. Android malware detection using deep learning on API method sequences. CoRR.

Karbab, ElMouatez Billah, Debbabi, Mourad, Derhab, Abdelouahid, MalDozer, Djedjiga Mouheb, 2018. Automatic framework for android malware detection using deep learning. Digit. Invest.

Kevin, Allix, Bissyandé, Tegawendé F., Klein, Jacques, Le Traon, Yves, 2016. AndroZoo: collecting millions of Android apps for the research community. In: International Conference on Mining Software Repositories. MSR.

Kolbitsch, Clemens, Comparetti, Paolo Milani, Kruegel, Christopher, Kirda, Engin, Zhou, Xiao-yong, Wang, XiaoFeng, 2009. Effective and efficient malware detection at the end host. In: USENIX Security Symposium.

Mariconti, Enrico, Onwuzurike, Lucky, Andriotis, Panagiotis, De Cristofaro, Emiliano, Ross, Gordon, Stringhini, Gianluca, 2017. MaMaDroid: detecting android malware by building Markov chains of behavioral models. In: Symposium on Network and Distributed System Security. NDSS.

Martinelli, Fabio, Mercaldo, Francesco, Saracino, Andrea, Aaron Visaggio, Corrado, 2016. I find your behavior disturbing: static and dynamic app behavioral analysis for detection of Android malware. In: Conference on Privacy, Security and Trust. PST.

MonkeyRunner - https://tinyurl.com/j6ruqkj, 2016.

Nataraj, Lakshmanan, Yegneswaran, Vinod, Porras, Phillip, Zhang, Jian, 2011. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In: ACM Workshop on Security and Artificial Intelligence. AISec.

Paul, Irolla, Filiol, Eric, 2016. Glassbox: Dynamic Analysis Platform for Malware Android Applications on Real Devices. CoRR.

Rieck, Konrad, Trinius, Philipp, Willems, Carsten, Holz, Thorsten, 2011. Automatic analysis of malware behavior using machine learning. J. Comput. Secur.

Severi, Giorgio, Tim Leek, Dolan-gavitt, Brendan, 2018. Malrec: compact full-trace malware recording for retrospective deep analysis. In: Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA.

Sgandurra, Daniele, Muñoz-González, Luis, Mohsen, Rabih, Lupu, Emil C., 2016. Automated dynamic analysis of ransomware: benefits, limitations and use for detection. CoRR.

Shi, Qinfeng, James, Petterson, Dror, Gideon, Langford, John, Smola, Alexander J., Strehl, Alexander L., Vishwanathan, Vishy, 2009. Hash kernels. In: International Conference on Artificial Intelligence and Statistics. AISTATS.

tf-idf - https://tinyurl.com/mcdf46g, 2016.

Wang, Chi-Wei, Winston Shieh, Shiuhpyng, 2015. DROIT: dynamic alternation of dual-level tainting for malware analysis. J. Inf. Sci. Eng.

Weinberger, Kilian, Dasgupta, Anirban, Attenberg, Josh, Langford, John, Alex Smola, 2009. Feature hashing for large scale multitask learning. In: Annual International Conference on Machine Learning. ICML).

Willems, Carsten, Holz, Thorsten, Freiling, Felix, 2007. Toward automated dynamic malware analysis using CWSandbox. In: IEEE Symposium on Security and Privacy. SP.

Wong, Michelle Y., Lie, David, 2016. Intellidroid: a targeted input generator for the dynamic analysis of android malware. In: Symposium on Network and Distributed System Security. NDSS.

Wu, Wen-Chieh, Hung, Shih-Hao, 2014. DroidDolphin: a dynamic android malware detection framework using big data and machine learning. In: Conference on Research in Adaptive and Convergent Systems. RACS.

Zhou, Yajin, Jiang, Xuxian, 2012. Dissecting android malware: characterization and evolution. In: IEEE Symposium on Security and Privacy. SP.