J Comput Virol Hack Tech (2016) 12:59–67
DOI 10.1007/s11416-015-0244-0

ORIGINAL PAPER

Behavior-based features model for malware detection

Hisham Shehata Galal[1] · **Yousef Bassyouni Mahdy**[1] · **Mohammed Ali Atiea**[1]

Received: 12 December 2014 / Accepted: 26 May 2015 / Published online: 4 June 2015

Abstract The sharing of malicious code libraries and techniques over the Internet has vastly increased the release of new malware variants in an unprecedented rate. Malware variants share similar behaviors yet they have different syntactic structure due to the incorporation of many obfuscation and code change techniques such as polymorphism and metamorphism. The different structure of malware variants poses a serious problem to signature-based detection technique, yet their similar exhibited behaviors and actions can be a remarkable feature to detect them by behavior-based techniques. Malware instances also largely depend on API calls provided by the operating system to achieve their malicious tasks. Therefore, behavior-based detection techniques that utilize API calls are promising for the detection of malware variants. In this paper, we propose a behavior-based features model that describes malicious action exhibited by malware instance. To extract the proposed model, we first perform dynamic analysis on a relatively recent malware dataset inside a controlled virtual environment and capture traces of API calls invoked by malware instances. The traces are then generalized into high-level features we refer to as actions. We assessed the viability of actions by various classification algorithms such as decision tree, random forests, and support vector machine. The experimental results demonstrate that the classifiers attain high accuracy and satisfactory results in the detection of malware variants.

✉ Hisham Shehata Galal
Hisham.galal@fci.au.edu.eg

Yousef Bassyouni Mahdy
mahdy@aun.edu.eg

Mohammed Ali Atiea
m_a_atiea@fci.au.edu.eg

[1] Faculty of Computers and Information, Assiut University, Assiut, Egypt

1 Introduction

Malware is a term generalized from two words: malicious and software. It is a piece of code that deliberately fulfills the harmful intent of an attacker. Malware is considered the root cause of many Internet security problems [1]. Typically, it appears in various forms [2] such as Virus, Worm, Trojan Horse, and Bots. Unlike in the past, the creation of a malware today is relatively an easy task due to the availability of malicious code libraries and techniques over the Internet. Malware author can slightly modify existing malicious code and release the new variant into the wild. Consequently, the ever-increasing release rate of malware variants puts a high stress on Anti-Virus (AV) vendors. Malware authors also know that AV vendors analyze thousands of malware instances submitted to them before they can distribute patches to their AV software which gives the malware a window of time to infect and spread flawlessly.

AV vendors receive thousands of new potentially malware samples every day. These samples come from users who found a suspicious program in their systems, and by organizations that use honeypots to capture wild malware samples [3]. In order to effectively analyze thousands of malware samples every day, AV vendors use a tool that utilizes a detection model based on machine learning techniques [4]. This tool is responsible for filtering the submitted malware samples such that only unknown ones are sent to malware analyst for signature extraction.

1.1 Malware detection techniques

There are two malware detection techniques: signature-based and behavior-based [5]. Currently, the signature-based detection techniques are the tool of choice in combat against

malware used by AV software. Of course, modern AV also depends on an heuristic engine component to detect unknown malware instances based on a set of rules [6]. A signature-based detection technique matches a previously generated set of signatures against the suspicious samples. A signature is a sequence of bytes at specific locations within the executable, a regular expression, a hash value of binary data, or any other formats created by malware analyst which should accurately identify malware instances. This approach has at least three major drawbacks [7]. First, the signatures are commonly created by a human; this is an error-prone task and can lead to creating a signature that falsely alarm a benign program. Second, being dependent on previously analyzed malware signatures inherently prevents the detection of unknown malware for which no signatures yet exist. Finally, malware samples use obfuscation techniques such as packing, polymorphism, and metamorphism [8] to evade signature-based detection methods due to the sensitive nature of signatures to the smallest changes in malware binary images.

On the contrary, behavior-based techniques assume malware can be detected by observing the malicious behaviors exhibited by malware during runtime [9]. They do not suffer the limitations found in signature-based techniques since they make their decision after observing malware actions rather than looking for previously known signatures. Therefore, they are effective in the detection of malware variants which share similar behaviors, yet have a different structure. Nonetheless, they suffer from a high false positive rate where a benign program is falsely classified as a malicious program. Additionally, they are evaded by mimicry attacks which involve reconstructing the malicious behavior to appear as legitimate behavior. For example, a Trojan could inject its code into a web-browser application, such that it gains the privileges of the web-browser, hence its communication with the attacker can bypass through the firewall successfully since it appears to be from the web-browser application rather than from another suspect process.

1.2 Malware analysis techniques

The signature-based and behavior-based detection techniques depend on a variety of malware analysis techniques. Malware analysis is the art of dissecting malware to understand how it works, how to identify it, and how to defeat or eliminate it. While malware appears in many different forms, three common techniques exist for malware analysis such as static analysis, dynamic analysis, and hybrid analysis [10].

Static analysis is a passive method. In a sense, malware sample is not executed, but it is inspected using tools such as Disassemblers and Executable analyzers. Static analysis has many advantages. First, it is considered a safe analysis method since malware is not executed and there is a less chance of infecting the analysis machine. Second, the disassembling of malware provides information about all possible execution paths might be taken by the malware. However, a packed malware sample is a quite challenge for static analysis method since it requires a high experience and skills to figure out the unpacking routine in order to extract the real payload [11].

On the other hand, dynamic analysis [9] is considered an active method. It involves executing the malware and monitoring the actions and impacts on the system. Unlike the static analysis, it provides information about the only running execution path. The malware sample is analyzed within a controlled environment such as Virtual Machines (VM) [9]. Dynamic analysis has a considerable time overhead when compared to the static analysis time. Additionally, the increased usage of VM in dynamic analysis inspired malware authors to incorporate additional code to detect the VM presence [12]. Hence, once the malware sample detects VM presence, it can either infect host machine by exploiting vulnerabilities found in VM or changes their execution path to turn into a passive process without any malicious impact on system [12].

In this paper, we aim to provide a behavior-based features model used by the AV filtering tool to cope with the increasing release rate of malware variants. The proposed model describes the malicious actions exhibited by malware during runtime. It is extracted by performing dynamic analysis on a relatively recent malware dataset. We employ API hooking library [13] to log information about API calls and their parameters, however, we further process these API calls into a set of sequences that share a common semantic purpose. After that, the sequences are analyzed by a set of heuristic functions to infer a representative semantic feature which we refer to as *actions*.

The contributions of the paper are as follow:

- We provide a new processing approach on the raw information gathered by API call hooking and produce a set of actions representing the malicious behaviors exhibited.
- We demonstrate the semantic value provided by actions and their insight to help malware analyst.
- We assess actions as a feasible features model and employ various classification techniques to evaluate its accuracy.

The rest of this paper is organized as follows. Section 2 provides a review about the related techniques to extract features for malware detection while we describe the process of extracting actions in Section 3. In Section 4, we evaluate the efficiency and value of actions by various experiments. Finally, conclusions and limitations are presented in Section 5.

2 Related work

In this section, We cover research efforts that claim to detect malware variants. We group the research techniques into three groups. Namely, they are statistical-based, graph-based, and structural-based.

2.1 Statistical-based techniques

Wong and Stamp [8] proposed a technique to detect metamorphic malware based on hidden Markov Model (HMM) and provided a benchmark used in other studies as Canfora et al. [14], Kalbhor et al. [15], Lin and Stamp [16], Musale et al. [17], Shanmugam et al. [18] on metamorphic malware. They analyzed the similarity degree of metamorphism produced by different malware generators such as G2, MPC-GEN, NGVCK and VCL32 by training HMM on the opcode sequences of the metamorphic malware samples.

Annachhatre et al. [19] have used (HMM) analysis to detect certain challenging classes of malware. In their research, they considered the related problem of malware variants classification based on HMMs. More than 8,000 malware variants are then scored against these models and separated into clusters based on the resulting scores. They observed that the clustering results could be used to group the malware samples into their appropriate families with good accuracy.

Faruki et al. [20] used API call-gram to detect malware. API call-gram captures the sequence in which API calls are made in a program. First a call graph is generated from the disassembled instructions of a binary program. This call graph is converted to call-gram. The call-gram becomes the input to a pattern matching engine.

2.2 Graph-based techniques

Park et al. [21] proposed a method to construct a common behavioral graph representing the execution behavior of a family of malware instances. The method generates one common behavioral graph by clustering a set of individual behavioral graphs, which represent kernel objects and their attributes based on system call traces. The resulting common behavioral graph has a common path, called HotPath, which is observed in all the malware instances in the same family. The derived common behavioral graph is highly scalable regardless of new instances added. It is also robust against system call attacks.

Eskandari and Hashemi [22] proposed a technique that uses Control Flow Graph (CFG) to represent the control structure and the semantic aspects of a malware sample. The extracted CFG is annotated by API calls only rather than assembly instructions. This new representation model is referred to as API-CFG. Finally, they converted the resulting graphs into binary feature vectors and trained on them various classification techniques.

2.3 Structural-based techniques

Eskandari et al. [5] presented a novel approach that utilizes machine learning techniques with taking advantages of hybrid analysis methods in order to improve the accuracy of malware analysis procedure while preserving its speed at a point reasonable. They called their approach as HDMAnalyzer that stands for a Hybrid Analyzer based on Data Mining techniques. They used dynamic analysis to extract API call sequences during the execution time of malware sample, meanwhile, they used static analysis to extract Enriched Control Flow Graph (ECFG) which incorporates information about API calls. After the extraction of features, they used a matching engine that combines features obtained by dynamic analysis with corresponding ECFG, and then, each conditional jump receives a label according to the dynamic information. At this point, a machine learning algorithm is employed to build a learning model with the labeled nodes of ECFG. This learning model is used by HDM-Analyzer at scanning time for analyzing unknown executable files.

Islam et al. [23] proposed classification approached based on features extracted from static analysis and dynamic analysis. During the static analysis, they extracted Function Length Frequency (FLF) and Printable String Information (PSI) vectors. The FLF feature is based on counting the number of functions in different length ranges or bins. They derived a vector interpretation of an executable file based on the number of bins chosen and where the function lengths lie across the bins. In PSI, they extracted all printable strings from malware samples to create a global list of strings. Then, they reported for each sample, the count of distinct strings, followed by a binary report on the presence of each string in the global list, where a 1 represents the fact that the string is present and a 0 that it is not. On the other hand, during the dynamic analysis they extracted API features such as API function names and parameters from the log files, then they again construct a global list and use a binary vector where a 1 represents an API function in the global list is called by sample, otherwise they set a 0.

3 The actions model

In this section, we demonstrate the dynamic analysis technique to extract the actions model as outlined in Fig. 1. Each malware sample will pass through three stages: API extraction, Sequence extraction, and Action extractions, which we discuss below.

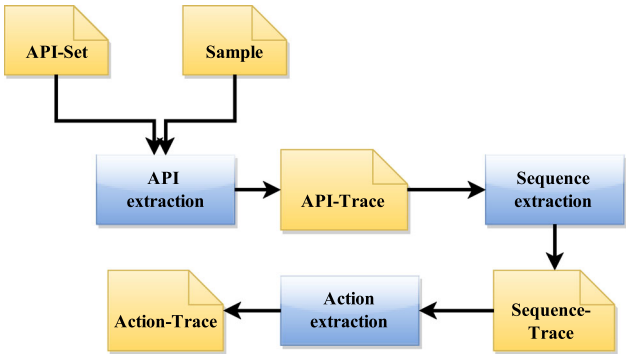

Fig. 1 Dynamic analysis stages to extract actions

3.1 API extraction

The dynamic analysis method of intercepting API calls is known as API hooking that involves manipulating a process such that when a specific, (i.e., to be hooked), API function is called, the execution will be redirected to another code. This code can record API call information to a log file, analyze its parameters or modify them. After that, it may redirect the execution back to the original API code. API hooking introduces a performance penalty in the system. Therefore, only the most frequent API functions imported by malware should be hooked, we refer to them as *API-Set*. It contains API functions from six categories: Process and Threads, Memory Management, Files, Registry, Network, and Internet.

The interception of API call information from a sample is carried out inside a virtual machine (VM) with 32-bit Microsoft Windows seven professional Operating System (OS). An API function can be called by the main module of a process as well as the mapped system modules. It has to be noted, that we only capture API calls invoked by the main module only. Additionally, we also capture API calls invoked by the child processes created by the sample. The following tasks are carried out during this stage:

1. Copy sample to the virtual machine guest operating system.
2. Install hooks on API-Set and execute the sample.
3. Terminate the sample and its associated child processes after five minutes.
4. Output a list of the intercepted API calls along with parameters values.
5. Rollback the virtual machine to a clean snapshot

The output is referred to as *API-Trace*, which is a log file with each line formatted as $(l, a, r, p_1, \ldots p_n)$ where l is the line number, a is the API function name, r is the return value of API, and p_i is the ith parameter's value. Figure 2 shows an example of the output produced by this stage.

```
1, CreateFile, 0x5c, C:\Windows\...
2, RegCreateKey, 0x60, HKLM, Software\...
3, WriteFile, 0x5c, ..
4, RegSetValue, 0x60, test, C:\Windows\...
5, CloseHandle, 0x5c
6, MoveFile, C:\src.exe, C:\Windows\...
7, RegCloseKey, 0x60
```

Fig. 2 An example of API-Trace output where the bold values indicate API handles

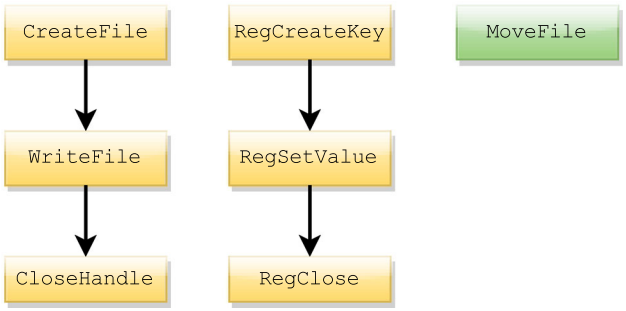

Fig. 3 Sequences of API calls extracted from API-Trace

3.2 Sequence extraction

After the extraction of the API-trace, we identify data-flow dependencies among the intercepted API calls to create sequences of API calls. For this purpose, we divide the API functions into the following four categories:

1. Handle-creation category consists of API functions that open or create an object in Microsoft Windows OS such as `CreateFile`. The return of these API functions is the *Handle* that is used to identify the object.
2. Handle-dependent category consists of API functions that operate on objects and require *Handle* value as input parameter such as `WriteFile`.
3. Handle-release category consists of API functions that free resources associated with the obtained object when there are no more operations to be performed such as `CloseHandle`.
4. Simple category consists of API functions that do not require *Handle* such as `CopyFile`.

We use the handle value to identify data-dependence among API calls of the top three categories while API calls of the last category will be represented individually into separate sequences as shown in Fig. 3. We refer to the collection of all extracted sequences as *Sequence-Trace*.

3.3 Actions extraction

To extract actions from sequence-traces, we use a set of heuristic functions that infer unique actions. The heuristic function selects a sequence of API calls based on their API-category. In Table 1, we list some of the actions produced by heuristic functions. Actions consist of fields with a semantic value that describe the behavior of a sample. The collection of actions for a given sample is referred to as *Action-Trace*, where each action is formatted as $(N, F_i = V_i, ..)$ such that N is the action name, F_i and V_i are the ith field and value, respectively. For example, the output of this stage after processing the extracted sequence-trace above is shown in Fig. 4.

The unique value of the proposed actions model is the high-level insight it gives to malware analyst compared to other techniques based on n-grams such as API n-grams [20] or techniques based on CFG such as API-CFG proposed in [22]. This insight is very helpful for AV malware analyst when it comes to filtering thousands of submitted samples. It gives a brief report on the actions exhibited by the sample during its runtime. Moreover, malware analysts can design new heuristic functions based on their expertise to infer additional complex actions.

4 Experimental results

In this section, we describe the dataset used during experiments and actions extraction. Then, we present an evaluation of the proposed features model performance and provide insights gained from the experiments. Finally, we compare our work with a recent technique discussed in related work.

4.1 Dataset

In this research, we have separate datasets for malware and benign samples. The malware dataset has a diverse number of malware families for different malware types, and each family is represented by an equal number of different variants.

We downloaded 9993 samples from *VirusSign* [24] in the period from October 14, 2013 to March 2, 2014. However, the obtained samples are labeled by MD5 hash values which do not provide any information about their malware family. We scanned the samples by AV software to identify their malware family. Then, we selected 2000 samples for a 50 different malware family such that each malware family is represented by 40 malware variants. A partial list of malware families along with their type is given in Table 2. On

Table 1 Partial listing of action examples

Action	Field	Description
Process Creation	`Name`	Name of the created process
	`IsDropped`	Set to true if the process image is created by the running sample
	`IsModified`	Set to true if the created process is hooked or its memory is modified
Process Termination	`Name`	Name of the terminated process
	`IsSystem`	Set to true if the terminated process is located inside Windows directory
	`IsSecurity`	Set to true if the killed process is a security program such as anti-virus or firewalls
Remote Threads	`Name`	Name of the process with injected remote threads
	`IsInjected`	Set to true if the remote thread is used to load DLL
	`IsDropped`	Set to true if the injected DLL is created by the running sample
Registry	`Value`	Path of the registry value accessed by the sample
	`Data`	The data set in or retrieved from registry value
	`Operation`	Type of operation either as Set or as Get
File 1	`Directory`	Path of the directory containing the file
	`Type`	For example, Executable, DLL, Binary, and Text
	`Mode`	Access mode such as Open and Create
	`Operation`	Such as Write, Read, etc
File 2	`Source`	Path of the source directory
	`Destination`	Path of the destination directory
	`Type`	For example, Executable, DLL, Binary, and Text
	`Operation`	Such as Copy, Move, Delete, Rename etc

```
{File, Directory = C:\Windows\..., Type = Executable, Mode = Create, Operation = Write}
{Registry, Value = HKLM\Software\Microsoft..., Data = C:\Windows\..., Operation = Set}
{File 2, Source = C:\ , Destination= C:\Windows\..., Operation = Move, Type = Executable}
```

Fig. 4 Action-Trace output produced by heuristic functions

Table 2 Part of malware families in dataset

Type	Malware Family		
Backdoor	Androm	Bifrose	DarkKomet
	Hupigon	Kelihos	Zegost
Trojan	Buzus	Graftor	Sirefef
	Urusay	Vundo	ZBot
Virus	Alman	Chir	Elkern
	Jadtre	Neshta	Sality
Worm	Ratab	Allaple	Darkbot
	Fesber	Mydoom	Vofbus

the other hand, the benign dataset contains 2000 sample represented by executables found in the installation directory of Microsoft Windows 7 operating system, Microsoft Office, common web-browsers, and other utilities.

4.2 Performance metric

In the experiments, we employ three supervised classification algorithms Random Forest (RF) [25], Support Vector Machine (SVM) [26], and Decision Tree (DT) [27] to evaluate the proposed features model.

A decision tree is a widely used inductive learning method. It is often used for approximating discrete-valued functions. A decision tree is a rooted tree with internal nodes corresponding to attributes and the leaf nodes corresponding to class labels.

A random forest is an ensemble learning method, that operate by constructing a multitude of decision trees at training time and outputting the class that receives highest votes from them.

A support vector machine constructs a hyperplane or set of hyperplanes in a high-dimensional space. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data point of any class (so-called functional margin).

For performance comparison between the different classifiers, we define the following terms:

1. True positive (TP) is the number of predicted malware samples correctly classified as malicious.
2. True negative (TN) is the number of predicted benign samples correctly classified as benign.
3. False positive (FP) is the number of predicted benign samples incorrectly classified as malicious.
4. False negative (FN) is the number of predicted malware incorrectly classified as benign.

These terms are used to define four performance comparison criteria between DT, RF, and SVM. The first criterion is *Sensitivity* which measures the proportion of actual positives, (malware samples), which are correctly identified.

$$Sensitivity = \frac{TP}{TP + FN} \tag{1}$$

The second criterion is *Specificity* which measures the proportion of negatives, (benign samples), which are correctly identified.

$$Specificity = \frac{TN}{TN + FP} \tag{2}$$

The third criterion is *Accuracy* which measures the overall accuracy to classify malware and benign samples correctly.

$$Accuracy = \frac{TP + TN}{TN + TP + FN + FP} \tag{3}$$

The fourth criterion is Area under the ROC curve or commonly referred to as (*AUC*). The ROC curve is a graphical plot that illustrates the performance of a binary classifier system as its discrimination threshold is varied. The curve is created by plotting the true positive rate (*sensitivity*) against the false positive rate (1-*specificity*) at various threshold settings. An AUC of 1.0 indicates ideal separation (i.e., there exists a threshold which divides the samples between the two classes correctly), while an AUC of 0.5 represents a worthless classifier.

4.3 Model evaluation

To evaluate actions as features for malware classification, we extract actions for all samples in benign and malware dataset and accumulate them into the global *Action-List*. After that, we represent every sample as a binary vector that has the same length as the action-list, such that the binary feature f_i is set to 1, if the sample has exhibited the $action_i$, otherwise, f_i is set to 0. This vector will serve as the input to various classification algorithms.

Table 3 Classifiers results for the proposed features model

Algorithm	Sensitivity	Specificity	Accuracy	AUC
DT	97.3 %	96.53 %	97.19 %	97.65 %
RF	97.19 %	96.35 %	96.84 %	99.48 %
SVM	92.28 %	96.35 %	93.98 %	98.55 %

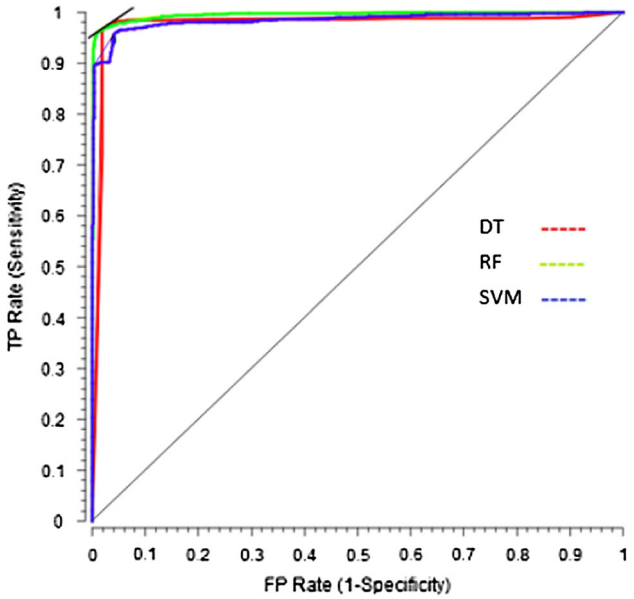

Fig. 5 ROC curves of RF, SVM, and DT classifiers

We used *Orange* machine learning toolbox [28] to train and test classifiers. It contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization. Figure 6 outlines the workflow between the different widgets employed to train and test the DT, RF, and SVM classifiers. First, we loaded the binary vectors of all samples into a data table. Then, we filtered the features by selecting only the features with information gain above a certain threshold. The threshold value is obtained after carrying out several experiments to achieve highest classification accuracy, we used 0.03 as a threshold value. The feature selection resulted in a reduced data that we fed to DT, RF, and SVM classifiers. The defaults parameters for classifiers are used without changes. The 10-cross validation technique is used to validate the accuracy achieved by classifiers. After training and testing classifiers, we report the evaluation results in Table 3, and the ROC curves for each classifier is shown in Fig. 5.

During the experiments, we investigated the reason for false positives and false negatives. An example of false positive is the *Virtual-CD* setup program. It had many actions similar to those exhibited by malware samples. For example, some of these actions were installing a service driver in a way similar to rootkit behaviors, in addition to setting an auto-start extensibility point in the registry.

On the other hand, one of the *IRCBot* malware samples could escape the detection due to inherent drawbacks of dynamic analysis. Precisely, It did not show its malicious actions due to the detection of virtual machine artifacts.

We have implemented the technique in [19] with 50 hidden Markov Model (HMM) for each malware family in our malware dataset. The opcode sequences are extracted using *ObjDump* tool. We used the following parameters, the number of states $N = 2$, and the number of different symbols $S = 827$, (i.e, number of different opcodes), the number of $iterations = 800$, and $likelihood = 0.001$. We used a vector that hold scores from all HMM to be used as features. We have noticed that, since we trained HMM only on malware

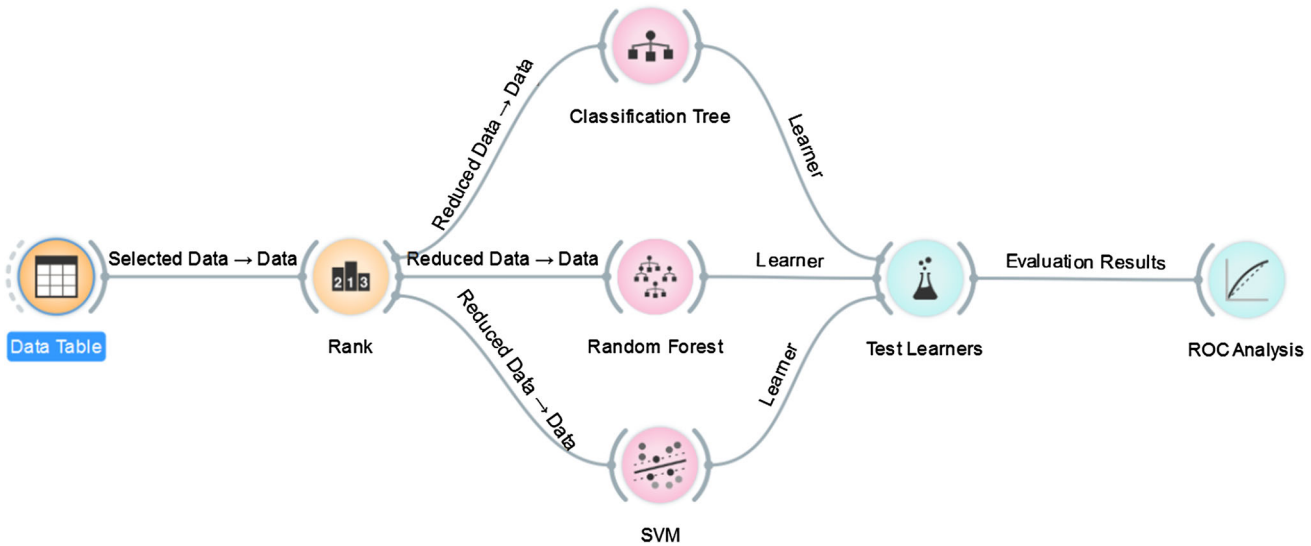

Fig. 6 Work-flow between widgets of Orange toolbox to train and test RF, SVM, and DT and perform ROC analysis

Table 4 Classification results for HMM based features model [19]

Algorithm	Sensitivity	Specificity	Accuracy	AUC
DT	97.6 %	97.63 %	96.89 %	97.72 %
RF	97.1 %	96.3 %	96.14 %	97.68 %
SVM	93.17 %	95.45 %	94.8 %	95.55 %

families, we found that benign samples have low scores on all models while malware samples have at least one model that produced a high score. The evaluation results are given in Table 4.

While the results of this state-of-the-art technique and our proposed features model are comparable in terms of performance, there are some key advantages of using actions. First, actions provide helpful semantic insight of malware behavior to malware researcher. Second, the proposed technique to extract actions is extensible as it relies on a set of heuristic functions which can be improved by the technical expertise of malware researcher. This leads to extracting complex and new actions exhibited by evolved malware samples. Finally, actions are easier to understand by non-experts than statistical HMM as indicated previously in Fig 4.

5 Conclusion

In this paper, we proposed a behavior-based features model to describe the malicious actions exhibited by malware during the runtime. The proposed features model is referred to as actions; it is created by performing dynamic analysis over a relatively recent malware dataset. In dynamic analysis, we used API hooking technique to trace API calls invoked by malware sample, then we further process the traced API calls into groups of semantically dependent API calls. The API sequences are further processed by a set of heuristic functions that extract the actions.

The unique value of actions is the high-level insight it gives to malware analyst. More precisely, actions describe the malicious behaviors of a sample with a better semantic value than API n-gram based techniques. Additionally, malware analysts can utilize their technical knowledge by adding more heuristic functions to extract additional actions. During the experiments, we assessed the actions as features for malware and benign programs classification. Based on the experimental results, the classifiers achieved high classification accuracy rates.

5.1 Limitations

The technique used to extract the actions has some limitations inherent from dynamic analysis methods, which observe a partial behavior of malware sample. In other words, it is not suitable for malware samples that depend on external events such as receiving a remote command or depending on specific time to trigger their malicious actions. Additionally, the dynamic analysis fails against malware sample that checks for the existence of virtual machine artifacts [12] and become a passive process or simply terminate itself resulting in clean actions. Finally, the extraction technique is not fully automatic since it requires malware analyst support to create the set of heuristic functions.

5.2 Future work

The future work on the actions includes refining them by utilizing data flow dependence, which should provide more insight into malware behavior. Additionally, the tool which extracts API call information can be modified to work outside the virtual machine environment; this approach provides a more robust solution against malware samples that check for the existence of mentioned tool.

References

1. Fossi, M., Egan, G., Haley, K., Johnson, E., Mack, T., Adams, T., Blackbird, J., Low, M.K., Mazurek, D., McKinney, D., et al.: Symantec internet security threat report trends for 2010, vol. 16 (2011)
2. Gennari, J., French, D.: Defining malware families based on analyst insights. In: Technologies for Homeland Security (HST), 2011 IEEE International Conference on IEEE, pp. 396–401 (2011)
3. Mairh, A., Barik, D., Verma, K., Jena, D.: Honeypot in network security: a survey. In: Proceedings of the 2011 International Conference on Communication, Computing & Security ACM, pp. 600–605 (2011)
4. Kiemt, H., Thuy, N.T., Quang, T.M.N.: A machine learning approach to anti-virus system (artificial intelligence i). IPSJ SIG Notes. ICS **2004**(125), 61–65 (2004)
5. Eskandari, M., Khorshidpour, Z., Hashemi, S.: Hdm-analyser: a hybrid analysis approach based on data mining techniques for malware detection. J. Comput. Virol. Hacking Tech. **9**(2), 77–93 (2013)
6. Kaspersky. Heuristic analysis in anti-virus. http://support.kaspersky.com/8641 (2013). Accessed in 1 April 2015
7. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Twenty-third annual IEEE Computer security applications conference, 2007. ACSAC 2007, pp. 421–430 (2007)
8. Wong, W., Stamp, M.: Hunting for metamorphic engines. J. Comput. Virol. **2**(3), 211–229 (2006)
9. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. (CSUR) **44**(2), 6 (2012)
10. Sikorski, M., Honig, A.: Practical malware analysis: the hands-on guide to dissecting malicious software. No Starch Press (2012)
11. Cesare, S., Xiang, Y., Zhou, Wanlei: Malwise&# x2014; an effective and efficient classification system for packed and polymorphic malware. IEEE Trans. Comput. **62**(6), 1193–1206 (2013)
12. Lindorfer, M., Kolbitsch, C., Comparetti, P.M.: Detecting environment-sensitive malware. In: Recent Advances in Intrusion Detection, pp. 338–357. Springer (2011)

13. Nektra Advanced Computing. Deviare api hook. http://www.nektra.com/products/deviare-api-hook-windows/ (2015). Acces sed in 1 April 2015
14. Canfora, G.: Antonio Niccolò Iannaccone, and Corrado Aaron Visaggio. Static analysis for the detection of metamorphic computer viruses using repeated-instructions counting heuristics. J. Comput. Virol. Hacking Tech. **10**(1), 11–27 (2014)
15. Kalbhor, A., Austin, T.H., Filiol, E., Josse, S., Mark, S.: Dueling hidden markov models for virus analysis. J. Comput. Virol. Hacking Tech. **11**, 1–16 (2014)
16. Lin, D., Stamp, M.: Hunting for undetectable metamorphic viruses. J. Comput. Virol. **7**(3), 201–214 (2011)
17. Musale, M., Austin, T.H., Stamp, M.: Hunting for metamorphic javascript malware. J. Comput. Virol. Hacking Tech. 1–14 (2014)
18. Shanmugam, G., Low, R.M., Stamp, M.: Simple substitution distance and metamorphic detection. J. Comput. Virol. Hacking Tech. **9**(3), 159–170 (2013)
19. Annachhatre, C., Austin, T.H., Stamp, M.: Hidden markov models for malware classification. J. Comput. Virol. Hacking Tech. 1–15 (2014)
20. Faruki, P., Laxmi, V., Gaur, M.S., Vinod, P.: Mining control flow graph as api call-grams to detect portable executable malware. In Proceedings of the Fifth International Conference on Security of Information and Networks ACM, pp. 130–137 (2012)
21. Park, Y., Reeves, D.S., Stamp, M.: Deriving common malware behavior through graph clustering. Comput. Secur. **39**, 419–430 (2013)
22. Eskandari, M., Hashemi, Sattar: A graph mining approach for detecting unknown malwares. J. Vis. Lang. Comput. **23**(3), 154–162 (2012)
23. Islam, R., Tian, R., Batten, L.M., Versteeg, S.: Classification of malware based on integrated static and dynamic features. J. Netw. Comput. Appl. **36**(2), 646–656 (2013)
24. VirusSign. Malware research and data center. http://www.VirusSign.com (2015). Accessed in 1 April 2015
25. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
26. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. **20**(3), 273–297 (1995)
27. Safavian, S.R., Landgrebe, D.: A survey of decision tree classifier methodology (1990)
28. Demšar, J., Curk, T., Erjavec, A., Gorup, Č., Hočevar, T., Milutinovič, M., Možina, M., Polajnar, M., Toplak, M., Starič, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M., Zupan, B.: Orange: Data mining toolbox in python. J. Mach. Learn. Res. **14**, 2349–2353 (2013)