IET Information Security

Research Article

IET Journals
The Institution of Engineering and Technology

Malware classification based on API calls and behaviour analysis

ISSN 1751-8709
Received on 1st September 2017
Accepted on 27th September 2017
doi: 10.1049/iet-ifs.2017.0430
www.ietdl.org

Abdurrahman Pektaş[1] ✉, Tankut Acarman[1]

[1]Computer Engineering Department, Galatasaray University, Çırağan Cad. No:36, 34349, Ortakoy, Istanbul, Turkey
✉ *E-mail: apektas@yandex.com*

Abstract: This study presents the runtime behaviour-based classification procedure for Windows malware. Runtime behaviours are extracted with a particular focus on the determination of a malicious sequence of application programming interface (API) calls in addition to the file, network and registry activities. Mining and searching n-gram over API call sequences is introduced to discover episodes representing behaviour-based features of a malware. Voting Experts algorithm is used to extract malicious API patterns over API calls. The classification model is built by applying online machine learning algorithms and compared with the baseline classifiers. The model is trained and tested with a fairly large set of 17,400 malware samples belonging to 60 distinct families and 532 benign samples. The malware classification accuracy is reached at 98%.

1 Introduction

The proliferation of the runtime packer and obfuscation easily facilitates the creation of behaviourally identical but statically different malware samples [1]. In other words, the majority of new malware samples can be deployed as the variant of the previously known samples. According to the reports published by software security groups in 2016, more than 430 million new unique pieces of malware were detected in 2015 with an increase of 36% from the previous year, and a new zero-day vulnerability was discovered at each week on average with a two times higher release frequency in comparison with the previous year [2]. In this study, file system, network, and registry activities observed along the execution traces and patterns identified by applying data mining over application programming interface (API) function call sequences are used to represent behaviour-based features of a malware. An invariant compact representation of the general behaviour of a malware is rather complicated by many possible modifications injected by malware writers in the malware code. However, The Windows API call sequence is not changed by obfuscation techniques and can be exploited for behaviour analysis of an executable file. The amount of information contained in the critical low-level system call sequence by knowing the presence or absence of malign patterns and their frequencies, for instance, API calls encoded into characters belonging to a representative malware family, is used to detect and classify unknown malware samples.

Malware classification systems can be grouped into two distinct categories based on the feature set. The first group examines an executable file without running it on the system, and static features are used. The second group observes the activities of the suspicious file while allowing their execution in a sandbox environment. Namely, file, registry, network and process activities are tracked and reported. Dynamic features are the result of an integrated scenario provided to a malware sample to be deployed and to execute its malicious tasks. After extracting the base feature set, machine learning or data mining tools can be applied for classification purposes.

Static features are provided by the different behavioural representation of the executable. For example, low-level static features such as strings, byte-sequences, and opcode sequences can be extracted by disassembling a malware, address table can be imported for classification (see for instance [3]). The information gathered from API call sequences, control flow graph and functions composing a malware can be used as high-level static features [4]. However, static features are sensitive to obfuscation techniques, for instance, the structure of the opcode sequences can be drastically changed and a malware variant cannot be detected or its detection becomes very complicated. Behavioural features, particularly API call sequences, have attracted a lot of attention towards detection and classification of a malware. Features can be extracted from the API call sequences either by static or dynamic analysis. The API call sequences provided by static analysis need to be discovered by following multiple paths of the malware code, and a meaningful representation of the extracted information is rather difficult and time-consuming. Extraction of features while the malicious code is running, i.e. dynamic analysis, provides more reliable and meaningful results. However, dynamic analysis can be evaded when a virtualised environment is detected or avoided by a malware's pre-coded execution scheduling and inactivity during analysis. Unlike the static approach, a single path is discovered. Dynamic analysis is useful and reliable when features are extracted and a classification model is built by using a large set of malware samples belonging to representative families.

For dynamic analysis of the API call sequences, a pioneering work by Forrest and Longstaff [5] was the first to profile benign and malign UNIX processes based on the characteristic fixed size contiguous sequences of system calls (so-called n-gram). System call sequences of lengths 5, 6, and 11 are used to determine whether a process is normal or abnormal. However, the creation of sequences by using fixed size n-gram modelling leads to the misplacement of system calls and the extracted base feature set cannot sufficiently cover the malicious behaviours. In our presented API mining approach, we mitigate this limitation by extracting variable length n-gram features. In [6], malware behaviour extraction based on API calls is presented. To obtain ordinary malicious behaviours such as self-delete, remote process injection, gain persistency, etc., 236 known malware samples are analysed. Basic operations leading to a change in the system status are considered as the malicious behaviour features. The maliciousness of a sample is scored according to the count of malicious behaviours. In [7], the API call frequencies are extracted by using dynamic analysis for a given suite of 100 malicious behaviour patterns and 24 critical API calls. However, since the proposed system barely depends on the frequency of the API calls, the system is unable to detect samples that invoke excessive and redundant API calls. In [8], detection of a malware variant in the Windows Operating System (OS) is elaborated. The API call sequence is extracted from a portable executable file. A similarity analysis between known malware samples and unknown malware variants is performed. However, since API invocation cannot be extracted accurately from packed malware, level in extraction accuracy is not guaranteed. In [9], information flows are analysed to detect Android malware. The structure of information flows is scrutinised to discover behavioural patterns exhibited by Android

applications. The n-gram features are extracted from the API call sequences to identify both distinct and common behaviour patterns. Then, a support vector machine (SVM) algorithm is applied to a balanced set of 3,899 benign and malicious samples, 97.6% true positive and 9.0% false positive rates are reached. In [10], the API calls are abstracted into their packages (e.g. java.lang, java.io) or their sources (e.g. android, java). Then, they are modelled as a Markov chain to extract Android application behavioural features. Experiments were conducted with a large dataset including almost 44 K applications (8.5 K benign and 35.5 K malware samples) captured for the duration of 6 years and 99% *F*-measure is measured.

In [11], a collection of featured malicious behaviours and outcomes with representative malware samples during run-time execution is presented and then, based on the abstraction of explored multipaths and triggered malicious behaviours, the idea of building a malicious feature matrix is given. Although the identification steps and architecture of this approach are well presented, run-time malicious behaviour features, categories, outcomes are not given, which limits the usability of the results. The SVM algorithm is used to classify a small set of 100 malware variants to two representative malware families, namely, W32.Ramnit and W32.Sality with an accuracy of 84 and 80%, respectively. The number of samples and families is not sufficient and other classification algorithms are not evaluated towards a reliable evaluation of the level of accuracy. In [12], regular expression-based behaviour representation is introduced by characterising high-level patterns of *object-accessing* mappings. For example, the patterns of 'CreateFile, WriteFile (one or more times) and CloseFile' are mapped to the higher-level behaviour of *file-writing*. In this work, malware samples are executed on a Windows XP SP2 QEMU emulator and behaviour traces are collected via SSDT hooking technique [13]. Then, regular expressions are built according to *object-accessing* mappings to characterise software. Finally, a sequence similarity metric is applied to make a decision whether a sample is malign or benign. An automated malware classification system using solely network-related behavioural information is presented in [14]. Features are extracted in the form of graph size, a number of nodes including domain name system (DNS), hypertext transfer protocol (HTTP), simple mail transfer protocol (SMTP) connections, and a behavioural graph is created for each sample illustrating the network activity and network connection dependencies. Classification accuracy using J48 decision tree in WEKA [15] tool is reached at 94.5%. In [16], 26 network related events such as DNS query types, IP address, connection protocols are mapped into an alphabet. The n-gram feature extraction is used to generate a feature vector. SVM, decision tree, and the k-nearest neighbour (K-NN) are applied to evaluate a dataset constituted by 2,700 malware samples belonging to three malware families. Decision tree classifier reaches an accuracy level of 80%. Although the system is very effective at modelling and identifying malware samples that generate network activities, samples not creating network connections cannot be detected (see also [14]). We consider features derived from not only network but also registry, file system, process and service interaction.

In [17], behaviours are described by means of special representation called Malware Instruction Set, which has been inspired by instruction sets used in a central processing unit (CPU). In this representation, the behaviour of a sample is characterised with a sequence of instructions, and the SVM algorithm is used to classify malware samples. In [18], based on the interactions between a malware and security critical OS resources, a set of high-level activities is extracted. A limited feature space is chosen to reduce computational requirement and less memory usage. Although bounded-feature space decreases the response time of the system and requires less computational resources to process 5,000 malware samples, some important behavioural information is discarded and classification results are not reliable. In [19], malware samples are executed in a virtual environment and the API call sequences are traced during runtime by using user-space hooking library called Detour [20]. The common API call sequence patterns are extracted by applying the longest common sub-sequences (LCS) algorithm. 2,727 kinds of API are categorised into 26 groups in compliance with MSDN library. Classification accuracy is reached at 99% with a dataset of 6,910 malware and 34 benign samples. The main limitation of this method is that computing LCS is non-deterministic polynomial time-hard problem, therefore computational complexity is high, which requires much more computational resources and time.

In these studies, there is an inevitable trade-off between 'curse of dimensionality' and 'poor interpret-ability'. On the one hand, if the feature-space is increased, the analysis framework becomes infeasible. On the other hand, if the feature space is reduced, important information can be ignored, and the classification results are not reliable. Further concerns exist about the usage of some single behaviour such as network connection, or observation of high-level patterns, API calls, and similarity scoring based on their counts in comparison with a benign or a malign sample is not a reliable classification scheme for an evolving malware.

In this study, we exploit all the aspects of extracted information about behaviour including Windows API call sequences and changes made to the OS. Our proposed classification system uses the Voting Experts algorithm to identify episode boundaries efficiently and accurately. The identification of episode boundaries is exploited to mine and search n-gram API call sequences and to discover the malicious behaviour of an executable. Additional information is extracted by tracking state changes made by a malware to the OS. Overall, our proposed method fuses both the results about mining and searching n-gram API call sequences and OS state changes. The malware API call sequences and malware's interactions with the OS are complementary and exploitation of both observations increases the level of accuracy of malware classification.

2 Methodology and implementation

This section presents our proposed malware classification system stages with a particular focus on the behavioural feature extraction and discovery of meaningful API sequences.

2.1 Behaviour analysis

In this study, we deploy VirMon to extract Windows kernel-level notification routines as the underlying automated dynamic platform and we use Cuckoo to extract API call sequences.

In VirMon, any system change occurring on the analysis machine is monitored through Windows kernel-level notification routines [21]. During the analysis, the state changes of OS resources such as file, registry, process/thread, network activities and intrusion detection system (IDS) alerts are logged into a report file. The automatic analysis capability of VirMon enables analysis of a lot of malware samples without user's intervention. Some critical API call sets can expose the aim of software and call behaviours may be tracked during dynamic analysis. These calls can be stated as the most useful and reliable features to classify a malware. As VirMon does not extract API call sequences, an open source Cuckoo sandbox is used to fulfil this functionality gap [22].

Cuckoo reports artefacts with its agents while executing a file in an isolated environment, and provides function calls of the file under analysis via hooking. In this study, we modified default Cuckoo configuration in order to increase the number of concurrent analysis.

We separated network traffic by defining virtual network interfaces (i.e. virtual local area network interfaces). For instance, an analysis machine had access to the Internet but did not have access to other analysis machines.

2.2 Feature extraction

Researchers introduced malware sharing methods such as Malware Attribute Enumeration and Characterisation (MAEC) [23] and Open Indicators of Compromise (OpenIOC) [24] to identify malware infection based on its network and host level indicators. Consequently, these standardised malware reporting formats characterise malware samples uniformly and save malware community from the redundant analysis. In this study, common

Table 1 Features and their types

Feature category	Type	Value
episodes (sequence API calls)	n-gram	'FbFcDaFc', 'AbAaFaDaFc', 'DcFc',…
mutex names	string	'z3sd'
created processes	string	'reg.exe'
copy itself	Boolean	False
delete itself	Boolean	False
DNS requests	string	'fewfwe.com www.hugedomains.com fewfwe.net'
remote IPs	string	'54.209.61.132 216.38.220.26'
TCP Dst port	string	'80'
UDP Dst port	string	none
presence of the special APIs	Boolean	'isdebuggerpresent': false, 'setwindowshook': true
read files	string	none
registry keys	string	none
changed/created files	string	'%Document and Settings%/ftpdll.dll %SYSTEM%/drivers/spools.exe %SYSTEM%/ftpdll.dll %APP DATA%/cftmon.exe ...'
changed/created registry keys	string	'HKCU\Software\ Microsoft\Windows\ CurrentVersion\Run HKLM\SOFTWARE\Microsoft\WindowsNT\ CurrentVersion\ Winlogon ...'
downloaded EXE	string	'spools.exe cftmon.exe'
downloaded DLLs	string	'ftpdll.dll'
user-agents	string	'–'
IDS signature	string	none
HTTP requests	string	'/?&v = Chatty /domain profile.cfm?d = fewfwe&e = com'
ICMP data	string	none
ICMP host	string	none
IRC commands	string	none

Table 2 Encoding API calls with two characters

Category	Code	API #	APIs
Hooking	Aa	1	unhookwindowshookex
Network	Ba … Bj	10	dnsquery, getaddrinfo, httpopenrequest, httpsendrequest, internetclosehandle, internetconnect, internetopen, internetopenurl, internetreadfile, internetwritefile
Windows	Ca, Cb	2	findwindow, findwindowex,
Process	Da … Du	21	createprocessinternal, exitprocess, ntallocatevirtualmemory, ntcreateprocess, ntcreateprocessex, ntcreatesection, ntcreateuserprocess, ntfreevirtualmemory, ntopenprocess, ntopensection, ntprotectvirtualmemory, ntreadvirtualmemory, ntterminateprocess, ntwritevirtualmemory, readprocessmemory, shellexecuteexw, system, virtualfreeex, virtualprotectex, writeprocessmemory, zwmapviewofsection
Misc	Ea, Eb	2	getcursorpos, getsystemmetrics
System	Fa…Fj	10	exitwindowsex, isdebuggerpresent, ldrgetdllhandle, ldrgetprocedureaddress, ldrloaddll, lookupprivilegevalue, ntclose, ntdelayexecution, setwindowshookex, writeconsole
Threading	Ga…Gk	11	createremotethread, createthread, exitthread, ntgetcontextthread, ntcreatethread, ntopenthread, ntresumethread, ntsetcontextthread, ntsuspendthread, ntterminatethread, rtlcreateuserthread
synchronisation	Ha…Hc	3	ntcreatemutant, ntcreatenamedpipefile, ntopenmutant
device	Ia	1	deviceiocontrol
Registry	Ja…Jy	25	ntcreatekey, ntdeletekey, ntdeletevaluekey, ntenumeratekey, ntenumeratevaluekey, ntloadkey, ntopenkey, ntquerykey, ntquerymultiplevaluekey, ntqueryvaluekey, ntrenamekey, ntreplacekey, ntsavekey, ntsetvaluekey, regclosekey, regcreatekeyex, regdeletekey, regdeletevalue, regenumkeyex, regenumkey, regenumvalue, regopenkeyex, regqueryinfokey, regqueryvalueex, regsetvalueex
Filesystem	Ka…Kq	17	createdirectory, removedirectory, findfirstfile, deletefile, ntcreatefile, ntopenfile, ntreadfile, ntwritefile, ntdeviceiocontrolfile, ntquerydirectoryfile, ntqueryinformationfile, ntsetinformationfile, ntopendirectoryobject, ntcreatedirectoryobject, movefilewithprogress, copyfile, ntdeletefile
Services	La…Lf	6	controlservice, createservice, deleteservice, openscmanager, openservice, startservice,
Socket	Ma…Mv	22	accept, bind, closesocket, connect, gethostbyname, ioctlsocket, listen, recv, recvfrom, select, send, sendto, setsockopt, shutdown, socket, transmitfile, wsarecv, wsarecvfrom, wsasend, wsasendto, wsasocket, wsastartup

malicious features, which are also the most significant and representative features, are used to present malware behaviour both in MAEC and OpenIOC format.

Instead of directly taking into account API call based modelling, we choose category based modelling due to its lower feature space. Malware's feature category, type and value information is listed in Table 1. Furthermore, when we use categories of the API call, processing of the machine learning algorithms costs less time and CPU power. API calls and categories are given in Table 2.

To assign a weight to each feature extracted as a result of n-gram search over API call sequences, the metric of frequency and inverse document frequency is used. The statistical measure about how often a term occurs in a document is the frequency metric,

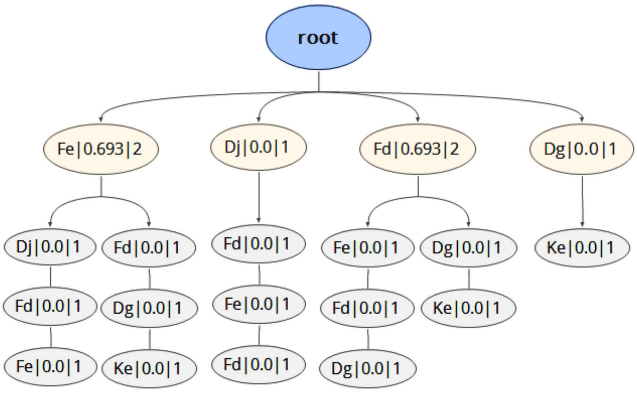

Fig. 1 *Trie with depth 5 built with S_1*

denoted by **tf**. Inverse document frequency (**idf**) measures whether the given term is common or rare in all documents (in our case, a document refers to a malware sample). **idf** leverages the feature that rarely occurs in the document corpus. The weight of each feature is the product of these two metrics.

2.2.1 API call sequence mining: We extract meaningful API call sequences from a continuous sequence of commands by adapting the Voting Experts algorithm proposed by Cohen *et al*. [25]. These API call sequences are called episodes and episodes are discovered based on the score assigned to every element of the sequence. The higher value of the score implies more likelihood for the element to be the break point of an episode. To discover the episodes, frequency and entropy experts score for each position of a sliding window with constant size, and the scores for each element are aggregated. The frequency expert determines whether a subsequence is a possible episode based on its frequency of occurrence in the entire sequence. The higher the frequency of the subsequence, the lower its probability of containing a break point. By following the same motivation, the entropy expert determines break points by calculating the entropy value of the subsequence. The main reasoning behind the entropy analogy is that if an element precedes different elements then it has a high probability of being a break point. Indeed, an element preceding various distinct elements, it has a high value of entropy. In contrast, the entropy is lower for an element preceding redundant patterns occurring in a subsequence.

Definition: Entropy of a node is defined as the entropy of the sequence from the root (starting node) to this given node and it is formulated as

$$e(x) = -\sum_{i=1}^{\ell} p(x_i)\log p(x_i), \tag{1}$$

where x is the node and $e(x)$ denotes its entropy. The probability of the subsequence existence at the child node, denoted by x_ℓ for $i = 1, \ldots, \ell$ is given by

$$p(x_\ell) = \frac{f(x_\ell)}{f(x)}, \tag{2}$$

where $f(x)$ is the frequency of the node x and $f(x_\ell)$ is the frequency of its child node.

Example: Let us consider the following four sets of API call sub-sequences belonging to a malware sample named Zbot, Chindo, Dorkbot and Ramnit, respectively. These sub-sequences are obtained via dynamic analysis:
S_1 = (Fe Dj Fd Fe Fd Dg) # Zbot
S_2 = (Fc Fe Fd Kf Ia Fe Fd) # Chindo
S_3 = (Dg Fc Fd Fc Fd Fc Fd Fc Fd) # Dorkbot
S_4 = (Km Fe Fd Fc Fd) # Ramnit

All sequences are combined in a row to generate a single trie. A single trie structure allows cheap calculation of the frequency and entropy of API calls captured during dynamic analysis. The trie is a pre-fix tree of depth d, and along this trie, each distinct subsequence of length $d-1$ is represented by a path from 'root' node to a leaf node (i.e. terminal node). Every node in the trie has a frequency value of the sequence computed by the path from 'root' to the subject (current) node and entropy value that will be used to find the possible location of break points by scoring each location of the sequence.

For each level of the trie, mean frequency denoted by $\bar{f}_l$, mean entropy $\bar{e}_l$, the standard deviation of mean level frequency $\sigma_{\bar{f}_l}$ and standard deviation of the mean level of entropy $\sigma_{\bar{e}_l}$ are computed. Entropy and frequency of each node are normalised with respect to its mean and standard deviation:

$$f(x) = \frac{f(x) - \bar{f}_l}{\sigma_{\bar{f}_l}}, \quad e(x) = \frac{e(x) - \bar{e}_l}{\sigma_{\bar{e}_l}},$$

where $e(x)$ and $f(x)$ are the entropy and frequency of the node, respectively.

2.2.2 Episode discovery using the n-gram trie structure: The first API call sequence example, which is denoted by S_1, is used to generate a trie with depth 4 in Fig. 1. The nodes have entropy and frequency values, respectively. In this case, all sub-sequences occur once in Fig. 1. Along this trie, (Fe) and (Fd) appears twice in S_1 and (FeDjFdFe) appears once. After embedding S_2 on the trie, the representative trie is generated for S_1 and S_2 (see for instance Fig. 2). However, the nodes representing the sequence FcFdFcFd in Fig. 3 occur three times (coloured with red) in the entire sequence and its entropy value which is calculated by (1) is 0. Please note that if a node does not have a sub-node (child node) its entropy value is equal to 0. The pseudo-code algorithm discovering episodes by applying the trie structure is given in Algorithm 1 (see Fig. 4).

The proposed approach stores the information of all API call sequences of malware samples in a single trie. To find episodes in a sequence, we use a sliding window of size $l = d - 1$. Let us consider, a sequence is given by $S_t = x_1, x_2, \ldots, x_l, x_{l+1}, \ldots, x_n$ and for each possible break position in a window, the entropy at location i (between x_i and x_{i+1}) is the entropy of node x_i at level i of the trie representing a sequence of $x_1, x_2, \ldots, x_i$. For example, when we use sliding window $l = 4$, four possible break positions such as (Fc|Fe|Fd|Kf|) exist for S_1. The entropy of the first location is derived by the node labelled as Fc at the first level of the trie. Similarly, entropy at the second location is the entropy of the node labelled as Fe at the second level of the trie with Fc as its parent node. The position with the highest entropy in the window is determined as a candidate break point and its score is increased by 1. The frequency at the location i in a window is calculated by the sum of the sub-sequences $(x_1, x_2, \ldots, x_i)$ and $(x_{i+1}, x_{i+2}, \ldots, x_l)$. For example, the frequency of the first location in the window (i.e. FeDjFdFe) is equal to the sum of f(Fe) + f(DjFdFe), where the frequency of the subsequence Fe and DjFdFe is denoted by f(Fe) and f(DjFdFe), respectively. The highest frequency position in a window is incremented by 1.

After sliding the fixed size window along the sequence, a score is obtained for each position. After a position is recurrently scored by frequency and entropy experts, the position with a local maximum in the score array is chosen as the break point of the episode. Discovery of episodes by using two voting experts (i.e. frequency and entropy) is illustrated in Fig. 5 for the second set of the API call subsequence belonging to a malware Chindo as an example. By applying the episode discovery algorithm to these sequences, the episodes are discovered as follows:
E_1: FeDjFd, FeFdDg
E_2: FcFeFd, KfIaFeFd
E_3: DgFcFd, FcFd, FcFd, FcFd, FcFd
E_4: KmFeFdFcFd

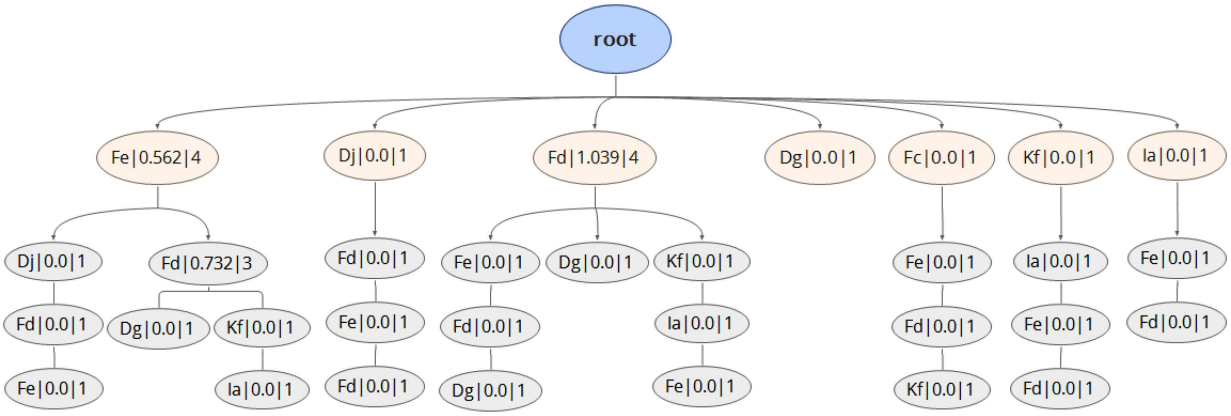

Fig. 2 *Trie with depth 5 built with S_1 and S_2*

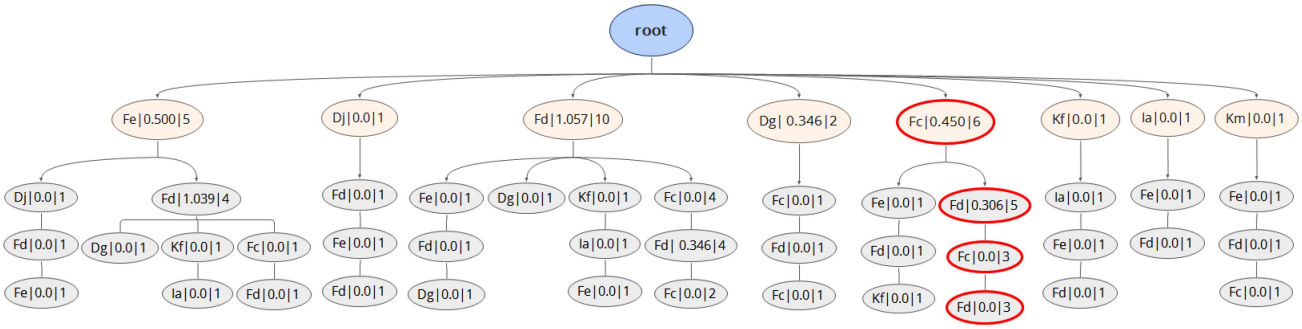

Fig. 3 *Final trie with depth 5 after four sub-sequences denoted by S_1, S_2, S_3 and S_4 are committed*

Remark: Although the discovered episodes involve meaningful sequences, if a fixed length n-gram is applied, for instance $n = 4$, critical information is lost by ignoring 5 g.

Table 3 shows the relationship between malicious API call patterns and malicious activity. The Voting Experts algorithm extracts malicious API call patterns that are written by malware authors. The impact caused by a malware can be identified via malicious API call patterns and accurate classification to its ground truth malware family can be assured.

The proposed Voting Experts based API mining approach contributes to the computationally efficient identification of n-grams. The number of unique API call sequences discovered in the dataset constituted by malware and benign samples is compared in Fig. 6. The number of unique API call sequences generated by the n-gram approach is increased exponentially when the length of the n-gram is incremented (see, for instance, the left side of Fig. 6). In consequence, the computational complexity and processing time of the classification algorithm is increased. The proposed Voting Experts based API mining approach generates significantly fewer features than the n-gram method, the number of generated unique n-grams versus the given trie depth is plotted in the right side of Fig. 6. The dataset used for comparison purposes is elaborated in Section 3.1.

2.3 Labelling malware

For labelling malware samples, Virustotal [26], an online web-based multi anti-virus scanner, is used. Malware labelling may differ between different anti-virus engines [27, 28]. To be more consistent in labelling, the most common scan result is determined as the tag of a sample. As the labelling process completely affects the classification accuracy, researchers need to precisely determine anti-virus labelling and cross check the labelling results. Table 4 shows the malware classes and their counts used to evaluate the proposed method.

2.4 Building classification model

2.4.1 Online learning: An online learning algorithm works in a sequence of discrete-time instant, denoted by $t \geq 0$. The algorithm considers an instance $\vec{x}_t \in R^d$, d-dimensional vector as an input, to make the prediction $\hat{y}_t \in \{+1, -1\}$ (for binary classification) regarding its current prediction model. After predicting, it receives the true label $y_t \in \{+1, -1\}$ and updates its model (a.k.a. hypothesis) based on prediction loss $\ell(y_t, \hat{y}_t)$ meaning the incompatibility between prediction and actual class. The goal of online learning is to minimise the total number of incorrect predictions; $\text{sum}(t: y_t \neq \hat{y}_t)$. The pseudo-code for generic online learning is given in Algorithm 2 (see Fig. 7).

A multi-class classification problem is considered:

- $\vec{x}_t$ represents the feature vector of a malware instance at the tth iteration. When implementing the algorithm for detection of a malware sample, its set of features is constituted by using the basis of the independent feature vector set given in Table 1. For each independent feature, its category, type and value are known so the sample can be represented in terms of these independent features.
- $\vec{y}_t$ is the set of labels at the tth iteration. $\hat{y}_t$ is the output of the algorithm or more simply, prediction of a malware family for the given $\vec{x}_t$. The families to be matched are listed in the first column of Table 4, and their union constitutes the whole family set.
- ℓ_t is the function using the relation between the set of features for the given sample, the computed weight, denoted by $\vec{w}_t$ and the estimated malware label.
- $\vec{w}_{t+1}$ denotes the updated weight vector at the $(t+1)$th iteration towards the final class prediction.

Our focus, in particular, is not on theoretical contribution to online learning algorithms as this has been addressed in many studies over the past few years. Instead, the attention is drawn to the computationally efficient identification of n-grams and improving the accuracy of classifying malware. Online machine learning algorithms differ according to how to initialise the weight vector $\vec{w}_{t=1}$ and update function used to alter the weight vector at the end of each iteration. The confidence weighted (CW) [29] and

Input : Trie of depth d: (T_d) and a set of sequence S
Output: Set of episodes(E)
foreach node(x) $\in T_d$ **do**
  calculate entropy $e(x)$
end
foreach level $(l) \in T_d$ **do**
  calculate mean frequency ($\bar{f}_l$) and mean entropy ($\bar{e}_l$)
  calculate standard deviations ($\sigma_{\bar{f}_l}$ and $\sigma_{\bar{e}_l}$)
  foreach node(x) $\in l$ level *of* T_d **do**
    update frequency and entropy by mean and standard deviation values
    $f(x) = \frac{f(x)-\bar{f}_l}{\sigma_{\bar{f}_l}}, \quad e(x) = \frac{e(x)-\bar{e}_l}{\sigma_{\bar{e}_l}}$
  end
end
episodes(E) = list()
foreach $S_t(s_1, s_2, s_3, ..., s_{length(S_t)}) \in S$ **do**
  scores = [0 for i in length(S_t)]
  for $i = 1$ *to* (length($S_t - d$)) **do**
    take a windows of length $k = d - 1$ at position i in S
    for *each possible boundary in the window* **do**
      find $\text{Max}_j f(s_i, ..., s_{i+j}) + f(s_{i+j+1}, ..., s_{i+k-1})$, $0 \leq j < k$
      score$[i + j]$ = score $[i + j] + 1$
      find $\text{Max}_j e(s_i, ..., s_{i+j})$, $0 \leq j < k$
      score$[i + j]$ = score $[i + j] + 1$
    end
  end
  end, start = 0, 0
  for $i = 1$ *to (length(scores)-2)* **do**
    if scores$[i] >$ scores$[i - 1]$ *and* scores$[i] >$ scores$[i + 1]$ **then**
      end = i
      episodes.append(s_{start}, s_{end})
      start = end
    end
    episodes.append(s_{start}, s_{end})
  end
end

Fig. 4 *Algorithm 1: Episode extraction algorithm for a set of sequences using a trie structure*

Frequency	Fc	Fe	Fd	Kf	Ia	Fe	Fd
Entropy	Fc	Fe	Fd	Kf	Ia	Fe	Fd
Frequency	Fc	Fe	Fd	Kf	Ia	Fe	Fd
Entropy	Fc	Fe	Fd	Kf	Ia	Fe	Fd
Frequency	Fc	Fe	Fd	Kf	Ia	Fe	Fd
Entropy	Fc	Fe	Fd	Kf	Ia	Fe	Fd
Frequency	Fc	Fe	Fd	Kf	Ia	Fe	Fd
Entropy	Fc	Fe	Fd	Kf	Ia	Fd	Fd
Score	2	0	4	0	0	1	1
Episodes	Fc	Fe	Fd	Kf	Ia	Fe	Fd

Fig. 5 *Voting expert applied to S_2 sequence*

adaptive regularisation of weight (AROW) [30] online classifiers are evaluated in our distributed computing environment in order to empirically reach the highest level of accuracy.

2.4.2 Offline(batch) learning: The model is updated after processing the whole dataset. We compare online learning algorithm results with respect to the baseline classifiers such as K-NN Random Forest (RF) and SVM [31–33]. However, offline algorithms cannot update their model in response to the introduction of a new malware variant to the classification scheme.

3 Experiments and results

In this section, we present experimental results obtained using different machine learning algorithms. We first describe the dataset. Then, we define the evaluation metrics and compare the accuracy of the classifiers based on the n-gram approach and the proposed approach. Finally, we examine the responsiveness of the proposed method when the run-time model is changed by introducing a new malware sample that is new to the online learning system.

3.1 Dataset

The testing malware dataset was obtained from 'VirusShare Malware Sharing Platform' [34]. A large amount of malware with different types including PE, HTML, Flash, Java, PDF, APK was downloaded. Since VirMon can analyse only executable files, these files are considered. All experiments were conducted under the Windows 7 SP1 operating system with Intel(R) Core™ i5-2410M@2.30 GHz processor and 2 GB of RAM.

Throughout the analysis, 25% of samples belonging to malware dataset did not run because either some particular samples checked hardware specification, file format errors or detect virtual machine environment and refused to run. 45% of samples did not perform enough activities because the majority of these samples required user interaction to be installed. Besides that, some samples needed more CPU and memory resource to run. Since these samples could cause false positives, we removed them from the dataset. 5% of samples reached timeout of 3 min and halted automatically.

The analysis with 50 guest machines took 15 days to analyse 60,000 samples. At the end of the analysis, we prepared a set of testing dataset constituted by 17,400 responding samples belonging to 60 distinct families. Also, 532 benign executables are obtained from the system32 folder of Microsoft windows. These classes are not balanced in terms of the same sample size but since 60 families of malicious executables and one benign class constitute the dataset, balance problem among class size is reduced. For instance, sample count of each family is given in Table 4. (The interested readers may also investigate results obtained by using unbalanced dataset in [35].) The dataset used in this study is available for academic and research communities [36].

Table 3 Malicious activities and their API call patterns

Malicious activity	Malicious API call
process hollowing	CreateProcess, GetModuleHandle, GetProcAddress, VirtualAllocEx
	WriteProcessMemory, SetThreadContext, ResumeThread
create remote thread	OpenProcess, GetModuleHandle, GetProcAddress, VirtualAllocEx
	WriteProcessMemory, CreateRemoteThread
enumerating all processes	(CreateToolhelp32Snapshot, Process32First, Process32Next) or (WTSEnumerateProcesses)
drop file from PE resource section	GetModuleHandle, FindResource, LoadResource, CreateFileA
IAT hooking	GetModuleHandle, strcmp, VirtualProtect
delete itself	GetModuleFileName, ExitProcess, DeleteFile
download and execute PE file	URLDownloadToFile, ShellExecute
bind TCP port	WSAStartup, socket
capture network traffic	socket, bind, WSAIoctl, recvfrom

3.2 Classification metrics

To evaluate the proposed method, the following class-specific metrics are used: *precision*, *recall* (a.k.a. sensitivity), *F-measure*, and *overall accuracy* (the overall correctness of the model). Recall (a.k.a positive predictive value) is the probability for a sample in class c to be classified correctly, the maximum value 1 means that the classifier is always correct about its prediction whether an instance belongs to class c. Precision gives the probability for an estimated instance about classification in class c to be actually in class c. Low precision means that a large number of samples are incorrectly classified in class c.

To better evaluate classification performance we employ F-measure metric, which is a common comparison metric to compute the harmonic mean of precision and recall. F-measure takes value in the interval between 0 and 1. A higher value of F-measure implies better overall performance and a value of 1 means perfect classification. The metrics are given as follows:

$$\text{precision} = \frac{\text{tp}}{\text{tp} + \text{fp}} \tag{3}$$

$$\text{recall} = \frac{\text{tp}}{\text{tp} + \text{fn}} \tag{4}$$

$$F\text{-measure} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{5}$$

$$\text{accuracy} = \frac{\text{correctly classified instances}}{\text{total number of instances}} \tag{6}$$

For instance, consider a given class c. True positives (tp) refer to the number of samples in class c that are correctly classified while true negatives (tn) are the number of samples not in class c that are correctly classified. False positives (fp) refer the number of samples not in class c that are incorrectly classified. Similarly, false negatives (fn) denote the number of samples in class c that are incorrectly classified. The term positive and negative indicates the classifier's success whereas the term true and false denotes whether that particular prediction is matched with ground truth label or not.

To prevent over fitting, a 10-fold cross-validation approach is used. The set of malware samples is randomly divided into ten equal size subsets. Then, a single subset is used for testing the model, and the remaining nine subsets are used for building the classification model. This process is then repeated ten times such that each subset is used at least once as the testing data. Finally, the

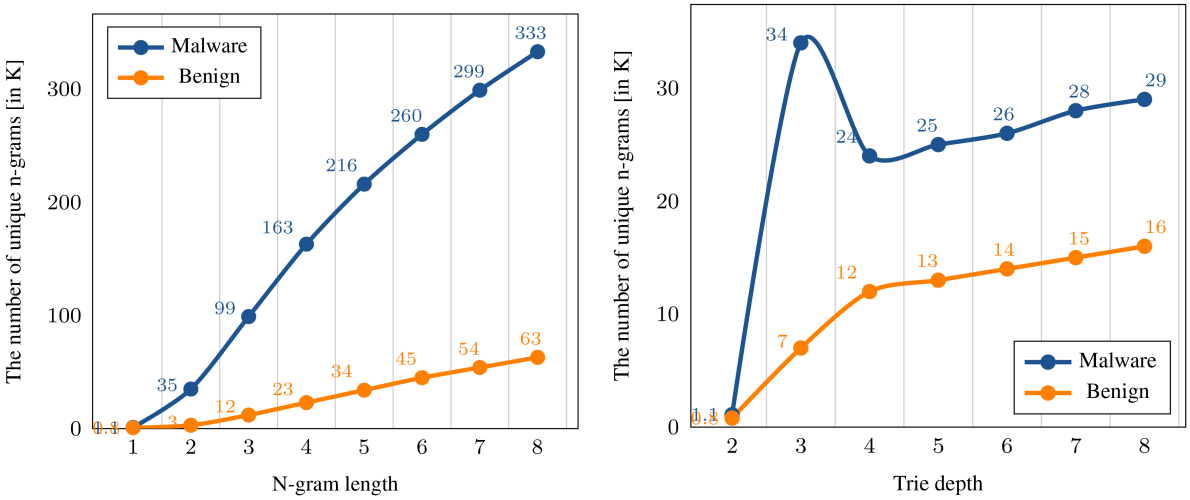

Fig. 6 *Comparison of the number of unique n-grams identified by applying n-gram and the proposed approach*

Table 4 Malware families and class-specific performance measure

Family	ID	Count	Precision	Recall	*F*-measure
Neshta	1	12	1.000	1.000	1.000
Ridnu	2	12	1.000	1.000	1.000
Zegost	3	12	0.500	1.000	0.667
Hotbar	4	13	0.667	1.000	0.800
Rebhip	5	13	1.000	1.000	1.000
Rungbu	6	13	1.000	1.000	1.000
Startpage	7	14	1.000	1.000	1.000
Azero	8	16	1.000	1.000	1.000
Wintrim	9	16	1.000	1.000	1.000
Beebone	10	17	1.000	1.000	1.000
Downloader	11	17	1.000	1.000	1.000
Lamechi	12	17	1.000	1.000	1.000
Simda	13	17	1.000	1.000	1.000
Tzeebot	14	17	1.000	1.000	1.000
Malagent	15	18	0.667	1.000	0.800
Phorpiex	16	18	1.000	0.667	0.800
Renos	17	18	1.000	1.000	1.000
ICLoader	18	19	1.000	1.000	1.000
Chir	19	20	1.000	1.000	1.000
Swrort	20	20	1.000	1.000	1.000
Brontok	21	21	1.000	1.000	1.000
Winwebsec	22	23	1.000	0.667	0.800
Fareit	23	25	1.000	1.000	1.000
Gamarue	24	25	0.500	1.000	0.667
Emotet	25	26	1.000	0.600	0.750
Injector	26	26	0.667	1.000	0.800
Chindo	27	27	1.000	1.000	1.000
Sality	28	27	1.000	0.600	0.750
Hicrazyk	29	28	1.000	1.000	1.000
Swizzor	30	28	0.875	1.000	0.933
Upatre	31	29	1.000	0.750	0.857
VBInject	32	29	1.000	1.000	1.000
CeeInject	33	30	0.500	1.000	0.667
Allaple	34	33	0.500	1.000	0.667
Beaugrit	35	34	1.000	1.000	1.000
Expiro	36	35	1.000	1.000	1.000
Fynloski	37	38	1.000	1.000	1.000
Jadtre	38	45	1.000	0.833	0.909
Kuluoz	39	46	1.000	1.000	1.000
Luder	40	48	0.750	0.750	0.750
CostMin	41	54	1.000	1.000	1.000
Kelihos	42	54	0.800	0.800	0.800
Urausy	43	55	1.000	1.000	1.000
Usteal	44	55	0.800	1.000	0.889
Danglo	45	57	1.000	1.000	1.000
Obfuscator	46	59	0.500	1.000	0.667
OxyPumper	47	61	1.000	1.000	1.000
Parite	48	63	1.000	1.000	1.000
Dynamer	49	65	0.625	1.000	0.769
Vobfus	50	67	1.000	1.000	1.000
Zbot	51	71	1.000	1.000	1.000
Virut	52	79	0.500	0.750	0.600
C2Lop	53	113	1.000	1.000	1.000
Bladabindi	54	141	1.000	1.000	1.000
SquareNet	55	190	1.000	1.000	1.000
VB	56	309	1.000	1.000	1.000
BetterSurf	57	321	1.000	1.000	1.000
CouponRuc	58	799	1.000	0.988	0.994
Ogimant	59	2373	0.992	1.000	0.996
Tugspay	60	5070	1.000	0.982	0.991
Benign	61	532	1.000	0.992	0.996

classification accuracy is calculated by averaging the accuracy obtained at the end of ten iterations.

3.3 Classification results

Two different feature sets are extracted by the n-gram approach and the proposed method, respectively. Then, the malware detection rate, *i.e.* accuracy, and *F*-measure is compared in Fig. 8. For evaluation purposes, statistical measures for accuracy are used and online machine learning algorithm results are compared with the baseline *K*-NN (where $K = 5$), RF and SVM classifier results.

For the first feature set extracted by the n-gram approach, when $n = 4$, the detection rate 98% and *F*-measure 97.5% is reached by

Input : $\vec{w}_{t=1} = (0, ..., 0)$
foreach *round t in (1,2,..,N)* **do**
  Receive instance $\vec{x}_t \in \mathbb{R}^d$
  Predict label of $\vec{x}_t$: $\hat{y}_t = \text{sign}\,(\vec{x}_t.\vec{w}_t)$
  Obtain true label of the $\vec{x}_t$: $y_t \in \{+1, -1\}$
  Calculate the loss: ℓ_t
  Update the weights: $\vec{w}_{t+1}$
end
Output: $\vec{w}_{t=N} = (w_1, ..., w_d)$

Fig. 7 *Algorithm 2: Generic online learning algorithm*

RF and CW algorithms, respectively (see for instance two figures denoted by (a) and (b) on the left side of Fig. 8). For the second feature set extracted by the proposed method, when trie depth is given 4, the CW algorithm identifies malware samples with 99.9% detection rate and 99% *F*-measure and benign samples with 99.6% accuracy and 99.8% *F*-measure (see two figures denoted by (c) and (d) on the right side of Fig. 8). When the length of the n-gram and the depth of trie is increased, the detection rate of the algorithms is decreased. Since on the one hand, when the size of the n-gram is increased the feature space becomes very large and may contain irrelevant API sequences. On the other hand, when the trie depth is increased more than 5, a longer API call sequence does not improve further the identification performance. Please note that as illustrated in Fig. 1–3, in a single trie, the root is indexed by 1, therefore trie depth indexing 2 is equivalent to n-gram 1.

The classification metrics across all families are plotted subject to different n-gram and trie depth features in Figs. 9 and Fig. 10, respectively. The proposed method improves the accuracy of classification. The baseline and online machine learning algorithms' classification results in Fig. 9 are obtained with the first feature set extracted by the n-gram approach, and they are lower than the proposed Voting Experts based API mining algorithm results plotted in Fig. 10. When the trie depth is given 4, the CW algorithm gives higher classification results; 95.5% precision, 96% recall, 94.5% *F*-measure and 95% accuracy.

For an online machine learning algorithm, the regularisation weight parameter can be tuned to enhance classification results. This parameter, denoted by *C*, determines the size of weight change at each iteration. A larger value means a possibility of a

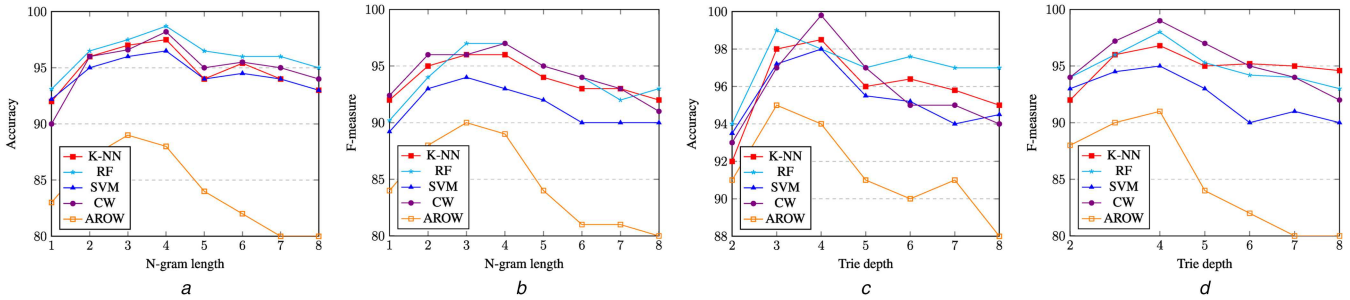

Fig. 8 *Comparison of malware detection rate and F-measure of the selected algorithms versus different n-gram size and the proposed API mining approach*
(a) Accuracy of the n-gram method, ***(b)*** *F*-measure of the n-gram method, ***(c)*** Accuracy of the proposed method, ***(d)*** *F*-measure of the proposed method

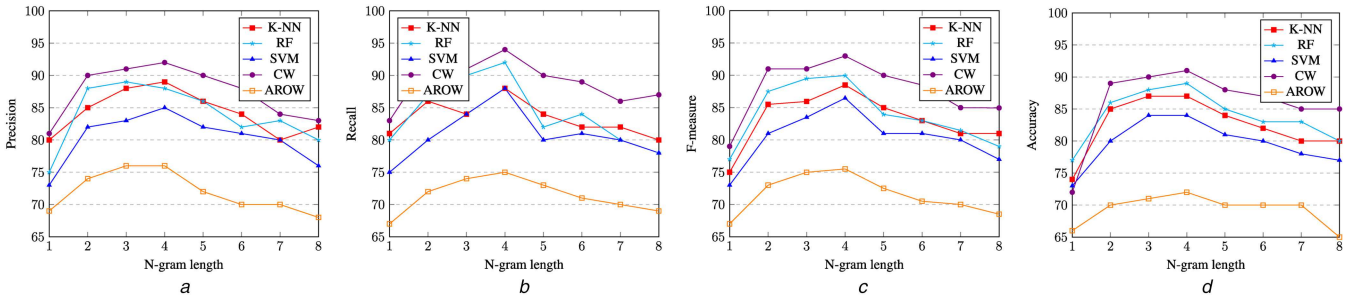

Fig. 9 *Performance of the selected algorithms based on different n-gram length*
(a) Precision, ***(b)*** Recall, ***(c)*** *F*-measure, ***(d)*** Accuracy

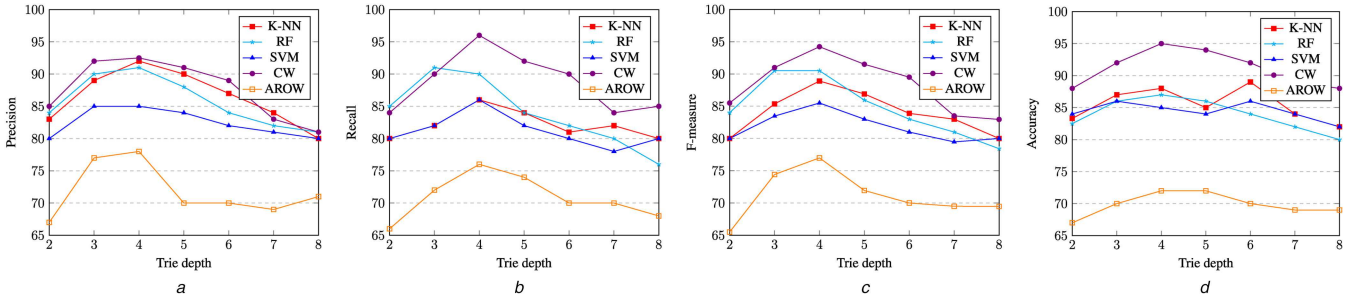

Fig. 10 *Performance of the selected algorithms based on different trie depths used in the proposed sequence of the API mining method*
(a) Precision, ***(b)*** Recall, ***(c)*** *F*-measure, ***(d)*** Accuracy

Table 5 Classification accuracy of online learning algorithms versus different regularisation weight parameters

	Regularisation weight (C)															
	$C = 1.0$		$C = 2.0$		$C = 3.0$		$C = 4.0$		$C = 5.0$		$C = 10.0$		$C = 100.0$		$C = 1000.0$	
	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
CW	0.960	0.943	0.953	0.933	0.963	0.943	0.960	0.953	0.967	0.953	0.980	0.970	0.980	0.980	0.980	0.967
AROW	0.720	0.717	0.720	0.713	0.703	0.707	0.697	0.713	0.707	0.693	0.703	0.720	0.707	0.703	0.703	0.700

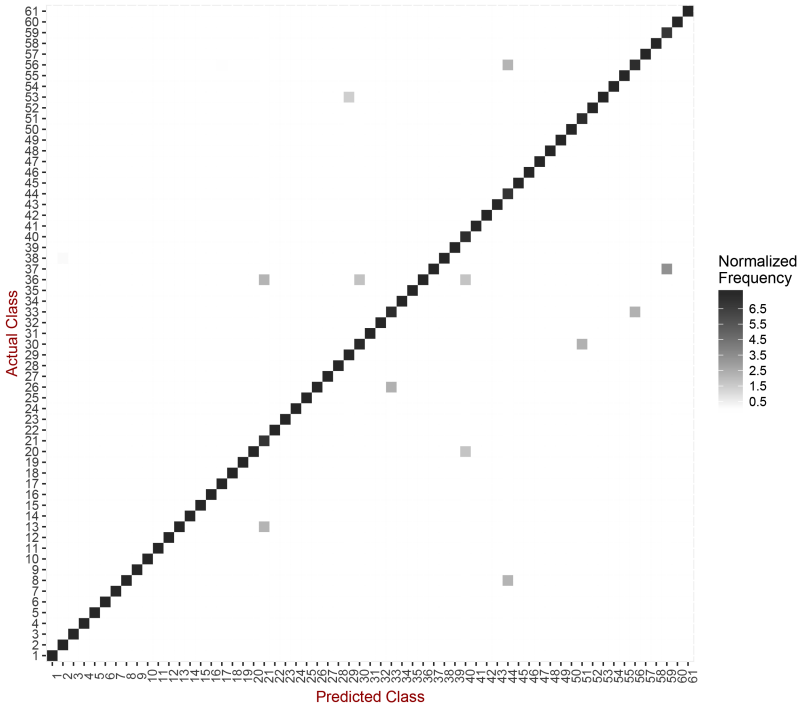

Fig. 11 *Normalised confusion matrix*

higher change in the updated weight vector and the model is built faster. However, as a consequence, the model becomes more dependent on the training set and more susceptible to noise data. The interested reader may read [29, 30] to investigate the vector norm used for each individual choice of the loss function and weight update rule in the algorithms.

The features listed in Table 1 including the sequence of API for the variable sized n-gram type is used. In Table 5, the accuracy of training and testing for online learning algorithms is computed subject to different values of regularisation weight parameters. Training and testing accuracy can vary significantly due to model building capability while using the same set of features and samples. AROW uses the adaptive update method while handling a new sample at each learning step, and this algorithm alters weights with respect to instantaneous changes such as mislabelled training sample at the learning stage. However, still, its accuracy level is limited by 72.0%. CW learning is mainly a binary classification problem and can use a sign function for updating weights and reaching the maximum accuracy that is 98.0%

For each family, CW learning class-specific metrics are calculated and the number of samples belonging to each specific family in the malware dataset is given in Table 4. The confusion matrix illustrates the success of the classifier at recognising instances of different classes Fig. 11). The confusion matrix displays the number of correct and incorrect predictions made by the classifier with respect to the ground truth (correct classes). The confusion matrix has $n \times n$ entries, where n is the total number of independent classes. The rows of the table correspond to actual classes and columns correspond to predicted classes. The diagonal elements of the matrix represent the number of correctly classified instances for each class, while the off-diagonal elements represent the number of incorrectly classified elements. When the diagonal values of the confusion matrix are higher in comparison with other row values (in the case of perfect matching, the diagonal value is one and the remaining row values are zero), the model is well matched with the dataset and high (the highest) accuracy is reached at predicting that particular family. The confusion matrix also shows that the CW algorithm correctly classifies almost all samples into their respective families. The number of wrongly estimated samples for each class, which is marked in the off-diagonal zone in Fig. 10 is only one.

We also evaluate the performance of the proposed method on new malware samples, i.e. when the model is updated for covering a new malware sample that has not been trained and tested by the online learning system yet. To accomplish this new malware evaluation task, we gather ten new malware samples belonging to the Stimilini family and eight new samples belonging to the NetWiredRC family from VirusShare. After obtaining behavioural profiles from dynamic analysis, only one sample from these two new families is randomly selected, and used to train and update the model. Subsequently, the remaining samples, i.e. 18–2 = 16 samples that are new to the model, are evaluated by the updated model during 1.3 s. The results show that all samples are classified correctly. It should be noted that the proposed method does not need to start from scratch to build a training model.

4 Conclusions

Currently, malware samples are the variants of existing ones, and since many samples are armed with obfuscation techniques, common security solutions can be easily evaded. This study addresses the challenge of classifying and identifying malware samples by using runtime artefacts while being robust to obfuscation. Computationally efficient identification of n-grams is presented and the accuracy of binary classifiers using rich dynamic features with implicit behaviour captured by n-grams is improved.

The proposed method exploits run-time behaviours of an executable to build the feature vector. We evaluate five machine learning algorithms with 17,400 malware samples belonging to 60 families and 532 benign samples. CW algorithm's training and

testing accuracy are reached at 98%. Our proof-of-concept implementation and results show that runtime behaviour extraction enhances malware classification. Moreover, the proposed method provides valuable insights towards providing practical solutions to anti-virus companies or malware research institutes deploying and working on large-scale malware classification schemes.

5 References

[1] Sharma, A., Sahay, S.K.: 'Evolution and detection of polymorphic and metamorphic malwares: a survey', *Int. J. Comput. Appl.*, 2014, **9**, (2), pp. 7–11

[2] Symantec.: 'Internet Security Threat Report' 2016

[3] Pektas, A., Eris, M., Acarman, T.: 'Proposal of n-gram based algorithm for malware classification'. The Fifth Int. Conf. on Emerging Security Information, Systems and Technologies, 2011, pp. 1–6

[4] Zhong, Y., Yamaki, H., Takakura, H.: 'A malware classification method based on similarity of function structure'. IEEE/IPSJ 12th Int. Symp. on Applications and the Internet, 2012, pp. 256–261

[5] Forrest, S., Longstaff, T.A.: 'A sense of self for Unix processes'. IEEE Symp. on Security and Privacy, 1996, pp. 120–128

[6] Liu, W., Ren, P.: 'Behavior-based malware analysis and detection'. First Int. Workshop on Complexity and Data Mining (IWCDM), 2011, pp. 39–42

[7] Natani, P., Vidyarthi, D.: 'Malware detection using API function frequency with ensemble based classifier', in Thampi, S.M., Atrey, P.K., Fan, CI., Perez, G.M. (Eds): '*Security in computing and communications*' (Springer, Berlin, Heidelberg, 2013), pp. 378–388

[8] Xu, J.Y., Sung, A., Chavez, P.*, et al.*: 'Polymorphic malicious executable scanner by API sequence analysis'. Fourth Int. Conf. on Hybrid Intelligent Systems, 2004, pp. 378–383

[9] Shen, F., Del Vecchio, J., Mohaisen, A.*, et al.*: 'Android malware detection using complex-flows'. IEEE 37th Int. Conf. on Distributed Computing Systems, Atlanta, GA, 2017, pp. 2430–2437

[10] Mariconti, E., Onwuzurike, L., Andriotis, P.*, et al.*: 'Mamadroid: detecting android malware by building markov chains of behavioral models', CoRR, 2016, http://arxiv.org/abs/1612.04433

[11] Bai, H., Hu, C., Jing, X.*, et al.*: 'Approach for malware identification using dynamic behaviour and outcome triggering', *IET Inf. Sec.*, 2014, **8**, (2), pp. 140–151

[12] Taeho, K., Zhendong, S.: 'Behavior-based malware analysis and detection'. IEEE 11th Int. Conf. on Data Mining, 2011, pp. 1134–1139

[13] Kim, S., Park, J., Lee, K.*, et al.*: 'A brief survey on rootkit techniques in malicious codes', *J. Internet Services Inf. Sec.*, 2012, **3**, (4), pp. 134–147

[14] Nari, S., Ghorbani, A.A.: 'Automated malware classification based on network behavior'. Int. Conf. on Computing, Networking and Communications, 2013, pp. 642–647

[15] Hall, M., Frank, E., Holmes, G.*, et al.*: 'The WEKA data mining software: an update', *ACM SIGKDD Explor. Newsl.*, 2009, **11**, (1), pp. 10–18

[16] Mohaisen, A., West, A.G., Mankin, A.: 'Chatter: classifying malware families using system event ordering'. IEEE Conf. on Communications and Network Security, 2014, pp. 283–291

[17] Rieck, K., Philipp, T., Willems, C.*, et al.*: 'Automatic analysis of malware behavior using machine learning', *J. Comput. Sec.*, 2011, **19**, (4), pp. 639–668

[18] Chandramohan, M., Tan, H.B., Kuan, B.*, et al.*: 'A scalable approach for malware detection through bounded feature space behavior modeling'. IEEE/ACM 28th Int. Conf. on Automated Software Engineering, 2013, pp. 312–322

[19] Ki, Y., Kim, E., Kim, H.K.: 'A novel approach to detect malware based on API call sequence analysis', *Int. J. Distrib. Sensor Netw.*, 2015, **19**, (4), p. 4

[20] Hunt, G., Brubacher, D.: 'DETOURS: binary interception of Win32 functions'. 3rd Usenix Windows NT Symp., 1999, pp. 14–14

[21] Tirli, H., Pektas, A., Falcone, Y.*, et al.*: 'Virmon: a virtualization-based automated, dynamic malware analysis system'. Proc. of Int. Conf. on Information Security and Cryptology, 2013, pp. 57–62

[22] 'Cuckoo Sandbox', http://www.cuckoosandbox.org/, accessed 24 July 2017

[23] Kirillov, I., Beck, D., Chase, P.*, et al.*: 'Malware attribute enumeration and characterization'. Tech. Rep, The MITRE Corporation, 2010

[24] Caltagirone, S., Pendergast, A., Betz, C.: 'The diamond model of intrusion analysis', DTIC Document, 2013

[25] Cohen, P., Adams, N., Heeringa, B.: 'Voting experts: an unsupervised algorithm for segmenting sequences', *Intell. Data Anal.*, 2006, **11**, (6), pp. 607–625

[26] 'Virustotal: An online multiple AV Scan Service', http://www.virustotal.com/, accessed 17 July 2017

[27] Mohaisen, A., Alrawi, O.: 'Av-meter: an evaluation of antivirus scans and labels', in Dietrich, S. (Ed): '*Detection of intrusions and malware, and vulnerability Assessment*' (Springer International Publishing, Cham, 2014), pp. 112–131

[28] Mohaisen, A., Alrawi, O., Matt, L.*, et al.*: 'Towards a methodical evaluation of antivirus scans and labels', in Kim, Y., Lee, H., Perrig, A. (Eds): '*Information security applications*' (Springer International Publishing, Cham, 2013), pp. 231–241

[29] Dredze, M., Crammer, K., Pereira, F.: 'Confidence-weighted linear classification'. Proc. 25th Int. Conf. on Machine learning, 2008, pp. 264–271

[30] Crammer, K., Kulesza, A., Dredze, M.: 'Adaptive regularization of weight vectors', in Bengio, Y., Schuurmans, D., Lafferty, J. D., Williams, C. K. I., Culotta, A. (Eds): '*Advances in neural information processing systems*' (Curran Associates, Inc., 2009), pp. 414–422, http://papers.nips.cc/paper/3848-adaptive-regularization-of-weightvectors.pdf

[31] Altman, N.S.: 'An introduction to kernel and nearest-neighbor nonparametric regression', *Am. Stat.*, 1992, **46**, (32), pp. 175–185

[32] Wu, T.F., Lin, C.J., Weng, R.C.: 'Probability estimates for multi-class classification by pairwise coupling', *J. Mach. Learn. Res.*, 2004, **5**, pp. 975–1005

[33] Breiman, L.: 'Random forests', *Mach. Learn.*, 2001, **45**, (1), pp. 5–32

[34] 'Virusshare: Malware Sharing Platform', http://www.virusshare.com/, accessed 17 July 2017

[35] Kolosnjaji, B., Eraisha, G., Webster, G.*, et al.*: 'Empowering convolutional networks for malware classification and analysis'. Int. Joint Conf. on Neural Networks, May 2017, pp. 3838–3845

[36] 'Analysis reports of the malware samples used in this study', http://research.pektas.in, accessed 28 August 2017