MalSPM: Metamorphic Malware Behavior Analysis and Classification using Sequential Pattern Mining

M. Saqib Nawaz[1[0000−0001−9856−2885]], Philippe Fournier-Viger[1[0000−0002−7680−9899]], M. Zohaib Nawaz[2[0000−0001−9205−912X]], Guoting Chen[3[0000−0003−2072−1588]], and Youxi Wu[4[0000−0001−5314−3468]]

[1]College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
[2]Department of Computer Science & IT, University of Sargodha, Pakistan
[3]School of Science, Harbin Institute of Technology (Shenzhen), Shenzhen, China
[4]Department of Computer Science and Engineering, Hebei University of Technology, Tianjin, China
saqib_dola@yahoo.com, philfv@szu.edu.cn, zohaib.nawaz@uos.edu.pk, chenguoting@hit.edu.cn, wuc567@163.com

Abstract. Malware poses a serious threat to the computers of individuals, enterprises and other organizations. In the Windows operating system (OS), Application Programming Interface (API) calls are an attractive and distinguishable feature for malware analysis and detection as they can properly reflect the actions of portable executable (PE) files. In this paper, we propose MalSPM, an intelligent malware detection system based on sequential pattern mining (SPM) for the analysis and classification of malware behavior during executions. A dataset that contains sequences of API calls made by different malware on Windows OS is abstracted into a suitable format. SPM algorithms are first used on the corpus to find frequent API calls and their patterns. Moreover, sequential rules between API calls patterns as well as maximal and closed frequent API calls patterns are discovered. Obtained frequent patterns are then used for the classification of different malware. Seven classifiers are used and their performance is compared by using various metrics. Moreover, the performance of MalSPM is compared with existing state-of-the-art malware detection approaches and obtained results show that MalSPM outperforms these approaches.

Keywords: Malware, API calls, Sequential pattern mining, Frequent patterns, Classification.

1 Introduction

The name malware (malicious software) is used collectively to denote software designed primarily to access restricted data, block data to get ransom and damage or destruct computers and mobile devices without the owners' knowledge or

permission [15,57]. Since its emergence in the late 1980s, malware pose a serious threat to computing systems and networks. Although computing devices have become more secure with each new Operating System (OS) update, attackers can still bypass these security components by using different methods such as anti-debugging, anti-virtualization, obfuscation, packing, and tunneling. Due to the novel coronavirus (COVID-19) pandemic, cybercrime is up by 600%[1] and cybercrime by the end of 2021 and 2025 is expected to cost the world approximately $6 and $10.5 trillion respectively[2].

As malware can cause enormous loss and adverse effects, their analysis and detection, as early as possible, is an important research topic in cybersecurity. Anti-malware software [27] that provides protection against malware mainly uses signature-based methods for malware detection. During the scanning process, these tools search for unique signatures (short and unique string set) that are already extracted from known malicious files. An executable file is identified as a malicious code if its signature matches with the list of available signatures. This approach is fast in identifying known malware. However, creating signatures is a tedious and time-consuming activity and requires funds and expertise. Moreover, signature-based methods look for already known malicious patterns, thus they are unreliable and ineffective against new malicious codes [15]. Nowadays, new malware samples are created at a high speed. For example, AV-TEST Institute found 72.56 million new malware programs in the second half of 2021[3].

To overcome the drawbacks of signature-based detection methods, various approaches were developed to analyze unknown executables that can be divided into three types: (1) static, (2) dynamic and (3) hybrid [5,25,49,55]. Static malware detection approaches investigate the files in binary format without actually running them. This means that important features from the files such as string patterns, byte sequences and operation codes are extracted, that are then used to check whether a file is malware or not. However, these techniques can be easily bypassed by using obfuscation techniques [51] and does not work well on polymorphic malware. Dynamic malware detection approaches, on the other hand, analyze the behavior of malware during execution. This means that malware is executed in a safe isolated environment and its behavior is monitored during real time execution. In dynamic analysis, various important features of malware can be extracted such as network behavior, system calls, memory usage and registry change [45]. Hybrid malware detection analyzes both static and dynamic attributes of malware.

The focus in this study is to analyze and detect/classify Windows based-metamorphic malware. Windows OS is by far the most popular desktop OS with a market share of about 72.98% [52]. On the Windows OS, malware programs (just like ordinary software) must use some of the OS's services. The entire set of requests made by a malware to get these services through application programming interface (API) calls creates malicious behavior. On the other had,

[1] purplesec.us/resources/cyber-security-statistics
[2] cybersecurityventures.com/cybercrime-damage-costs-10-trillion-by-2025
[3] av-test.org/en/statistics/malware

metamorphic malware that are quite advanced and considered the most challenging malware type for cybersecurity. A metamorphic malware program uses code obfuscation to transform itself during each iteration by translating, editing and rewriting its own code without changing its functionality [8, 26]. These malware programs can evolve and can have numerous forms. This reduces the effectiveness of signature-based methods for detecting metamorphic malware. Such malware can generate different sequences in different environments, but they demonstrate similar behavioral features in all environments. API calls sequences thus provide important knowledge about the behavioral features of metamorphic malware.

In literature, frequent pattern mining (FIM) [38] and association rule mining (ARM) [60] have been used on the API calls for malware behavior analysis and classification [14, 44, 46, 48, 58, 59]. However, most FIM and ARM algorithms do not take into account the sequential ordering of events (API calls). Moreover, handling a high number of discovered patterns/rules and finding effective ones in them for classification is a challenging and time-consuming task. Thus, sequential pattern mining (SPM) [21] is used in this work to first analyze API calls of malware by considering their sequential ordering. Second, not all but a portion of discovered frequent API calls patterns are used for malware classification.

Based on the analysis of API calls sequences, we design a novel malware detection system called MalSPM (Malware analysis and detection via Sequential Pattern Mining), which offers:

1. An SPM-based approach for the analysis of API calls sequences of various malware that were run in an isolated secured environment. The basic idea is to first convert the API calls sequences into a corpus that is suitable for learning, where each API call is converted into an integer code. SPM techniques are then applied on the corpus to identify frequent API calls patterns. Moreover, sequential rules between API calls patterns as well as maximal and closed frequent API calls patterns are discovered.
2. A malware detection approach that first takes frequent API calls patterns, discovered using SPM algorithms, from various malware programs. These patterns are then used for the classification of programs into several malware types (also called families or classes). Several classifiers are used for classification and extensive experiments are performed with various evaluation metrics to assess the efficiency of the proposed classification approach. Obtained results show that the MalSPM performs better than existing state-of-the-art malware detection approaches.

The remainder of this paper is organized as follows. Section 2 describes the related work on using data mining and machine learning for malware analysis and detection. Then, details about the dataset used in this work are provided in Section 3. Thereafter, the proposed MalSPM system is presented, which is used for the analysis of API calls and the classification of various malware types in Section 4. Section 5 presents and discusses the obtained results. Moreover, the performance of MalSPM for classification is compared with recent approaches. Finally, Section 6 concludes the paper, and discusses research opportunities for future work.

2 Related Work

Hofmey et al. first used the API as a feature [29] in a method for anomaly intrusion detection based on sequences of system calls. API calls were also used by Ye et al. [58, 59] to develop intelligent malware detection systems (IMDS and CMIDS). The Apriori [2] and FP-Growth [28] algorithms were used to find frequent sets of API calls. But IMDS and CMIDS can only be used to detect/predict whether a PE file is malicious or not. Moreover, using all the frequent sets of API calls in classification is infeasible. The malware detection framework of Ahmadi et al. [3] used frequent iterative patterns to extract features from API calls. D'Angelo et al. [14] used association rules between API calls to model and classify malicious behaviors. Frequent API names and their arguments were used by Qiao et al. [46] to represent the behavior of a malware. The Apriori algorithm [2] was used to mine frequent itemsets composed of API names and/or API arguments. However, the Apriori algorithm does not take into account the sequential ordering of events, such as API calls. Thus, it fails to discover important patterns and also ignore the sequential relationship between events.

The malware classification framework based on API calls of Sami et al. [48] used closed frequent API calls patterns as a feature set. The CloSpan algorithm [56] was used to find frequent closed API calls. However, their approach does not take into consideration the call sequences. Moreover, the malware detection method of Petkas et al. [44] used the CM-SPAM [17] algorithm to discover frequent API calls patterns. Three classifiers were then used on the discovered patterns for the classification of malware samples. For a *minsup* threshold of 50%, frequent patterns with minimum and maximum lengths of 4 and 8 were discovered. The dataset was not discussed in detail and no link was provided to download the dataset. Another issue for FIM/ARM-based malware detection approaches is that the presence of similar patterns/rules in different malware families could lead to an incorrect or overestimated classification.

Another common approach for malware analysis and classification is to use sequence alignment on API calls. To capture the common API sequences among different malware types, the longest common sequence (LCS) was used [32] and sequence alignment (SA) techniques were used [11–13, 33, 34] to find common API calls sequences and their classification into different malware families. The clustering framework [4] used three distance measures (Optimal Matching (OM), LCS and Longest Common Prefix (LCP)) on API calls of malware to discover similar behaviors among them. However, these techniques for sequence alignment based malware analysis and detection only consider the common (sub)sequences of API calls for malware, which may have no or very little similarity with the sequence of target malware.

Recently, deep learning has been used extensively for the analysis and classification of malware [10, 30, 36, 37, 49, 50, 54]. Most of the aforementioned studies used a single or a few types of dynamic features for malware detection. Kim et al. [35] proposed a multimodal deep neural network model that utilizes various static features for Android malware detection. Khasawneh et al. [31] showed that using an ensemble detector composed of specialized detectors can perform better

than a generic detector for malware classification. Ficoo [16] proposed a hybrid approach that combines generic and specialized malware detectors to improve the detection rate of unknown malware types. The alpha-count mechanism was also used to examine the effect of observation time window on the detection accuracy and speed. Some detailed reviews on malware detection using machine learning have been published [43,51,54,57]. A recent systematic literature review (SLR) [7] identified the common pitfalls and challenges in malware research.

The key differences between previous studies and this work are that we use a dataset to study API calls of metamorphic malware rather than other types of malware that are easier to detect. Moreover, not only one but several SPM algorithms are used on this dataset to discover frequent API calls and their patterns, sequential rules between discovered patterns as well as closed and maximal sequential patterns of API calls. The frequent patterns of different lengths are then used for classification. Using more than one SPM algorithm in the analysis and classification allows us to compare their effectiveness for the detection of API calls sequence of malware.

3 The Dataset

The benchmark dataset [9, 10] contains Windows API calls of various metamorphic malware such as WannaCry and Zeus. An application running on Windows uses APIs to utilize a function that the OS provides. The use of an OS function is called an API call. During execution, an application can make many API calls. For example, to create a file, an application may call the *CreateFileA* API. Similarly, the API call *DeviceIoControl* is used by an application to perform direct input and output operations on, or retrieve information about, a specified device driver. Thus, API calls are generally used in dynamic malware analysis.

In the dataset of Catak et al. [9, 10], approximately 20,000 malware were collected from GitHub and their MD5 (Message-Digest algorithm 5) hash values were obtained. The Cuckoo Sandbox Environment[4] was then used to check the behavior of each malware separately in an isolated environment. The final data in the dataset contains all Windows API calls requests made by the malware on the Windows 7 OS. The API calls sequences are further filtered and rows in the sequences were discarded that did not contain at least 10 different API calls. The hash values of malware were searched with the VirusTotal[5] service and the analysis results from the VirusTotal service was stored in a database. Each malware's families was identified by using each analysis result that was obtained with VirusTotal. The final dataset contains 8 different leading malware families, which are [9]:

1. *Adware*: Hides on a computing device and serves advertisements.
2. *Backdoor*: Bypasses the system security mechanism and stays undetected to get access to a computing device.

[4] cuckoosandbox.org
[5] virustotal.com

3. *Downloader*: Download and install contents.
4. *Dropper*: Secretly carries malware such as viruses and backdoors so that they can be executed on a computing device.
5. *Virus*: Spreads from one computing device to another and can replicate itself.
6. *Worms*: Spreads copies of itself from one computing device to another.
7. *Trojan*: Misleads a user about its true intent.
8. *Spyware*: Enables one to get covert information about another computing device activities by transmitting data covertly from their hard drive.

Table 1: Malware distribution according to their families

Malware Family	Samples	API Calls	MinL	MaxL	ASL
Spyware	832	229	14	1,764,421	46,951
Downloader	1001	232	14	870,719	6,522
Trojan	1001	255	12	1,232,913	13,818
Worms	1001	236	10	1,245,582	33,614
Adware	379	212	19	1,450,685	6,867
Dropper	891	226	12	1,068,329	16,008
Virus	1001	241	13	1,062,231	18,370
Backdoor	1001	227	12	1,402,652	11,293
Total	7101	278	13.25	1,187,191	19,647

Table 1 shows the number of malware programs that belong to each of the eight malware families in the dataset. The whole dataset contains 7,101 sequences in total. Moreover, the total distinct API calls, the minimum length (MinL), maximum length (MaxL) and average sequence length (ASL) of API calls sequences in each malware type is presented in Table 1. From this we can see that on average, a sequence contains approximately 20,000 API calls.

4 The MalSPM System

The proposed MalSPM system is designed for the analysis of API calls sequences made by various malware programs and their classification. This process is depicted in Figure 1, and consists of three steps:

1. *Corpus development*: The first step consists of converting different malware API call datasets into a suitable format for SPM. The sequences of API calls made by malware are transformed in a suitable abstraction, so that no meaningful information from the sequences are left out. For this, we use the "*API calls sequences to integers*" abstraction, where each API call type is converted into a distinct positive integer. Such abstraction allows wide diversity and makes the approach more general in nature.
2. *Learning through SPM*: SPM algorithms are applied on the corpus to discover common API calls and their frequent sequences, sequential relationships between frequent API calls, patterns revealing their relationships in the form of rules, and frequent closed and maximal frequent API calls [41].

3. *Classification through sequential patterns*: Frequent patterns identified by various SPM algorithms in Step 2 are used for the classification of different malware types. The classification task is composed of two main parts: (1) The training phase contains two phases, frequent API calls representation and classifier training, that are performed sequentially. (2) The testing phase contains three phases, frequent API calls representation, hypothesis prediction and evaluation. Both integer-based and string-based representation are used for the representation of frequent API calls in the classification process.

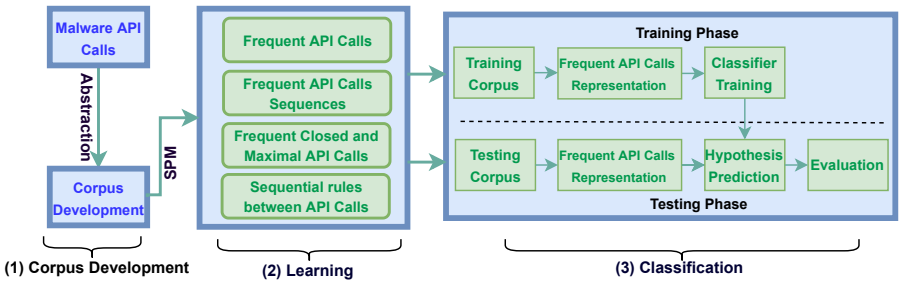

Fig. 1: The three step process of MalSPM for malware behavior analysis and classification

The next three subsections provide more explanation for the three parts.

4.1 Corpus Development

The first step is to collect and represent the data as sequences of API calls in an appropriate format. The next paragraphs first introduce key concepts related to sequences.

Let $AC = \{ac_1, ac_2, ..., ac_m\}$ represent the set of API calls. An *API calls set* ACS is a set of API calls such that $ACS \subseteq AC$. The notation $|ACS|$ denotes the set cardinality. An API calls set ACS has a length k (called k-ACS) if it contains k API calls, i.e., $|ACS| = k$. For example, consider the set of API calls $AC = \{$*LdrLoadDll, NtOpenKey, CopyFileA, FindResourceA, GetFileSize, Exception*$\}$. Then, the set $\{$*LdrLoadDll*, *NtOpenKey*, *Exception*$\}$ is an API calls set that contains three API calls. To facilitate the discovery of patterns, a total order relation is defined on API calls, denoted as $\prec$. This order is used as processing order to search for patterns and is the lexicographical order, in the system implementation.

An API calls sequence is a list of API calls sets $S = \langle ACS_1, ACS_2, ..., ACS_n \rangle$, such that $ACS_i \subseteq ACS$ $(1 \leq i \leq n)$. An *API call corpus* ACD is a list of API call sequences $ACD = \langle S_1, S_2, ..., S_n \rangle$, where each sequence has a unique identifier (ID). For example, Table 2 shows an ACD that has four

API calls sequences with IDs 1, 2, 3 and 4. The first sequence indicates that the *LdrLoadDll* API was first called, followed by *RegOpenKeyExA*, then *NtOpenKey*, and finally *NtQueryValueKey*.

Table 2: A sample of an API calls corpus

ID	API call sequence
1	⟨{*LdrLoadDll, RegOpenKeyExA, NtOpenKey, NtQueryValueKey*}⟩
2	⟨{*NtClose, DeviceIoControl, NtClose, NtClose, NtReadFile, WSAStartup*}⟩
3	⟨{*NtCreateFile, GetFileSize, NtClose, NtReadFile, DrawTextExA, NtDelayExecution, NtClose, GetKeyState*}⟩
4	⟨{*FindWindowExW, FindFirstFileExA, OpenKey, NtClose, NtClose, NtCreateFile, LdrGetProcedureAddress, GetAsyncKeyState, GetKeyState, DeviceIoControl, NtClose, NtDelayExecution*}⟩

After collecting API calls sequences, they are converted into sequences of integers to bring the dataset in a more suitable format for SPM algorithms. In the final corpus, each line represents an API calls sequence for a malware. Each API call type in sequences is replaced by a unique positive integer. For example, the API call *NtClose* is replaced by 6. Moreover, API calls are separated by a single space followed by a negative integer -1 and a negative integer -2 appears at the end of each line to indicate the end of the sequence. The four API calls sequences of Table 2 are converted into the integer sequences shown in Table 3.

Table 3: Representation of API calls sequences as integer sequences

ID	API call sequence
1	1 -1 3 -1 4 -1 5 -1 -2
2	6 -1 12 -1 6 -1 6 -1 32 -1 23 -1 -2
3	7 -1 11 -1 6 -1 32 -1 15 -1 18 -1 6 -1 22 -1 -2
4	19 -1 22 -1 14 -1 6 -1 6 -1 7 -1 19 -1 31 -1 22 -1 12 -1 6 -1 18 -1 -2

4.2 Learning using SPM

The second step is to extract patterns from the API calls sequences. An API calls sequence $S_\alpha = \langle \alpha_1, \alpha_2, ..., \alpha_n \rangle$ is present or contained in another API calls sequence $S_\beta = \langle \beta_1, \beta_1, ..., \beta_m \rangle$ iff there exist integers $1 \leq i_1 < i_2 < ... < i_n \leq m$, such that $\alpha_1 \subseteq \beta_{i1}, \alpha_2 \subseteq \beta_{i2}, ..., \alpha_n \subseteq \beta_{im}$ (denoted as $S_\alpha \sqsubseteq S_\beta$). If S_α is present in S_β, then S_α is a *subsequence* of S_β. In SPM, various measures are used to investigate the importance and interestingness of a subsequence. The *support* measure is used by most SPM techniques. The *support* of an API calls sequence S_α in an ACD is the total number of sequences (S) that contain S_α, and is represented by $sup(S_\alpha)$:

$$sup(S_\alpha) = |\{S | S_\alpha \sqsubseteq S \wedge S \in ACD\}|$$

SPM is an enumeration problem that aims at identifying all the *frequent subsequences* in a sequential dataset. A sequence S is a *frequent sequence* (also called *sequential pattern*) iff $sup(S) \geq minsup$, where $minsup$ (minimum support) is a threshold determined by the user. A sequence containing n items (API calls in this work) in a corpus can have up to $2^n - 1$ distinct subsequences. This makes the naive approach to calculate the support of all possible subsequences infeasible for most corpora [42]. Several efficient algorithms have been developed in recent years that do not explore all the search space of all possible subsequences but can guarantee finding all sequential patterns.

The search space of sequential patterns is traversed by SPM algorithms using two operations: *s-extensions* and *i-extensions*. A sequence $S_\alpha = \langle \alpha_1, \alpha_2, ..., \alpha_n \rangle$ is a *prefix* of another sequence $S_\beta = \langle \beta_1, \beta_1, ..., \beta_m \rangle$, if $n < m$, $\alpha_1 = \beta_1$, $\alpha_2 = \beta_2$, ..., $\alpha_{n-1} = \beta_{n-1}$, where α_n is equal to the first $|\alpha_n|$ items of β_n according to the $\prec$ order. Note that SPM algorithms follow a specific order $\prec$ so that the same potential patterns are not considered twice and the choice of the order $\prec$ does not affect the final result produced by SPM algorithms. A sequence S_β is an *s-extension* of a sequence S_α for an item x if $S_\beta = \langle \alpha_1, \alpha_2, ..., \alpha_n, \{x\} \rangle$. Similarly, for an item x, a sequence S_γ is an *i-extension* of S_α if $S_\gamma = \langle \alpha_1, \alpha_2, ..., \alpha_n \cup \{x\} \rangle$. SPM algorithms generally either employ a breadth-first search or depth-first search. In the following, a brief overview of some state-of-the-art SPM algorithms is presented.

A special case of SPM is frequent itemset mining (FIM). It consists of analyzing records without considering the sequential order between items (API calls). The first and most popular algorithm for FIM is Apriori [2]. It can find frequent itemsets (e.g. sets of API calls) in large databases. It proceeds by first discovering items that frequently occur. Then, these items are extended to find larger itemsets that appear sufficiently often. From the itemsets found by Apriori, one can also derive association rules (relationships) between items. In the last decades, several memory efficient and fast FIM algorithms have been proposed. They generally find the same patterns but utilize different data structures, optimizations and search strategies.

For the general problem of SPM, the TKS (Top-k Sequential) algorithm [18] finds the top-k sequential patterns in a corpus, where k is set by the user. The parameter k represents the number of sequential patterns to be extracted by the algorithm. TKS employs the basic candidate generation procedure of the SPAM algorithm [6] and a vertical database representation. A vertical representation is used to facilitate counting the support of patterns by avoiding costly database scans. This makes vertical SPM algorithms perform better on dense or long sequences. TKS also uses many strategies for search space reduction and utilize a PMAP (Precedence Map) data structure to avoid costly operations of bit vector intersection. Another representative SPM algorithm is CM-SPAM [17], which examines the whole search space to discover frequent sequential patterns in the corpus. The CMAP (Co-occurrence MAP) data structure is used by CM-SPAM to store information about item co-occurrences. A generic pruning mechanism

that is based on CMAP is used for pruning the search space using the vertical database representation, to efficiently discover sequential patterns.

Thought the above SPM algorithms have many applications, a drawback is that they may find too many sequential patterns for users. Browsing through thousands of sequential patterns can be time consuming and some information may be even deemed as redundant. To address this issue, compact representations of sequential patterns have been studied, which are small sets of sequential patterns that summarize all sequential patterns. The complexity of pattern extraction can be reduced by finding these compact representations. The two main compact representations are the closed and maximal sequential patterns. A sequential pattern s_a is said to be *closed* if there is no other sequential pattern s_b, such that s_b is a superpattern of s_a, that is $s_a \sqsubseteq s_b$, and their supports are equal. A sequential pattern s_a is said to be *maximal* if there is no other sequential pattern s_b, such that s_b is a superpattern of s_a, that is $s_a \sqsubseteq s_b$. The problem of mining closed (maximal) sequential patterns is to discover the set of closed (maximal) sequential patterns. CloFAST (Closed FAST sequence mining algorithm based on sparse id-lists) [24] and VMSP (Vertical mining of Maximal Sequential Patterns) [22] are two efficient algorithms used for finding closed and maximal sequential patterns, respectively, in large databases. The VMSP algorithm incorporates three efficient strategies named EFN (Efficient Filtering of Non-maximal patterns), FME (Forward Maximal Extension checking) and CPC (Candidate Pruning by Co-occurrence map) to effectively identify maximal patterns and prune the search space.

Sequential patterns that appear frequently in a corpus with low confidence are worthless for decision-making or prediction. For this reason, another pattern type called sequential rules can be discovered [19]. A sequential rule is a pattern that consider not only the support of events but also the confidence (conditional probability) that some events will be followed by others. A sequential rule $X \rightarrow Y$ is a relationship between two ACSs $X, Y \subseteq AC$ such that $X \cap Y = \emptyset$ and $X, Y \neq \emptyset$. A rule $r : X \rightarrow Y$ means that if items (API calls) of X occur in a sequence, items of Y will occur afterward in the same sequence. X is contained in S_α (written as $X \sqsubseteq S_\alpha$) iff $X \subseteq \bigcup_{i=1}^{n} \alpha_i$. A rule $r : X \rightarrow Y$ is contained in S_α ($r \sqsubseteq S_\alpha$) iff there exists an integer k such that $1 \leq k < n$, $X \subseteq \bigcup_{i=1}^{k} \alpha_i$ and $Y \subseteq \cup_{i=k+1}^{n} \alpha_i$. The confidence of r in ACD is defined as:

$$conf_{ACD}(r) = \frac{|\{S | r \sqsubseteq S \wedge S \in ACD\}|}{|\{S | X \sqsubseteq S \wedge S \in ACD\}|}$$

The support of r in ACD is defined as:

$$sup_{ACD}(r) = \frac{|\{S | r \sqsubseteq S \wedge S \in ACD\}|}{|ACD|}$$

A rule r is a *frequent sequential rule* iff $sup_{ACD}(r) \geq minsup$ and r is a *valid sequential rule* iff it is frequent and $conf_{ACD}(r) \geq minconf$, where the thresholds $minsup, minconf \in [0, 1]$ are set by the user. Mining sequential rules in a corpus aims to enumerate all the valid sequential rules. For this, the ERMiner

(Equivalence class based sequential Rule Miner) algorithm [19] is used. It relies on a vertical database representation and the search space of rules is explored using equivalence classes of rules with the same antecedent and consequent. ERMiner employs two operations (left and right merges) to explore the search space of frequent sequential rules, where the search space is pruned with the Sparse Count Matrix (SCM) technique, which makes ERMiner more efficient than prior algorithms for sequential rule mining.

4.3 Malware Classification using SPM

The third step performed by the proposed MalSPM system is to classify software programs according to different malware types using the sequential patterns.

API calls sequences of malware are generally very long and in most sequences, the same API calls are repeated many times. In fact, a close inspection of the dataset revealed that many sequences contained the same API calls hundreds or even thousands of times, repeated consecutively. We believe that this huge repetition of API calls in sequences is redundant and such repetition can be avoided without affecting the classification performance. Thus, in the proposed MalSPM system, contiguous identical API calls are treated as a single call. This is similar to prior work [36] where contiguous identical API calls were replaced by a single one, and as a result subsequences and the trained classifiers provided better classification performance.

More precisely, the MalSPM system utilizes the frequent patterns obtained using various SPM algorithms for the classification of different malware types. For classification, two methods are used, which are binary classification and multi-class classification.

Binary Classification Following the approach of Catak et al. [10], the whole corpus of malware sequences can be used to train a model to classify each malware type separately. For a selected malware type, binary classification assigns "1" to each sequence corresponding to that type and label all other sequences as "0" using Equation 1.

Definition 1. *Let C be the set of all malware classes (types). For a selected malware class $c \in C$, a sequence Y is labeled with respect to c as:*

$$Y_c = \begin{cases} 1, & \text{if } y = c, \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

According to equation 1, class labels that belong to c will be labeled as 1, while others are labeled as 0 to then train a binary classifier. For example, if the malware type of interest is Adware, Equation 1 will assign 1 to API calls sequences that belong to the Adware type and 0 to API calls sequences that belong to other malware types.

Multi-Class Classification A second way of training models for malware detection is using multi-class classification. For this, in the context of this study, each API call sequence in the corpus is labeled with its respective class name. In total, we have 8 different classes of malware as indicated in Table 1. Then, a model (classifier) can be trained to correctly label sequences.

Evaluation metrics We use five commonly used metrics to evaluate the performance of classifiers, which are: (1) accuracy, (2) recall, (3) precision, (4) F1 score and (5) Matthews correlation coefficient (MCC). In this work, the accuracy (ACC) is defined as the proportion of correctly classified malware types divided by the total number of malware types. The formal definition of ACC is:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{2}$$

whereas in the context of this paper,
TP stands for true positive, i.e. number of software programs correctly identified as belonging to a given malware type
FP stands for false positive, i.e. number of software programs incorrectly identified as belonging to a given malware type.
FN stands for false negative i.e. number of software programs incorrectly identified as not belonging to a given malware type, and
TN stands for true negative i.e. number of software programs correctly identified as not belonging to a given malware type.

The other four measures, precision, recall, F1 and MCC are calculated as follows:

$$Precision(P) = \frac{TP}{TP + FP} \tag{3}$$

$$Recall(R) = \frac{TP}{TP + FN} \tag{4}$$

$$F1 = 2 \times \frac{P \times R}{P + R} \tag{5}$$

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{6}$$

Classifiers For both binary and multi-class classification, seven popular machine learning algorithms are used, which are: (1) SVM (Support Vector Machine), (2) Random Forest (RF), (3) kNN (k-Nearest Neighbors), (4) NB (Naive Bayes), (5) J48 (Decision Tree), (6) NBMT (Naive Bayes Multinomial Text), and (7) SGDT (Stochastic Gradient Descent Text) [1, 40, 53]

It is important to point out here that some classifiers such as RF and J48 (that implements the famous ID3 algorithm) are decision tree-based classifiers. Five classifiers (SVM, RF, kNN, NB and J48) are integer-based classifiers.

Whereas, NBMT and SGDT work on text (string) data only. For NBMT and SGDT, we created a separate corpus where integer values in the sequence are replaced by the respective API calls.

For tokenization in string-based classifiers, we used *WordTokenizor* and *NGramTokenzior* or *CharacterNGramTokenzior*. The *WordTokenizor* is a simple tokenizor that is used to tokenize the strings. The *NGramTokenizor* splits a string into an *n*-gram with user specified minimum and maximum grams. Whereas, the *NGramTokenizor* splits a string into all character *n-grams* it contains on the basis of user specified maximum and minimum for *n*.

The main reason to use classifiers that only works on strings is to check whether they performed better than integer-based classifiers or not. The classifiers are trained using 80 (75)% of the data during the training phase. Validation is performed during training with 20 (25)% of the data allocated for training.

5 Results and Discussion

A set of experiments were performed on an HP laptop with a fifth-generation Core i5 processor and 8 GB RAM. The SPMF data mining library, developed in JAVA, is used to analyze the API calls corpus. It is an open-source cross-platform framework that is specialized in pattern mining tasks. SPMF offers implementations of more than 230 data mining algorithms [20]. To train classifiers, we used the open-source WEKA software [23], also developed in Java. We have chosen WEKA due to several reasons, such that it can run on any platform and it offers not only machine learning classifiers but also data preparation tools and meta learners. Moreover, along with its command-line interface, WEKA also provides a GUI interface that is very easy to use.

The MalSPM system first uses SPMF to convert the API call sequences into integer sequences that is suitable for SPM algorithms. The first 1000 sequences contain 254 distinct API calls in total and on average, each sequence contains approximately 11,502 API calls. Results obtained by applying SPM algorithms on the corpus that contains the first 1000 sequences are discussed next. Interested readers can find the converted datasets used in this work in an online repository: *github.com/saqibdola/MBAwSPM*.

5.1 Frequent API Calls Patterns and Sequential Rules

Frequent API Calls Set. The Apriori algorithm was applied on the corpus to find frequently occurring API calls. Apriori takes a corpus and a *minsup* threshold as input and outputs the frequent API calls. The number of sets extracted by Apriori are listed in Table 4. For high *minsup* values, Apriori generated less frequent patterns. By decreasing *minsup* to 10%, Apriori generated 274 patterns. Whereas the time and memory used by Apriori increased as *minsup* was decreased. The top five frequent API call sets in 1000 sequences were *GetAsyncKeyState*, *GetHostbyName*, *Exception*, *DeviceIoControl* and *NtClose* with 2,343,660, 1,103,492, 815,056, 800,088 and 680,206 occurrences, respectively.

Table 4: Frequent API calls extracted by Apriori

Frequent API Calls Count	Min. Sup	Time (Sec)	Memory (Mb)
117	100%	7	175.35
118	90%	6.9	172.96
120	80%	7.1	179.74
123	70%	7.7	184.38
127	60%	8.2	190.64
135	50%	9.6	185.46
141	40%	10.4	187.05
155	30%	15.2	182.95
170	20%	17.4	181.44
274	10%	61.4	370.46

The frequent API call sets obtained with Apriori are uninteresting in the sense that they are unordered as they do not follow any specific order. Moreover, Apriori does not ensure that API calls from a API calls set are contiguous in the sequence. Apriori fails to discover important sequential patterns and also ignore the sequential relationship between API calls. Next, we present the results from the application of SPM algorithms that overcome the drawbacks of Apriori. Thus they reveal more meaningful patterns and information.

Frequent Sequential API Calls Patterns. The TKS algorithm is applied on the corpus to find the top-k sequential patterns of API calls. TKS takes a corpus and a user-specified parameter k as input and returns the top-k most frequent patterns as output. Some API call frequent patterns discovered by the TKS algorithm with varying length are shown in Table 5. Note that the column **Sup** indicates the occurrence count of each pattern in the corpus. Table 5 provides some useful information related to frequent occurrences of API calls and patterns in sequences. The second last row shows that the same API call *NtClose* as a sequence of six occurrences appeared 920 times in the corpus. Similarly, the API call in the last row *LdrGetProcedureAddress* occurred as a sequence of ten occurrences 880 times in the corpus. The time taken by TKS to find the top k-patterns increased from 6.5 seconds to 19.5 seconds when k was increased from 50 to 300.

Table 5: Extracted API call patterns with TKS algorithm

Pattern	Sup
NtClose, NtQueryValueKey, NtClose	919
LdrLoadDll, LdrGetProcedureAddress, NtClose	904
NtClose, NtOpenKey, NtQueryValueKey, NtClose	915
NtClose, NtOpenKey, NtClose, NtClose, NtClose	916
NtClose, NtOpenKey, NtClose, NtClose, NtClose, NtClose	916
NtAllocateVirtualMemory, NtClose, NtClose	882
6 × *NtClose*	920
10 × *LdrGetProcedureAddress*	880

Unlike TKS, the CM-SPAM algorithm requires setting the *minsup* threshold. Table 6 lists some of the most useful frequent API calls patterns in the corpus which are extracted by the CM-SPAM algorithm. The first four API calls patterns appear in at least 90% of the sequences in the corpus. The next four patterns appear in at least 80% of the sequences. Patterns identified by the CM-SPAM algorithm are almost the same as those found by the TKS algorithm. It was found that API call *NtClose* appeared in almost every frequent pattern discovered using either TKS and CM-SPAM. Moreover, from 254 total API calls, approximately 15 API calls appeared in most of the frequent patterns. The time taken by CM-SPAM to find patterns increased from 8.7 seconds to 30.4 seconds when *minsup* was increased from 95% to 80%, respectively.

Table 6: Frequent API call patterns extracted with CM-SPAM

Pattern	**Min. Sup**	**Sup**
NtClose, NtQueryvalueKey, NtClose	0.9	919
NtClose, NtQueryValueKey, NtqueryValueKey, NtClose	0.9	915
NtOpenKey, NtClose, NtClose, NtClose	0.9	934
$7 \times$ *NtClose*	0.9	900
NtCreateFile, NtClose, NtClose	0.8	803
LdrGetDllHandle, LdrGetDllHandle, NtClose, NtClose	0.8	841
NtFreeVirtualMemory, NtClose, NtClose	0.8	803
$12 \times$ *NtClose*	0.8	816

Some of the discovered closed and maximal sequential API calls patterns are listed in Table 7. The closed sequential patterns provide a lossless representation of all sequential patterns and they represent the largest subsequences common to sets of sequences. For example, if each sequence represents the behavior of a malware, the closed patterns represent the largest patterns common to groups of malware sequences. Whereas the maximal API calls patterns are not lossless and is never larger than the set of closed sequential patterns and the set of all sequential patterns. Note that in maximal sequential API patterns, the *max gap* is set to 1. This means that no gap is allowed and each consecutive API call of a pattern must appears consecutively in a sequence. The time taken by CloFast (VMSP) to find patterns increased from 15.64 (9.6) seconds to 21.2 (31.3) seconds when *minsup* was increased from 95% to 80%, respectively.

Sequential Rules. Table 8 shows the relationships between frequent API calls that are found through sequential rule mining with the ERMiner algorithm. The confidence ($misconf$) threshold is set to 80%, which means that rules must have a confidence of at least 80% (a rule $X \rightarrow Y$ has a confidence of 80% if the set of API calls in X is followed by the set of API calls in Y at least 80% of the times when X appears in a sequence). For example, the first rule in Table 8 indicates that

Table 7: Closed (C) and Maximal (M) frequent sequential API patterns

Closed	**Sup**
NtAllocateVirtualMemory, NtAllocateVirtualMemory	729
NtClose, NtOpenKey, NtQueryValueKey	755
NtFreeVirtualMemory, NtClose, NtClose	803
NtOpenKey, NtQueryValueKey, NtClose, NtClose	700
LdrGetDllHandle, 4 × LdrGetProcedureAddress	702
16 × LdrGetProcedureAddress	702

Maximal	**Sup**
LdrLoadDll, LdrGetDllHandle	842
LdrLoadDll, LdrLoadDll, LdrGetDllHandle	814
LdrGetDllHandle, LdrGetProcedureAddress, NtOpenKey	806
LdrLoadDll, NtClose, NtOpenKey	817
3 × LdrGetProcedureAddress, LdrLoadDll, LdrGetDllHandle	807
36 × LdrGetProcedureAddress	801

94.5% of the time, the API call *LdrLoadDll* follows *LdrGetProcedureAddress* and this rule has occurred 898 times in the corpus. Similarly, the second rule is the opposite of the first rule: 98.9% of the time, the API call *LdrGetProcedureAddress* is followed by *LdrLoadDll* 917 times in the corpus. With ERminer, we found some interesting relationships and dependencies between frequent API calls.

Table 8: Discovered sequential rules

Rule	**Sup**	**Conf**
LdrGetProcedureAddress → LLdrLoadDll	898	0.945
LdrLoadDll → LdrGetProcedureAddress	917	0.989
NtClose → LdrLoadDll	859	0.86
LdrLoadDll → NtClose	913	0.98
LdrLoadDll, NtAllocateVirtualMemory → LdrGetProcedureAddress	859	0.97
LdrLoadDll, LdrGetProcedureAddress → NtAllocateVirtualMemory	841	0.91
LdrLoadDll, NtAllocateVirtualMemory → NtClose	871	0.98
LdrLoadDll, NtClose→ NtAllocateVirtualMemory	810	0.88
LdrLoadDll, LdrGetProcedureAddress, NtQueryValueKey → NtClose	855	0.99
NtClose →LdrLoadDll, LdrGetProcedureAddress, NtQueryValueKey	808	0.81
LdrGetProcedureAddress, NtClose → NtOpenKey, NtQueryValueKey	834	0.88
LdrGetProcedureAddress, NtOpenKey → NtQueryValueKey, NtClose	840	0.93
LdrLoadDll, NtOpenKey, NtQueryValueKey, LdrGetDllHandle → NtClose	808	0.991

During execution, we found that all the algorithms worked efficiently on the corpus and results obtained indicated that the total number of API calls in each sequence (abstraction simplicity) has a direct correlation with the efficiency of SPM algorithms.

5.2 Classification Results

In this section, experimental results for binary and multi-class classification of malware are discussed respectively, followed by a comparison with results from recent studies.

For classification, four SPM algorithms (TKS, CM-SPAM, VMSP and CloFast) are used to find frequent 100 and 200 patterns for each malware type. The discovered patterns are pre-processed further to make sure that:

- The discovered patterns contain at least 3 distinct frequent API calls, and
- The longest patterns contain a maximum of 10 frequent API calls for TKS, CM-SPAM and CloFast. For VMSP, the longest patterns contain a maximum of 15 frequent API calls.

The main reason to consider two different numbers of discovered patterns (100 and 200) with SPM is to see whether the pattern count has any effect on the performance of classifiers.

Binary classification. The binary classification results for patterns discovered with TKS, CM-SPAM, VMSP and CloFast are provided in Tables A1, A2, A3 and A4. To reduce the number of tables, we provided classifier results for 100 and 200 patterns in each table for TKS, CM-SPAM, VMSP and CloFast. The classification results for five classifiers (SVM, RF, kNN, NB and J48) obtained on 100 and 200 patterns that are discovered with TKS and CM-SPAM is provided in Table A1 of the Appendix. Whereas, Table A2 provides results of five classifiers on 100 and 200 patterns that are discovered with VMSP and CloFast. Similarly, Tables A3 and A4 provide classification results for two classifiers (NBMT and SGDT) using 100 and 200 patterns that are discovered with TKS, CM-SPAM, and VMSP, CloFast respectively.

In each table, the results for classifier metrics are shown with the following format: $\frac{A_1 100 (A_2 100)}{A_1 200 (A_2 200)}$. For example, consider the SVM ACC of $\frac{88.7(90.6)}{(87.8)(87.7)}$ in Table A1. It indicates that ACC of 88.7% is obtained using 100 patterns discovered by TKS, 90.6% ACC is obtained using 100 patterns identified by CM-SPAM, 87.8% ACC is obtained using 200 patterns from TKS and 87.7% ACC using 200 patterns found by CM-SPAM, respectively. We used the same format throughout this section in all tables.

By analyzing the classification results on the corpus of API calls for patterns found by TKS and CM-SPAM, which use the integer format (Table A1), it was found that the tree based classifiers (RF and J48) performed better than SVM, kNN and NB. Whereas, J48 performed better than RF. The main reason that tree-based classifiers performed better is that all the patterns in the corpus are used in the classification process and each pattern only contain frequent API calls (features). There is no noise or redundant data present in the corpus. Table A3 and Table A4 provide results for classifiers that work on text data. For this, each integer in the corpus is replaced with its respective API call name. The main reason to use string-based classifiers is to check if their performance is

better than classifiers that work with an integer representation. We found that NBMT and SGDT performed better than five classifiers (SVM, NB, kNN, RF and J48). Whereas, SGDT performed better than NBMT. The six classifiers (SVM, RF, kNN, NB, J48 and NBMT) generated the same results for both 100 and 200 patterns, with negligible difference. However, NBMT results for 100 patterns were far better than for 200 patterns.

For binary classification, all the classifiers performed similarly for patterns discovered by different SPM algorithms, with negligible decrease or increase in values for various metrics. The binary classification results for two best classifiers (RF and SGDT) is provided in Table 9. It is important to point out here that strings-based classifiers require (1) the preprocessing of the dataset, and (2) to use various embedding techniques such as WordTokenizor and NGramTokenizor. For integer-based classifiers, there is no need to do these steps.

Table 9: Binary classification results for RF and SGDT on patterns extracted using TKS(CM-SPAM)

Cl	Adware	Backdoor	Downloader	Dropper	Virus	Worms	Trojan	Spyware
RF								
ACC	98.75(92.5) 89(90.6)	83.7(86.2) 86.2(88.7)	80(86.2) 82(85.2)	81(82.5) 83.1(85.2)	92.5(92.5) 90(92.1)	85(84) 81.2(85.3)	88.7(90) 83.7(85.2)	86.5(90) 81.5(83.7)
P	0.98(0.91) 0.88(0.89)	0.77(0.83) 0.83(0.86)	0.69(0.84) 0.76(0.80)	0.73(0.74) 0.79(0.80)	0.92(0.88) 0.89(0.91)	0.74(0.78) 0.74(0.75)	0.84(0.86) 0.79(0.79)	0.79(0.91) 0.75(0.76)
R	0.98(0.92) 0.89(0.90)	0.83(0.86) 0.86(0.88)	0.80(0.86) 0.82(0.85)	0.81(0.82) 0.83(0.85)	0.92(0.91) 0.90(0.92)	0.85(0.84) 0.81(0.85)	0.88(0.90) 0.83(0.85)	0.86(0.90) 0.81(0.83)
F1	0.98(0.92) 0.88(0.89)	0.80(0.84) 0.84(0.86)	0.74(0.83) 0.79(0.82)	0.76(0.77) 0.80(0.82)	0.92(0.88) 0.89(0.91)	0.79(0.81) 0.77(0.80)	0.86(0.87) 0.81(0.82)	0.81(0.87) 0.78(0.79)
MCC	0.92(0.53) 0.43(0.47)	0.007(0.26) 0.20(0.30)	−0.08(0.36) −0.03(0.09)	−0.03(−0.004) 0.01(0.09)	0.50(0.16) 0.42(0.52)	−0.04(−0.014) −0.09(−0.04)	−0.05(0.07) 0.008(0.03)	0.03(0.46) −0.05(−0.02)
SGDT								
ACC	99.5(93.1) 88.4(91.2)	88.7(88.1) 91.2(90)	84.5(89) 80(87.1)	94.3(85.5) 88.1(88.1)	94.3(93.1) 91.2(92.5)	87(86) 86.8(87.1)	90.6(90) 85.6(87.7)	86.5(87.5) 86.5(87.1)
P	0.99(0.92) 0.86(0.90)	0.87(0.85) 0.90(0.88)	0.77(0.90) 0.78(0.80)	0.93(0.87) 0.89(0.89)	0.93(0.93) 0.92(0.91)	0.77(0.83) 0.82(0.83)	0.84(0.86) 0.77(0.83)	0.79(0.89) 0.82(0.83)
R	0.99(0.93) 0.88(0.91)	0.88(0.88) 0.91(0.90)	0.84(0.89) 0.80(0.87)	0.94(0.85) 0.88(0.88)	0.94(0.93) 0.91(0.92)	0.87(0.86) 0.86(0.87)	0.90(0.90) 0.85(0.87)	0.86(0.87) 0.86(0.87)
F1	0.99(0.91) 0.86(0.89)	0.87(0.85) 0.89(0.87)	0.78(0.86) 0.79(0.82)	0.93(0.79) 0.82(0.82)	0.93(0.91) 0.88(0.91)	0.81(0.84) 0.81(0.83)	0.87(0.87) 0.80(0.83)	0.82(0.82) 0.82(0.83)
MCC	0.97(0.54) 0.32(0.50)	0.40(0.33) 0.48(0.38)	0.06(0.48) 0.05(0.06)	0.59(0.16) 0.15(0.15)	0.59(0.44) 0.33(0.51)	−0.03(0.20) 0.12(0.21)	−0.03(0.07) 0.01(0.12)	0.05(0.20) 0.01(0.21)

Multi-class Classification results The multi-class classification results for patterns discovered with TKS, CM-SPAM, VMSP and CloFast are provided in Tables A5, A6, A7 and A8.

For integer-based classifiers, RF performed better than J48, SVM and NB. Whereas the performance of kNN was better than RF on patterns discovered with TKS, CM-SPAM (Table A5) and almost similar to RF on patterns discovered with VMSP, CloFast (Table A6). The SVM and NB generated the same results, with negligible difference on 100 and 200 patterns discovered with TKS or CM-SPAM (Table A5). Whereas kNN, RF and J48 performance were better on 100 patterns as compared to 200 patterns. Similarly for 100 and 200 patterns generated with VMSP and CloFast (Table A6), SVM performance was same. Whereas NB, kNN, RF and J48 performance was better on 100 patterns

as compared to 200 patterns. The performance of all classifiers were better on patterns generated with VMSP (CloFast) than TKS (CM-SPAM).

The string-based classifier results reported in Table A7 and Table A8 are only for the NBMT classifier as SGDT cannot be used for multi-class classification. Unlike integer-based classifiers, NBMT's performance was low compared to RF, J48 and kNN. NBMT performed better than SVM and NB. Moreover, there was no significant difference in the results obtained with 100 and 200 patterns. Moreover, NBMT performance was the same on patterns discovered with TKS (CM-SPAM) and VMSP (CloFast). The multi-class classification results for the two best classifiers (RF and kNN) is provided in Table 10.

Table 10: Multi-Class classification results for RF and kNN on patterns extracted using VMSP(CloFast)

Cl	Adware	Backdoor	Downloader	Dropper	Virus	Worms	Trojan	Spyware	W. Avg.
RF									
P	0.96(0.70)	0.85(0.45)	0.80(0.42)	0.92(0.48)	0.98(0.59)	0.91(0.32)	0.78(0.38)	0.84(0.43)	0.88(0.47)
R	0.99(0.70)	0.89(0.44)	0.90(0.50)	0.96(0.60)	0.93(0.85)	0.92(0.73)	0.93(0.90)	0.94(1)	0.93(0.71)
F1	0.98(0.66)	0.80(0.39)	0.82(0.40)	0.94(0.46)	0.90(0.63)	0.88(0.42)	0.87(0.38)	0.86(0.42)	0.88(0.47)
MCC	0.98(1)	0.97(0.94)	0.92(0.65)	0.93(0.66)	0.92(0.58)	0.93(0.40)	0.91(0.42)	0.90(0.38)	0.93(0.63)
	0.97(0.68)	0.82(0.41)	0.81(0.41)	0.93(0.47)	0.94(0.61)	0.89(0.36)	0.82(0.38)	0.85(0.42)	0.88(0.47)
	0.98(0.82)	0.92(0.60)	0.91(0.57)	0.95(0.63)	0.93(0.69)	0.92(0.52)	0.92(0.57)	0.92(0.55)	0.93(0.62)
	0.96(0.63)	0.80(0.34)	0.78(0.33)	0.92(0.39)	0.93(0.55)	0.88(0.26)	0.80(0.29)	0.83(0.34)	0.86(0.39)
	0.98(0.81)	0.91(0.58)	0.90(0.50)	0.94(0.57)	0.92(0.67)	0.91(0.50)	0.91(0.58)	0.91(0.59)	0.92(0.60)
kNN									
P	0.94(0.47)	0.71(0.32)	0.74(0.39)	0.92(0.59)	0.98(0.95)	0.92(0.78)	0.97(0.88)	1(1)	0.90(0.676)
R	0.96(0.70)	0.87(0.44)	0.87(0.50)	0.95(0.60)	0.93(0.85)	0.92(0.73)	0.99(0.90)	1(1)	0.93(0.71)
F1	1(1)	1(0.91)	0.94(0.55)	0.94(0.47)	0.90(0.41)	0.88(0.11)	0.68(0.15)	0.71(0.16)	0.88(0.47)
MCC	1(1)	0.99(0.94)	0.98(0.65)	0.95(0.66)	0.93(0.58)	0.92(0.40)	0.85(0.42)	0.85(0.38)	0.93(0.63)
	0.97(0.64)	0.83(0.47)	0.83(0.46)	0.93(0.52)	0.94(0.57)	0.90(0.19)	0.80(0.25)	0.83(0.27)	0.88(0.42)
	0.98(0.82)	0.93(0.60)	0.92(0.57)	0.95(0.63)	0.93(0.69)	0.92(0.52)	0.91(0.57)	0.91(0.55)	0.93(0.62)
	0.96(0.62)	0.82(0.44)	0.81(0.37)	0.92(0.47)	0.93(0.59)	0.88(0.26)	0.79(0.33)	0.82(0.37)	0.87(0.43)
	0.98(0.81)	0.92(0.58)	0.91(0.50)	0.94(0.57)	0.92(0.67)	0.91(0.50)	0.90(0.58)	0.91(0.59)	0.92(0.60)

In summary, we found that string-based classifiers performed better than integer-based classifiers for binary classification. Whereas, tree-based classifiers (RF and J48) performed better on overall than other classifiers (SVM, NB and kNN). NBMT performed better with 100 patterns. Moreover, no significant difference is found in results obtained with a particular SPM algorithm. All the classifiers performed similarly, with negligible difference, on patterns discovered with TKS, CM-SPAM, VMSP and CloFast. For multi-class classification, NBMT's performance was less than most of the integer-based classifiers. No notable performance difference was found in NBMT with 100 and 200 patterns. On the other hand, RF performed better than other classifiers (J48, SVM, NB and kNN). Some integer-based classifiers' performance was better on 100 patterns than on 200 patterns. Moreover, VMSP (CloFast) results were better than TKS (CM-SPAM) for all classifiers.

Overall, results show that MalSPM using frequent API calls for classification and detection of malware, can be used in place of providing the whole API call sequences. From Table 1 we can see that malware types contain thousands of API calls on average in each sequence. However, in MalSPM, the patterns discovered with TKS, CM-SPAM and CloFast contain 10 API calls at most. Whereas the VMSP patterns contained 15 API calls at most. Moreover, frequent

API call sequences contained 12-14 distinct API calls. For example, the frequent patterns for Adware contained 12 distinct API calls. Whereas the original API call sequences for Adware contain 212 distinct API calls (Table 1). This shows that approximately 5% (12) of all API calls (212) are frequent and are mostly present in most of the sequences of Adware. The same is the case for other malware types. The first 1000 sequences contain 254 distinct API calls in total and on average, each sequence contains approximately 11,502 API calls. Whereas 7,107 sequences contain 278 distinct API calls in total and on average, each sequence contains approximately 19,647 API calls. In this work, the sequence length for patterns discovered with TKS, CM-SPAM and CloFast only contain 10 frequent API calls in total, and VMSP patterns contain 15 frequent API calls, mostly.

Comparison with prior work. This section compares MalSPM with some of the recent (published in last two years) malware detection and classification methods.

The comparison for binary classification and multi-class classification is provided in Tables 11 and 12 respectively. For binary classification (Table 11), we can see that RF (highlighted in bold) results of MalSPM are much better than those of DT [10] and RF [49] for all five evaluation measures. RF also performed better than Cyberbert [39]. On the other hand, RF performed better than LSTM [10] for all parameters, except ACC. Note that we select RF results from this work for comparison as it performed better than other classifiers on both binary and multi-class classification. It is important to point out here that none of the prior studies used the MCC metric to evaluate the classification results. We used the MCC metric to confirm that the datasets used in this work for binary classification are imbalanced (see Definition 1). The low MCC scores for RF on six malware types in Table 11 indicate that indeed the 0 (other) class is a majority class compared to the 1 (malware type) class.

For multi-class classification (Table 12), we can see that both RF and kNN (highlighted in bold) results of MalSPM outperformed other classifiers. The high MCC scores for RF and kNN in Table 12 confirm that all the classes contain the same number of sequences in the datasets. For the computation time, we found that all the seven classifiers in MalSPM run fast and they were able to produce results in seconds.

The association rule-based malware classification approach [14] performance was also evaluated on the dataset [9]. It achieved 99.03% ACC, 96.30% R, 95.93% P and 96.10% F1, on average, across 8 malware classes, and performed 1.75% and 2.32% better than Markov chain-based and API call alignment-based malware detectors, respectively. Their and our results indicate that using frequent API calls or association rules between them indeed provide better results than previous state-of-the-art approaches for malware detection and classification.

In a prior study, Xu et al. [55] combined the 7,107 malware sequences with 100 benign samples provided by Amer and Zelinka [5]. Thus the total dataset contained 7,207 malware and benign samples (sequences). The classification is

Table 11: Binary classification comparison

	Adware	Backdoor	Downloader	Dropper	Virus	Worms	Trojan	Spyware
LSTM [10]								
ACC	0.985	0.877	0.909	0.877	0.927	0.865	0.835	0.876
P	0.90	0.60	0.72	0.52	0.76	0.57	0.36	0.46
R	0.77	0.40	0.57	0.44	0.70	0.37	0.22	0.42
F1	0.83	0.48	0.64	0.48	0.73	0.45	0.27	0.44
MCC	–	–	–	–	–	–	–	–
DT [10]								
ACC	0.954	0.824	0.839	0.815	0.856	0.825	0.786	0.824
P	0.53	0.39	0.43	0.32	0.48	0.41	0.26	0.30
R	0.80	0.41	0.48	0.38	0.52	0.40	0.28	0.41
F1	0.64	0.40	0.45	0.35	0.50	0.40	0.27	0.35
MCC	–	–	–	–	–	–	–	–
Cyberbert [39]								
ACC	–	–	–	–	–	–	–	–
P	–	–	–	–	–	–	–	–
R	–	–	–	–	–	–	–	–
F1	0.68	0.46	0.58	0.39	0.38	0.38	0.60	0.45
MCC	–	–	–	–	–	–	–	–
DT [49]								
ACC	0.68	0.44	0.58	0.43	0.59	0.41	0.27	0.27
P	–	–	–	–	–	–	–	–
R	–	–	–	–	–	–	–	–
F1	–	–	–	–	–	–	–	–
MCC	–	–	–	–	–	–	–	–
RF [49]								
ACC	0.55	0.22	0.49	0.03	0.25	0.13	0.006	0.03
P	–	–	–	–	–	–	–	–
R	–	–	–	–	–	–	–	–
F1	–	–	–	–	–	–	–	–
MCC	–	–	–	–	–	–	–	–
MalSPM(RF)								
ACC	0.987	0.837	0.80	0.81	0.925	0.85	0.887	0.865
P	0.98	0.77	0.69	0.73	0.92	0.74	0.84	0.79
R	0.98	0.83	0.80	0.81	0.92	0.85	0.88	0.86
F1	0.98	0.80	0.74	0.76	0.92	0.79	0.86	0.81
MCC	0.92	0.007	-0.08	-0.03	0.50	-0.04	-0.05	0.03

done for either samples belonging to the malware or benign class. No binary classification for 8 malware types was done nor multi-class classification. This is the reason the results of Xu et al. [55] are not included in the comparison tables. However, they provided the distribution of the top 10 API calls in the training and testing datasets. The obtained top 10 API calls were very similar to the ones we found earlier through various SPM algorithms. As mentioned at the start of this section, only those prior studies are compared that are published in the last two years and applied for metamorphic malware. That is why the results of some other studies [46, 48, 59] are not included in the comparison tables.

Table 12: Multi-Class classification comparison

	P	R	F1	MCC
LSTM [10]	0.50	0.47	0.47	–
2LSTM [10]	0.40	0.41	0.39	–
DT [10]	0.40	0.41	0.40	–
kNN [10]	0.35	0.35	0.34	–
RF [10]	0.46	0.47	0.46	–
SVM [10]	0.78	0.29	0.29	–
LSTM [36]	0.55	0.55	0.55	–
GRU [36]	0.55	0.55	0.54	–
Cyberbert [39]	–	–	0.50	–
RF [49]	–	–	0.898	–
kNN [49]	–	–	0.58	–
MalSPM(RF)	0.93	0.93	0.93	0.92
MalSPM(kNN)	0.93	0.93	0.93	0.92

6 Conclusion

In this study, sequential pattern mining (SPM) was used for the task of metamorphic malware analysis and classification, which is an important research topic in cybersecurity. More specifically, we proposed the MalSPM system, where SPM algorithms are used to extract patterns from various malware behavior. A dataset that contains API call sequences of various metamorphic malware was first converted into a suitable format. Various SPM algorithms were then used on the abstracted dataset to find frequent API calls, frequent API call sequences, the sequential relationships between frequent patterns as well as closed and maximal frequent API call patterns. All the algorithms performed efficiently on the dataset and some interesting patterns were found with SPM. Second, MalSPM utilizes the frequent API call patterns of various malware for classification. Five integer-based and two text-based classifiers were used. Obtained results after extensive experiments suggest that tree-based classifiers (RF and J48) performed better than others and results indicated that limited (or short) API call sequences that only contain frequent API calls can be used to reliably predict and classify whether a whole sequence is malicious or not.

There are several research directions for future work, some of which are:

- Using emerging pattern mining or contrast set mining techniques [47] on the API calls of various malware to discover emerging (or contrasting) trends that show a clear and useful difference (or contrast) between various malware behavior.
- Predicting the next API calls in a sequence using prediction models.
- Investigating the classification approach on other datasets that contain API call sequences of malware.
- Using metrics such as the mean time to detect (MTTD) to compare MalSPM with existing malware detection approaches.

Acknowledgement

This work was partially supported by the Shenzhen Science and Technology Program (Basic Research Project, grant no. JCYJ20210324133003011)

References

1. C. C. Aggarwal. *Data Classification Algorithms and Applications*. 1st Edition, CRC Press, 2015.
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of VLDB*, pages 487–499, 1994.
3. M. Ahmadi, A. Sami, H. Rahimi, and B. Yadegari. Malware detection by behavioural sequential patterns. *Computer Fraud and Security*, 2013(8):11–19, 2013.
4. F. Al Shamsi, W. L. Woon, and Z. Aung. Discovering similarities in malware behaviors by clustering of API call sequences. In *Proceedings of ICONIP 2018*, pages 122–133, 2018.
5. E. Amer and I. Zelinka. A dynamic windows malware detection and prediction method based on contextual understanding of API call sequence. *Computers & Security*, 92:101760, 2020.
6. J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the SIGKDD, 2002*, pages 429–435, 2002.
7. M. Botacin, F. Ceschin, R. Sun, D. Oliveira, and A. Grégio. Challenges and pitfalls in malware research. *Computers & Security*, 106:102287, 2021.
8. M. Campion, M. D. Preda, and R. Giacobazzi. Learning metamorphic malware signatures from samples. *Journal of Computer Virology and Hacking Techniques*, 17:167–183, 2021.
9. F. Ö. Çatak and A. F. Yazi. A benchmark API call dataset for windows PE malware classification. *CoRR*, abs/1905.01999, 2019.
10. F. Ö. Çatak, A. F. Yazi, O. Elezaj, and J. Ahmed. Deep learning based sequential model for malware analysis using Windows exe API calls. *PeerJ Computer Science*, 6:e285, 2020.
11. I. K. Cho and E. G. Im. Extracting representative API patterns of malware families using multiple sequence alignments. In *Proceedings of RACS*, pages 308–313, 2015.
12. I. K. Cho, T. Kim, Y. J. Shim, H. Park, B. Choi, and E. G. Im. Malware similarity analysis using api sequence alignments. *Journal of Internet Services and Information Security*, 4:103–114, 2014.
13. I. K. Cho, T. Kim, Y. J. Shim, M. Ryu, and E. G. Im. Malware analysis and classification using sequence alignments. *Intelligent Automation and Soft Computing*, 22(3):371–377, 2016.
14. G. D'Angelo, M. Ficco, and F. Palmieri. Association rule-based malware classification using common subsequences of API calls. *Applied Soft Computing*, 105:107234, 2021.
15. Y. Fan, Y. Ye, and L. Chen. Malicious sequential pattern mining for automatic malware detection. *Expert Systems with Applications*, 52:16–25, 2016.
16. M. Ficco. Malware analysis by combining multiple detectors and observation windows. *IEEE Transactions on Computers*, Early Access, 2021.
17. P. Fournier-Viger, A. Gomariz, M. Campos, and R. Thomas. Fast vertical mining of sequential patterns using co-occurrence information. In *Proceedings of PAKDD*, pages 40–52, 2014.

18. P. Fournier-Viger, A. Gomariz, T. Gueniche, E. Mwamikazi, and R. Thomas. TKS: Efficient mining of top-k sequential patterns. In *Proceedings of ADMA 2013*, pages 109–120, 2013.
19. P. Fournier-Viger, T. Gueniche, S. Zida, and V. S. Tseng. ERMiner: Sequential rule mining using equivalence classes. In *Proceedings of IDA 2014*, pages 108–119, 2014.
20. P. Fournier-Viger, J. C. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam. The SPMF open-source data mining library version 2. In *Proceedings of the ECML/PKDD 2016*, pages 36–40, 2016.
21. P. Fournier-Viger, J. C. W. Lin, R. U. Kiran, Y. S. Koh, and R. Thomas. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1(1):54–77, 2017.
22. P. Fournier-Viger, C. Wu, A. Gomariz, and V. S. Tseng. VMSP: Efficient vertical mining of maximal sequential patterns. In *Proceedings of CCAI 2014*, pages 83–94, 2014.
23. E. Frank, M. A. Hall, and I. H. Witten. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques. Fourth Edition.* Morgan Kaufmann, 2016.
24. F. Fumarola, P. F. Lanotte, M. Ceci, and D. Malerba. CloFAST: Closed sequential pattern mining using sparse and vertical id-lists. *Knowledge and Information Systems*, 48(2):429–463, 2016.
25. H. S. Galal, Y. B. Mahdy, and M. A. Atiea. Behavior based feature model for malware malware detection. *Journal of Computer Virology and Hacking Techniques*, 12:59–67, 2016.
26. D. Gibert, C. Mateu, J. Planes, and J. Marques-Silva. Auditing static machine learning anti-malware tools against metamorphic attacks. *Computers & Security*, 102:102159, 2021.
27. K. Griffin, S. Schneider, X. Hu, and T.-c. Chiueh. Automatic generation of string signatures for malware detection. In *Proceesings of RAID 2009*, pages 101–120, 2009.
28. J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8:53–57, 2004.
29. S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
30. Y. Jian, H. Kuang, C. Ren, Z. Ma, and H. Wang. A novel framework for image-based malware detection with a deep neural network. *Computers & Security*, 109:102400, 2021.
31. K. N. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev. Ensemblehmd: Accurate hardware malware detectors with specialized ensemble classifiers. *IEEE Transactions on Dependable and Secure Computing*, 17(3):620–633, 2020.
32. Y. Ki, E. Kim, and H. K. Kim. A novel approach to detect malware based on API call sequence analysis. *International Journal of Distributed Sensor Networks*, 11:659101:1–659101:9, 2015.
33. H. Kim, W. Khoo, and P. Li. Polymorphic attacks against sequencebased software birthmarks. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Software Security and Protection*, pages 1–8, 2012.
34. H. Kim, J. Kim, Y. Kim, I. Kim, K. J. Kim, and H. Kim. Improvement of malware detection and classification using API call sequence alignment and visualization. *Cluster Computing*, 22:921–929, 2019.

35. T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im. A multimodal deep learning method for android malware detection using various features. *IEEE Transactions on Information Forensics and Security*, 14(3):773–788, 2019.
36. C. Li and J. Zheng. API call-based malware classification using recurrent neural networks. *Journal of Cyber Security and Mobility*, 10:617–640, 2021.
37. Y. Liu and Y. Wang. A robust malware detection system using deep learning on API calls. In *Proceedings of ITNEC 2019*, pages 1456–1460, 2019.
38. J. M. Luna, P. Fournier-Viger, and S. Ventura. Frequent itemset mining: A 25 years review. *WIREs Data Mining and Knowledge Discovery*, 9(6):e1329, 2019.
39. S. McDonnell, O. Nada, M. R. Abid, and E. Amjadian. Cyberbert: A deep dynamic-state session-based recommender system for cyber threat recognition. In *Proceedings of Aerospace Conference*, pages 160–165, 2021.
40. F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 20:343–357, 2016.
41. M. S. Nawaz, P. Fournier-Viger, M. Z. Nawaz, G. Chen, and Y. Wu. Metamorphic malware behavior analysis using sequential pattern mining. In *Proceedings of ECML PKDD Workshops*, pages 90–103, 2021.
42. M. S. Nawaz, M. Sun, and P. Fournier-Viger. Proof guidance in PVS with sequential pattern mining. In *Proceedings of FSEN*, pages 45–60, 2019.
43. O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys*, 52(5), 2019.
44. A. Pektas, E. N. Pektas, and T. Acarman. Mining patterns of sequential malicious APIs to detect malware. *International Journal of Network Security & Its Applications*, 10(4):1–9, 2018.
45. Y. Qiao, Y. Yang, J. He, C. Tang, and Z. Liu. CBM: Free, automatic malware analysis framework using API call sequences. In *Knowledge Engineering and Management*, pages 225–236, 2014.
46. Y. Qiao, Y. Yang, L. Ji, and J. He. Analyzing malware by abstracting the frequent itemsets in API call sequences. In *Proceedings of TrustCom 2013*, pages 265–270, 2013.
47. V. S and J. M. Luna. *Supervised Descriptive Pattern Mining*. Springer, 2018.
48. A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamzeh. Malware detection based on mining API calls. In *Proceedings of SAC 2010*, pages 1020–1025, 2010.
49. M. Schofield. Comparison of malware classification methods using convolutional neural network based on API call stream. *International Journal of Network Security & Its Applications*, 13:1–19, 2021.
50. A. Tekerek and M. M. Yapici. A novel malware classification and augmentation model based on convolutional neural network. *Computers & Security*, 112:102515, 2021.
51. D. Ucci. Survey of machine learning techniques for malware analysis. *Computers & Security*, 81:123–147, 2019.
52. J. Umeh. From Pc to HC? *ITNOW*, 63(2):10–11, 2021.
53. R. J. Urbanowicz and W. N. Browne. *Introduction to Learning Classifier Systems*. 1st Edition, Springer, 2017.
54. Z. Wang, Q. Liu, and Y. Chi. Review of android malware detection based on deep learning. *IEEE Access*, 8:181102–181126, 2020.
55. Z. Xu, X. Fang, and G. Yang. Malbert: A novel pre-training method for malware classification. *Computers & Security*, 111, 2021.

56. X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets. In *Proceedings of SDM 2003*, pages 166–177, 2003.
57. Y. Ye, T. Li, D. A. Adjeroh, and S. S. Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys*, 50(3):41:1–41:40, 2017.
58. Y. Ye, T. Li, Q. Jiang, and Y. Wang. CIMDS: Adapting postprocessing techniques of associative classification for malware detection. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 40(3):298–307, 2010.
59. Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang. An intelligent PE-malware detection system based on association mining. *Journal of Computer Virology*, 4(4):323–334, 2008.
60. C. Zhang and S. Zhang. *Association Rule Mining Models and Algorithms*. Springer, 2002.

Appendix

Table A1: Binary classification results for patterns extracted using TKS (CM-SPAM)

Cl	Adware	Backdoor	Downloader	Dropper	Virus	Worms	Trojan	Spyware
SVM								
ACC	88.7(90.6)	86.2(86.25)	84(83.75)	85(85)	91.2(91.2)	88(87)	91.8(92.5)	87.5(87)
P	87.8(87.7)	88.2(88.2)	87.5(87.5)	87.5(87.5)	90(90)	86.8(86.5)	88(88)	86.8(86.8)
R	0.80(0.91)	0.75(0.75)	0.72(?)	?(?)	0.89(?)	?(0.80)	?(0.93)	?(0.76)
F1	?(?)	?(?)	?(?)	?(?)	?(?)	?(?)	?(?)	?(?)
MCC	0.88(0.90)	0.86(0.86)	0.84(0.83)	0.85(0.85)	0.91(0.91)	0.88(0.87)	0.91(0.92)	0.87(0.87)
RF	0.87(0.87)	0.87(0.88)	0.87(0.87)	0.87(0.87	0.90(0.90)	0.86(0.86)	0.88(0.88)	0.86(0.86)
ACC	0.84(0.86)	0.80(0.80)	0.77(?)	?(?)	0.90(?)	?(0.82)	?(0.89)	?(0.81)
P	?(?)	?(?)	?(?)	?(?)	?(?)	?(?)	?(?)	?(?)
R	−0.03(0.23)	−0.03(−0.03)	−0.04(?)	?(?)	0.33(?)	?(0.05)	?(0.26)	?(−0.02)
F1	?(?)	?(?)	?(?)	?(?)	?(?)	?(?)	?(?)	?(?)
MCC	98.75(92.5)	83.7(86.2)	80(86.2)	81(82.5)	92.5(92.5)	85(84)	88.7(90)	86.5(90)
kNN	89(90.6)	86.2(88.7)	82(85.2)	83.1(85.2)	90(92.1)	81.2(85.3)	83.7(85.2)	81.5(83.7)
ACC	0.98(0.91)	0.77(0.83)	0.69(0.84)	0.73(0.74)	0.92(0.88)	0.74(0.78)	0.84(0.86)	0.79(0.91)
P	0.88(0.89)	0.83(0.86)	0.76(0.80)	0.79(0.80)	0.89(0.91)	0.74(0.75)	0.79(0.79)	0.75(0.76)
R	0.98(0.92)	0.83(0.86)	0.80(0.86)	0.81(0.82)	0.92(0.91)	0.85(0.84)	0.88(0.90)	0.86(0.90)
F1	0.89(0.90)	0.86(0.88)	0.82(0.85)	0.83(0.85)	0.90(0.92)	0.81(0.85)	0.83(0.85)	0.81(0.83)
MCC	0.98(0.92)	0.80(0.84)	0.74(0.83)	0.76(0.77)	0.92(0.88)	0.79(0.81)	0.86(0.87)	0.81(0.87)
kNN	0.88(0.89)	0.84(0.86)	0.79(0.82)	0.80(0.82)	0.89(0.91)	0.77(0.80)	0.81(0.82)	0.78(0.79)
ACC	0.92(0.53)	0.007(0.26)	−0.08(0.36)	−0.03(−0.004)	0.50(0.16)	−0.04(−0.014)	−0.05(0.07)	0.03(0.46)
P	0.43(0.47)	0.20(0.30)	−0.03(0.09)	0.01(0.09)	0.42(0.52)	−0.09(−0.04)	0.008(0.03)	−0.05(−0.02)
R	91.5(86.2)	80.6(80.5)	78(75)	78(76)	87.5(88.1)	79(77.5)	81.2(82)	81.5(84.5)
F1	85.9(84.3)	80.9(83.7)	73.25(73.1)	79(79)	84(88.7)	75.6(79.3)	75(75.7)	73.2(76.7)
MCC	0.91(0.85)	0.79(0.77)	0.75(0.72)	0.74(0.71)	0.89(0.84)	0.76(0.77)	0.83(0.82)	0.75(0.84)
NB	0.85(0.82)	0.82(0.83)	0.78(0.75)	0.79(0.77)	0.87(0.87)	0.76(0.74)	0.76(0.77)	0.74(0.73)
ACC	0.91(0.86)	0.80(0.80)	0.78(0.75)	0.78(0.76)	0.87(0.88)	0.79(0.77)	0.81(0.82)	0.81(0.84)
P	0.85(0.84)	0.80(0.83)	0.73(0.73)	0.79(0.79)	0.84(0.88)	0.75(0.79)	0.75(0.75)	0.73(0.76)
R	0.91(0.85)	0.80(0.78)	0.76(0.73)	0.76(0.73)	0.88(0.86)	0.77(0.77)	0.82(0.82)	0.78(0.84)
F1	0.85(0.83)	0.81(0.83)	0.75(0.74)	0.79(0.78)	0.85(0.88)	0.75(0.76)	0.75(0.76)	0.73(0.75)
MCC	0.59(0.19)	0.09(−0.01)	0.03(−0.01)	0(−0.13)	0.31(0.04)	−0.11(−0.08)	−0.10(0)	−0.09(0.27)
J48DT	0.30(0.18)	0.26(0.20)	0.03(−0.13)	0.01(−0.03)	0.29(0.30)	−0.04(−0.1)	−0.01(−0.06)	−0.01(−0.05)
ACC	76(71.87)	63.5(82.5)	78.5(83.5)	82(79)	74.5(74.3)	67.5(77.5)	56.2(89.3)	88(67.5)
P	83.1(86.5)	86(87.7)	80(86.5)	87.1(88.1)	88.1(90.5)	80.3(86.5)	87.5(87)	86.5(84.2)
R	0.89(0.87)	0.82(0.82)	0.79(0.72)	0.81(0.89)	0.87(0.92)	0.83(0.80)	0.88(0.87)	0.89(0.87)
F1	0.87(0.79)	0.87(0.84)	0.80(0.80)	0.80(0.89)	0.86(0.88)	0.81(0.75)	0.77(0.82)	0.75(0.72)
MCC	0.76(0.71)	0.635(0.825)	0.78(0.835)	0.82(0.74)	0.74(0.91)	0.67(0.77)	0.56(0.89)	0.88(0.67)
J48DT	0.83(0.86)	0.86(0.87)	0.80(0.86)	0.87(0.88)	0.88(0.90)	0.80(0.86)	0.87(0.87)	0.86(0.84)
ACC	0.80(0.77)	0.69(0.82)	0.79(0.77)	0.81(0.79)	0.78(0.88)	0.72(0.79)	0.65(0.88)	0.82(0.728)
P	0.84(0.82)	0.86(0.85)	0.80(0.82)	0.82(0.82)	0.87(0.88)	0.80(0.80)	0.82(0.84)	0.80(0.78)
R	0.39(0.23)	0.17(0.21)	0.21(−0.05)	0.25(0.24)	0.29(0.25)	0.16(0.08)	0.13(0.14)	0.18(0.34)
F1	0.37(0.01)	0.39(0.20)	0.12(0.07)	0.04(0.15)	0.26(0.29)	0.17(−0.02)	−0.02(0.16)	−0.02(−0.04)
MCC	97.5(94.37)	86.5(85.62)	81.2(85.5)	81.5(83)	93.1(91.8)	87.5(87.5)	88.1(91.8)	87(90)
RF	90.6(90.2)	84.7(88.5)	85.6(86.7)	87.5(81.5)	90.9(92.1)	85.5(86.8)	85.5(86.2)	83.4(87.5)
ACC	0.97(0.94)	0.80(0.81)	0.746(0.829)	0.71(0.72)	0.92(0.92)	0.77(0.84)	0.84(0.89)	0.82(0.91)
P	0.89(0.89)	0.84(0.85)	0.78(0.80)	0.83(0.83)	0.89(0.91)	0.78(0.86)	0.79(0.81)	0.77(0.89)
R	0.97(0.94)	0.86(0.85)	0.813(0.855)	0.81(0.83)	0.93(0.91)	0.87(0.87)	0.88(0.91)	0.87(0.90)
F1	0.90(0.90)	0.84(0.88)	0.85(0.86)	0.86(0.87)	0.90(0.92)	0.85(0.86)	0.85(0.86)	0.83(0.87)
MCC	0.97(0.93)	0.81(0.82)	0.76(0.80)	0.76(0.77)	0.92(0.88)	0.82(0.84)	0.86(0.89)	0.83(0.86)
RF	0.89(0.88)	0.84(0.86)	0.81(0.81)	0.84(0.83)	0.89(0.91)	0.80(0.82)	0.81(0.82)	0.79(0.82)
ACC	0.85(0.64)	0.08(0.16)	0.05(0.17)	−0.08(−0.06)	0.47(0.25)	−0.02(0.21)	−0.05(0.24)	0.13(0.42)
P	0.49(0.43)	0.23(0.28)	0(0.05)	0.22(0.14)	0.40(0.48)	0.04(0.01)	0.01(0.14)	0.001(0.20)

Table A2: Binary classification results for patterns extracted using VMSP (CloFast)

Cl	Adware	Backdoor	Downloader	Dropper	Virus	Worms	Trojan	Spyware
SVM								
ACC	88(90)	87(87)	85(85)	85(85)	91.2(89.3)	88(88)	89(91)	87.5(87.5)
P	90(87.5)	85.9(87.7)	87.5(87.5)	87.8(87.8)	90(90)	86.8(86.2)	88(88.2)	86.8(86.8)
R	0.78(?)	?(?)	?(?)	?(?)	?(0.85)	0.83(0.74)	0.80(?)	?(?)
F1	0.89(0.83)	0.74(0.77)	?(?)	?(?)	?(?)	?(0.74)	?(0.89)	?(?)
MCC	0.88(0.90)	0.87(0.87)	0.85(0.85)	0.85(0.85)	0.91(0.89)	0.88(0.88)	0.89(0.91)	0.87(0.87)
RF	0.90(0.87)	0.85(0.87)	0.87(0.87)	0.87(0.87)	0.90(0.90)	0.86(0.86)	0.88(0.88)	0.86(0.86)
ACC	0.82(?)	?(?)	?(?)	?(?)	?(0.87)	0.83(0.83)	0.84(?)	?(?)
P	0.87(0.83)	0.79(0.82)	?(?)	?(?)	?(?)	?(0.80)	?(0.83)	?(?)
R	−0.02(?)	?(?)	?(?)	?(?)	?(0.07)	0.11(0.11)	−0.03(?)	?(?)
F1	0.402(0.16)	−0.02(−0.02)	?(?)	?(?)	?(?)	?(−0.02)	?(0.13)	?(?)
MCC								
RF								
ACC	92.5(90.6)	84.5(84.5)	83(83.5)	88.7(82)	88(88.7)	85(86.8)	85.6(86.8)	84(86.5)
P	92(90)	86.7(84.5)	84.6(82)	88.1(84.3)	89(87.8)	87.1(81.2)	85.7(83)	85.6(83.4)
R	0.92(0.89)	0.77(0.79)	0.76(0.75)	0.72(0.77)	0.86(0.87)	0.82(0.82)	0.82(0.85)	0.79(0.81)
F1	0.91(0.88)	0.83(0.82)	0.81(0.76)	0.85(0.77)	0.86(0.88)	0.84(0.75)	0.80(0.77)	0.81(0.79)
MCC	0.92(0.90)	0.84(0.84)	0.83(0.83)	0.83(0.82)	0.88(0.88)	0.85(0.87)	0.85(0.86)	0.84(0.86)
kNN	0.92(0.90)	0.86(0.84)	0.84(0.82)	0.88(0.84)	0.89(0.87)	0.87(0.81)	0.85(0.83)	0.85(0.83)
ACC	0.92(0.90)	0.80(0.81)	0.78(0.78)	0.77(0.79)	0.86(0.87)	0.83(0.83)	0.83(0.86)	0.81(0.82)
P	0.90(0.88)	0.84(0.83)	0.82(0.79)	0.86(0.80)	0.87(0.87)	0.85(0.78)	0.82(0.80)	0.83(0.81)
R	0.56(0.43)	0.007(0.06)	0.05(0.02)	−0.05(0.10)	0.27(0.19)	0.17(0.11)	−0.03(0.01)	0.41(0.11)
F1	0.56(0.41)	0.18(0.14)	0.15(−0.03)	0.31(−0.02)	0.25(0.33)	0.29(−0.06)	0.07(−0.05)	0.18(0.10)
MCC								
kNN								
ACC	88.1(86.2)	76.2(79)	76.5(79)	79.5(76.5)	83.7(86.8)	78.5(77.5)	78.1(81.2)	76.5(80)
P	87.1(87.8)	83.7(85.7)	80(76.5)	80.3(78.4)	82.1(79)	83.5(76)	79.3(76.5)	79.5(83.1)
R	0.89(0.85)	0.78(0.75)	0.74(0.73)	0.75(0.74)	0.88(0.85)	0.81(0.77)	0.84(0.85)	0.79(0.77)
F1	0.86(0.86)	0.83(0.83)	0.81(0.77)	0.82(0.78)	0.85(0.83)	0.82(0.76)	0.77(0.78)	0.75(0.80)
MCC	0.88(0.86)	0.76(0.79)	0.76(0.79)	0.79(0.76)	0.83(0.86)	0.78(0.81)	0.78(0.81)	0.76(0.80)
NB	0.87(0.87)	0.83(0.85)	0.80(0.76)	0.80(0.78)	0.82(0.79)	0.83(0.76)	0.79(0.76)	0.79(0.81)
ACC	0.88(0.85)	0.77(0.83)	0.75(0.76)	0.77(0.78)	0.85(0.86)	0.79(0.79)	0.81(0.83)	0.77(0.78)
P	0.86(0.86)	0.83(0.84)	0.80(0.76)	0.81(0.78)	0.83(0.81)	0.82(0.76)	0.78(0.77)	0.77(0.81)
R	0.42(0.19)	0.03(−0.07)	0.03(−0.02)	0.24(−0.02)	0.12(0.09)	−0.06(−0.04)	0.04(−0.02)	−0.09(−0.019)
F1	0.36(0.34)	0.22(0.19)	0.14(−0.06)	0.16(−0.01)	0.16(0.10)	0.24(0.02)	0.01(−0.03)	0.02(0.12)
MCC								
NB								
ACC	90(81.2)	85.6(74.5)	81(85)	68.5(83)	60.5(86.8)	80(76)	88.7(66)	55.5(85.5)
P	78.7(86.5)	64.5(77.5)	80.7(87.1)	84(87.1)	60.2(87.7)	86.8(86.8)	60.9(86.5)	52.7(82.7)
R	0.88(0.82)	0.75(0.85)	0.74(?)	0.79(0.72)	0.87(0.88)	0.84(0.84)	0.89(0.86)	0.79(0.80)
F1	0.81(0.84)	0.86(0.81)	0.79(0.76)	0.84(0.77)	0.85(0.84)	0.86(0.82)	0.85(0.82)	0.84(0.75)
MCC	0.90(0.81)	0.85(0.74)	0.81(0.85)	0.68(0.83)	0.60(0.86)	0.80(0.76)	0.88(0.66)	0.55(0.85)
J48DT	0.78(0.86)	0.64(0.77)	0.80(0.87)	0.84(0.87)	0.60(0.87)	0.86(0.86)	0.60(0.86)	0.52(0.82)
ACC	0.89(0.81)	0.80(0.78)	0.77(?)	0.72(0.77)	0.67(0.87)	0.81(0.79)	0.89(0.72)	0.63(0.82)
P	0.80(0.85)	0.86(0.79)	0.80(0.81)	0.84(0.81)	0.68(0.86)	0.86(0.81)	0.67(0.84)	0.58(0.78)
R	0.38(0.01)	−0.04(0.30)	0.01(?)	0.16(−0.06)	0.24(0.29)	0.32(0.24)	0.29(0.19)	0.05(0.07)
F1	0.14(0.29)	0.24(0.22)	0.08(−0.02)	0.26(−0.03)	0.14(0.17)	0.38(0.12)	0.26(0.16)	0.24(0.02)
MCC								
J48DT								
ACC	91.2(91.2)	83.7(86.2)	82(83.1)	81.8(82.5)	89.3(90.9)	88(87.5)	88.7(88)	84.3(88)
P	91(91.2)	88(87.2)	87.5(85.5)	89(86.2)	90.9(91.2)	88.2(83.5)	83(85.6)	87.5(83.4)
R	0.90(0.89)	0.77(0.81)	0.75(0.75)	0.77(0.75)	0.86(0.89)	0.84(0.82)	0.88(0.80)	0.75(0.85)
F1	0.90(0.90)	0.85(0.83)	0.84(0.77)	0.87(0.76)	0.89(0.89)	0.89(0.77)	0.80(0.79)	0.85(0.81)
MCC	0.91(0.91)	0.83(0.86)	0.82(0.83)	0.81(0.82)	0.89(0.91)	0.88(0.87)	0.88(0.88)	0.85(0.88)
RF	0.91(0.91)	0.88(0.87)	0.87(0.85)	0.89(0.86)	0.90(0.91)	0.88(0.83)	0.83(0.85)	0.87(0.83)
ACC	0.91(0.89)	0.80(0.82)	0.78(0.77)	0.78(0.78)	0.87(0.89)	0.84(0.83)	0.88(0.84)	0.79(0.83)
P	0.89(0.90)	0.86(0.84)	0.84(0.80)	0.86(0.81)	0.89(0.90)	0.84(0.80)	0.81(0.81)	0.83(0.82)
R	0.48(0.40)	0.007(0.14)	0.03(0.06)	0.18(0.04)	0.15(0.28)	0.20(0.13)	0.24(−0.04)	−0.06(0.20)
F1	0.49(0.52)	0.30(0.18)	0.24(−0.02)	0.35(−0.04)	0.37(0.41)	0.33(0.04)	0.06(0.09)	0.21(0.18)
MCC								

Table A3: String-based binary classification results for patterns extracted using TKS(CM-SPAM)

Cl	Adware	Backdoor	Downloader	Dropper	Virus	Worms	Trojan	Spyware
NBMT								
ACC	99.3(90.6)	90(75)	86(88)	84(85.5)	89.3(78.1)	85.5(72)	91.2(76.5)	88(76.5)
	90.6(88.5)	86.5(80.6)	63.2(64.6)	88.1(64.6)	90.5(90)	63(72)	70.5(63.7)	82.1(83.1)
P	0.99(0.91)	0.89(0.88)	0.83(0.87)	0.79(0.87)	0.89(0.92)	0.81(0.85)	0.84(0.83)	0.89(0.76)
	0.89(0.86)	0.84(0.86)	0.84(0.81)	0.89(0.79)	0.89(0.86)	0.81(0.81)	0.83(0.83)	0.79(0.80)
R	0.99(0.90)	0.90(0.75)	0.86(0.64)	0.84(0.85)	0.89(0.78)	0.85(0.72)	0.91(0.76)	0.88(0.76)
	0.90(0.88)	0.86(0.80)	0.63(0.64)	0.88(0.64)	0.90(0.90)	0.63(0.72)	0.70(0.63)	0.82(0.83)
F1	0.99(0.86)	0.88(0.78)	0.82(0.85)	0.80(0.79)	0.89(0.82)	0.83(0.76)	0.87(0.79)	0.82(0.76)
	0.89(0.86)	0.81(0.82)	0.69(0.70)	0.82(0.70)	0.87(0.85)	0.68(0.75)	0.75(0.69)	0.80(0.81)
MCC	0.96(0.23)	0.46(0.41)	0.28(0.41)	0.16(0.16)	0.31(0.39)	0.11(0.25)	−0.02(0.88)	0.18(−0.05)
	0.49(0.34)	0.14(0.40)	0.24(0.13)	0.15(0.01)	0.25(0.10)	0.16(0.17)	0.16(0.21)	0.10(0.12)
SGDT								
ACC	99.5(93.1)	88.7(88.1)	84.5(89)	94.3(85.5)	94.3(93.1)	87(86)	90.6(90)	86.5(87.5)
	88.4(91.2)	91.2(90)	80(87.1)	88.1(88.1)	91.2(92.5)	86.8(87.1)	85.6(87.7)	86.5(87.1)
P	0.99(0.92)	0.87(0.85)	0.77(0.90)	0.93(0.87)	0.93(0.93)	0.77(0.83)	0.84(0.86)	0.79(0.89)
	0.86(0.90)	0.90(0.88)	0.78(0.80)	0.89(0.89)	0.92(0.91)	0.82(0.83)	0.77(0.83)	0.82(0.83)
R	0.99(0.93)	0.88(0.88)	0.84(0.89)	0.94(0.85)	0.94(0.93)	0.87(0.86)	0.90(0.90)	0.86(0.87)
	0.88(0.91)	0.91(0.90)	0.80(0.87)	0.88(0.88)	0.91(0.92)	0.86(0.87)	0.85(0.87)	0.86(0.87)
F1	0.99(0.91)	0.87(0.85)	0.78(0.86)	0.93(0.79)	0.93(0.91)	0.81(0.84)	0.87(0.87)	0.82(0.82)
	0.86(0.89)	0.89(0.87)	0.79(0.82)	0.82(0.82)	0.88(0.91)	0.81(0.83)	0.80(0.83)	0.82(0.83)
MCC	0.97(0.54)	0.40(0.33)	0.06(0.48)	0.59(0.16)	0.59(0.44)	−0.03(0.20)	−0.03(0.07)	0.05(0.20)
	0.32(0.50)	0.48(0.38)	0.05(0.06)	0.15(0.15)	0.33(0.51)	0.12(0.21)	0.01(0.12)	0.01(0.21)

Table A4: String-based binary classification results for patterns extracted using VMSP (CloFast)

Cl	Adware	Backdoor	Downloader	Dropper	Virus	Worms	Trojan	Spyware
NBMT								
ACC	90.6(93.1)	78(66.5)	62(87)	66(86)	90(91.8)	63.5(70)	89.2(85)	71.5(78.7)
	90(89.5)	82.8(88.5)	70(66.7)	90(88.5)	90.3(88.7)	73.1(71.8)	77.1(62.7)	82.1(77.5)
P	0.88(0.93)	0.83(0.85	0.80(0.88)	0.82(0.88)	0.83(0.90)	0.86(0.84)	0.81(0.87)	0.79(0.84)
	0.90(0.87)	0.85(0.86)	0.86(0.83)	0.89(0.87)	0.88(0.87)	0.85(0.85)	0.82(0.82)	0.83(0.81)
R	0.90(0.93)	0.78(0.66)	0.62(0.87)	0.66(0.86)	0.90(0.91)	0.63(0.70)	0.89(0.85)	0.71(0.86)
	0.90(0.89)	0.82(0.88)	0.70(0.66)	0.90(0.88)	0.90(0.88)	0.73(0.71)	0.77(0.62)	0.78(0.77)
F1	0.87(0.91)	0.80(0.72)	0.67(0.82)	0.70(0.80)	0.86(0.89)	0.69(0.74)	0.85(0.86)	0.74(0.85)
	0.87(0.87)	0.83(0.83)	0.75(0.72)	0.87(0.84)	0.86(0.88)	0.77(0.76)	0.79(0.69)	0.80(0.79)
MCC	0.26(0.53)	0.26(0.28)	0.18(0.34)	0.25(0.23)	−0.03(0.28)	0.26(0.27)	−0.02(0.17)	0.04(0.30)
	0.40(0.37)	0.38(0.14)	0.31(0.20)	0.39(0.26)	0.18(0.37)	0.29(0.31)	0.23(0.12)	0.24(0.18)
SGDT								
ACC	91.8(90.6)	86.2(85.6)	84(87.5)	81.5(86)	87(91.8)	83.5(87)	86.8(88.1)	79(88)
	90(89)	88(90)	86.5(87.1)	90(88.5)	90.3(89.6)	86.5(87.1)	85.9(87.2)	85.9(86.2)
P	0.90(0.89)	0.80(0.79)	0.72(0.87)	0.76(0.85)	0.85(0.90)	0.79(0.80)	0.81(0.85)	0.77(0.89)
	0.90(0.87)	0.85(0.88)	0.76(0.76)	0.89(0.87)	0.88(0.89)	0.80(0.88)	0.80(0.79)	0.80(0.81)
R	0.91(0.90)	0.86(0.85)	0.84(0.87)	0.81(0.86)	0.87(0.91)	0.83(0.87)	0.85(0.88)	0.79(0.88)
	0.90(0.89)	0.88(0.90)	0.86(0.87)	0.90(0.88)	0.90(0.89)	0.86(0.87)	0.85(0.87)	0.85(0.86)
F1	0.91(0.90)	0.81(0.81)	0.77(0.84)	0.78(0.82)	0.86(0.89)	0.81(0.82)	0.83(0.86)	0.78(0.82)
	0.87(0.86)	0.85(0.88)	0.81(0.81)	0.87(0.84)	0.86(0.89)	0.81(0.81)	0.81(0.82)	0.81(0.82)
MCC	0.48(0.43)	0.08(0.05)	−0.04(0.37)	0.06(0.29)	0.25(0.28)	0.02(0.05)	−0.01(0.03)	−0.03(0.18)
	0.40(0.31)	0.24(0.39)	−0.03(−0.02)	0.39(0.26)	0.18(0.38)	0.05(0.14)	0.10(0.01)	0.10(0.14)

Table A5: Multi-Class classification results for patterns extracted usig TKS (CM-SPAM)

Cl	Adware	Backdoor	Downloader	Dropper	Virus	Worms	Trojan	Spyware	W. Avg.
SVM									
P	0.32(0.30)	0.33(0.28)	0.29(0.23)	0.08(0.20)	0.31(0.39)	0.20(0.18)	0.21(0.40)	0.35(0.25)	0.267(0.28)
R	0.27(0.23)	0.33(0.28)	0.22(0.25)	0.27(0.27)	0.43(0.45)	0.23(0.17)	0.25(0.27)	0.18(0.10)	0.27(0.27)
F1	0.80(0.57)	0.12(0.06)	0.20(0.34)	0.02(0.16)	0.82(0.40)	0.09(0.02)	0.11(0.09)	0.26(0.60)	0.303(0.28)
MCC	0.45(0.31)	0.62(0.70)	0.14(0.06)	0.15(0.01)	0.35(0.54)	0.37(0.17)	0.09(0.34)	0.09(0.14)	0.28(0.28)
RF	0.46(0.39)	0.17(0.09)	0.23(0.28)	0(0.17)	0.45(0.39)	0.12(0.03)	0.14(0.14)	0.30(0.35)	0.24(0.23)
P	0.33(0.27)	0.43(0.40)	0.17(0.09)	0.19(0.02)	0.38(0.49)	0.28(0.17)	0.13(0.30)	0.12(0.07)	0.26(0.23)
R	0.40(0.30)	0.13(0.08)	0.15(0.15)	−0.02(0.07)	0.40(0.30)	0.06(0.02)	0.07(0.14)	0.22(0.25)	0.179(0.16)
F1	0.22(0.15)	0.35(0.32)	0.09(0.06)	0.12(0.03)	0.31(0.41)	0.16(0.05)	0.08(0.19)	0.04(0.56)	0.17(0.16)
MCC									
kNN	0.92(0.62)	0.51(0.39)	0.38(0.35)	0.29(0.44)	0.62(0.64)	0.31(0.38)	0.32(0.36)	0.42(0.38)	0.475(0.45)
P	0.77(0.54)	0.74(0.52)	0.64(0.36)	0.65(0.37)	0.86(0.75)	0.49(0.28)	0.60(0.48)	0.57(0.45)	0.67(0.47)
R	0.93(0.57)	0.50(0.26)	0.32(0.30)	0.32(0.29)	0.60(0.49)	0.31(0.45)	0.33(0.40)	0.48(0.74)	0.47(0.43)
F1	0.89(0.62)	0.68(0.58)	0.55(0.32)	0.70(0.34)	0.78(0.65)	0.50(0.32)	0.64(0.47)	0.59(0.44)	0.66(0.47)
MCC	0.92(0.59)	0.50(0.31)	0.34(0.32)	0.30(0.35)	0.61(0.55)	0.31(0.41)	0.32(0.37)	0.44(0.87)	0.47(0.43)
NB	0.83(0.58)	0.71(0.55)	0.59(0.34)	0.67(0.35)	0.82(0.69)	0.50(0.30)	0.62(0.47)	0.58(0.44)	0.66(0.47)
P	0.91(0.54)	0.43(0.24)	0.26(0.23)	0.20(0.28)	0.55(0.50)	0.21(0.32)	0.23(0.28)	0.36(0.44)	0.39(0.36)
R	0.80(0.51)	0.67(0.48)	0.54(0.25)	0.63(0.26)	0.79(0.66)	0.42(0.19)	0.56(0.40)	0.52(0.37)	0.62(0.39)
F1									
MCC	0.75(0.41)	0.34(0.32)	0.29(?)	0.30(0.38)	0.94(0.89)	0.60(0.76)	0.60(0.93)	1(0.47)	0.60(?)
J48DT	0.72(0.36)	0.55(0.41)	0.52(0.35)	0.63(0.49)	0.85(0.92)	0.64(0.54)	0.90(0.90)	1(1)	0.73(0.62)
P	0.99(1)	0.83(0.84)	0.80(0)	0.03(0.43)	0.51(0.33)	0.18(0.26)	0.12(0.14)	0.27(0.58)	0.47(0.43)
R	1(0.84)	0.92(0.83)	0.78(0.45)	0.76(0.29)	0.78(0.58)	0.36(0.10)	0.39(0.27)	0.34(0.21)	0.66(0.47)
F1	0.85(0.57)	0.50(0.46)	0.43(?)	0.05(0.40)	0.66(0.48)	0.27(0.38)	0.20(0.25)	0.42(0.52)	0.42(?)
MCC	0.83(0.53)	0.69(0.55)	0.62(0.39)	0.69(0.36)	0.81(0.71)	0.46(0.16)	0.55(0.42)	0.50(0.34)	0.64(0.43)
P	0.82(0.54)	0.46(0.41)	0.36(?)	0.06(0.31)	0.66(0.51)	0.28(0.40)	0.23(0.33)	0.49(0.45)	0.42(?)
R	0.82(0.52)	0.67(0.51)	0.57(0.30)	0.64(0.31)	0.79(0.70)	0.43(0.19)	0.56(0.46)	0.55(0.43)	0.63(0.43)
F1									
MCC	0.36(0.40)	0.28(0.29)	0.39(0)	0(0.13)	0.33(0.38)	0.21(0.27)	0.17(0.37)	0.21(0.23)	0.26(0.26)
P	0.26(0.26)	0.35(0.31)	0.25(0.16)	0.24(0.24)	0.51(0.54)	0.24(0.17)	0.15(0.25)	0.10(0.15)	0.26(0.26)
R	0.82(0.46)	0.16(0.18)	0.09(0)	0(0.12)	0.68(0.48)	0.15(0.24)	0.22(0.06)	0.18(0.70)	0.28(0.28)
F1	0.70(0.48)	0.60(0.52)	0.23(0.07)	0.15(0.20)	0.19(0.27)	0.32(0.16)	0.04(0.38)	0.01(0.05)	0.28(0.26)
MCC	0.50(0.42)	0.20(0.22)	0.14(0)	0(0.12)	0.44(0.42)	0.17(0.25)	0.19(0.10)	0.19(0.34)	0.23(0.23)
P	0.37(0.34)	0.44(0.39)	0.24(0.09)	0.19(0.21)	0.28(0.36)	0.28(0.17)	0.07(0.30)	0.01(0.07)	0.23(0.24)
R	0.44(0.34)	0.13(0.14)	0.13(−0.01)	−0.04(0.01)	0.36(0.33)	0.08(0.15)	0.06(0.10)	0.09(0.24)	0.16(0.16)
F1	0.29(0.23)	0.36(0.29)	0.13(0.03)	0.10(0.11)	0.26(0.32)	0.16(0.05)	0.01(0.19)	0(0.01)	0.16(0.15)
MCC									
P	0.82(0.60)	0.38(0.38)	0.33(?)	0.25(0.30)	0.71(0.36)	0.25(0.36)	0.32(0.37)	0.48(0.37)	0.44(?)
R	0.59(0.47)	0.48(0.37)	0.43(0.26)	0.54(0.32)	0.66(0.58)	0.37(0.38)	0.50(0.34)	0.41(0.36)	0.50(0.39)
F1	0.93(0.45)	0.60(0.40)	0.45(0)	0.15(0.15)	0.60(0.74)	0.26(0.47)	0.25(0.12)	0.37(0.59)	0.45(0.36)
MCC	0.92(0.58)	0.66(0.72)	0.33(0.33)	0.47(0.17)	0.67(0.64)	0.36(0.07)	0.30(0.42)	0.37(0.20)	0.51(0.39)
	0.87(0.51)	0.46(0.39)	0.38(?)	0.18(0.20)	0.65(0.49)	0.25(0.33)	0.28(0.18)	0.41(0.45)	0.44(?)
	0.72(0.52)	0.55(0.49)	0.37(0.29)	0.50(0.22)	0.66(0.61)	0.36(0.12)	0.37(0.38)	0.39(0.26)	0.49(0.36)
	0.85(0.64)	0.38(0.30)	0.28(?)	0.11(0.13)	0.61(0.42)	0.15(0.22)	0.19(0.15)	0.35(0.37)	0.36(?)
	0.69(0.45)	0.49(0.42)	0.30(0.18)	0.44(0.15)	0.61(0.55)	0.27(0.12)	0.32(0.28)	0.31(0.19)	0.43(0.29)

Table A6: Multi-Class classification results for patterns extracted using VMSP(CloFast)

Cl	Adware	Backdoor	Downloader	Dropper	Virus	Worms	Trojan	Spyware	W. Avg.
SVM									
P	0.69(0.27)	0.29(0.26)	0.15(0.44)	0.29(0.323)	0.32(0.34)	0.41(0.31)	0.28(0.19)	0.46(0.28)	0.36(0.33)
R	0.75(0.37)	0.30(0.25)	0.34(0.28)	0.35(0.16)	0.29(0.35)	0.46(0.26)	0.19(0.29)	0.21(0.20)	0.36(0.27)
F1	0.40(0.23)	0.22(0.63)	0.03(0.04)	0.51(0.06)	0.69(0.61)	0.36(0.24)	0.41(0.16)	0.12(0.35)	0.34(0.29)
MCC	0.44(0.30)	0.62(0.60)	0.32(0.01)	0.28(0.12)	0.32(0.22)	0.24(0.23)	0.10(0.09)	0.32(0.47)	0.33(0.25)
RF	0.50(0.24)	0.25(0.37)	0.05(0.07)	0.37(0.10)	0.43(0.44)	0.38(0.27)	0.33(0.17)	0.19(0.31)	0.31(0.25)
P	0.55(0.33)	0.40(0.36)	0.33(0.01)	0.31(0.14)	0.30(0.27)	0.31(0.24)	0.13(0.13)	0.25(0.28)	0.32(0.22)
R	0.47(0.15)	0.16(0.27)	0.01(0.10)	0.26(0.16)	0.36(0.35)	0.30(0.18)	0.22(0.07)	0.18(0.20)	0.25(0.18)
F1	0.53(0.25)	0.31(0.25)	0.24(0.03)	0.23(0.04)	0.20(0.20)	0.27(0.14)	0.21(0.10)	0.19(0.15)	0.27(0.14)
MCC									
kNN	0.96(0.70)	0.85(0.45)	0.80(0.42)	0.92(0.48)	0.98(0.59)	0.91(0.32)	0.78(0.38)	0.84(0.43)	0.88(0.47)
P	0.99(0.70)	0.89(0.44)	0.90(0.50)	0.96(0.60)	0.93(0.85)	0.92(0.73)	0.93(0.90)	0.94(1)	0.93(0.71)
R	0.98(0.66)	0.80(0.39)	0.82(0.40)	0.94(0.46)	0.90(0.63)	0.88(0.42)	0.87(0.38)	0.86(0.42)	0.88(0.47)
F1	0.98(1)	0.97(0.94)	0.92(0.65)	0.93(0.66)	0.92(0.58)	0.93(0.40)	0.91(0.42)	0.90(0.38)	0.93(0.63)
MCC	0.97(0.68)	0.82(0.41)	0.81(0.41)	0.93(0.47)	0.94(0.61)	0.89(0.36)	0.82(0.38)	0.85(0.42)	0.88(0.47)
NB	0.98(0.82)	0.92(0.60)	0.91(0.57)	0.95(0.63)	0.93(0.69)	0.92(0.52)	0.92(0.57)	0.92(0.55)	0.93(0.62)
P	0.96(0.63)	0.80(0.34)	0.78(0.33)	0.92(0.39)	0.93(0.55)	0.88(0.26)	0.80(0.29)	0.83(0.34)	0.86(0.39)
R	0.98(0.81)	0.91(0.58)	0.90(0.50)	0.94(0.57)	0.92(0.67)	0.91(0.50)	0.91(0.58)	0.91(0.59)	0.92(0.60)
F1									
MCC	0.94(0.47)	0.71(0.32)	0.74(0.39)	0.92(0.59)	0.98(0.95)	0.92(0.78)	0.97(0.88)	1(1)	0.90(0.676)
J48DT	0.96(0.70)	0.87(0.44)	0.87(0.50)	0.95(0.60)	0.93(0.85)	0.92(0.73)	0.99(0.90)	1(1)	0.93(0.71)
P	1(1)	1(0.91)	0.94(0.55)	0.94(0.47)	0.90(0.41)	0.88(0.11)	0.68(0.15)	0.71(0.16)	0.88(0.47)
R	1(1)	0.99(0.94)	0.98(0.65)	0.95(0.66)	0.93(0.58)	0.92(0.40)	0.85(0.42)	0.85(0.38)	0.93(0.63)
F1	0.97(0.64)	0.83(0.47)	0.83(0.46)	0.93(0.52)	0.94(0.57)	0.90(0.19)	0.80(0.25)	0.83(0.27)	0.88(0.42)
MCC	0.98(0.82)	0.93(0.60)	0.92(0.57)	0.95(0.63)	0.93(0.69)	0.92(0.52)	0.91(0.57)	0.91(0.55)	0.93(0.62)
P	0.96(0.62)	0.82(0.44)	0.81(0.37)	0.92(0.47)	0.93(0.59)	0.88(0.26)	0.79(0.33)	0.82(0.37)	0.87(0.43)
R	0.98(0.81)	0.92(0.58)	0.91(0.50)	0.94(0.57)	0.92(0.67)	0.91(0.50)	0.90(0.58)	0.91(0.59)	0.92(0.60)
F1									
MCC	0.66(0.21)	0.45(0.27)	0(0.15)	0.41(0.34)	0.33(0.36)	0.28(0.30)	0.37(0.17)	0.31(0.22)	0.35(0.25)
J48DT	0.72(0.44)	0.28(0.25)	0.29(0.19)	0.46(0.21)	0.23(0.37)	0.37(0.27)	0.24(0.28)	0.25(0.23)	0.35(0.28)
P	0.41(0.07)	0.20(0.56)	0(0.03)	0.46(0.08)	0.73(0.53)	0.47(0.27)	0.50(0.27)	0.20(0.30)	0.37(0.26)
R	0.29(0.40)	0.73(0.60)	0.23(0.34)	0.22(0.12)	0.31(0.18)	0.30(0.09)	0.11(0.10)	0.28(0.27)	0.31(0.26)
F1	0.50(0.10)	0.27(0.37)	0(0.05)	0.43(0.13)	0.45(0.42)	0.35(0.28)	0.43(0.21)	0.24(0.25)	0.33(0.23)
MCC	0.42(0.42)	0.40(0.35)	0.26(0.25)	0.30(0.15)	0.26(0.24)	0.33(0.14)	0.15(0.15)	0.26(0.25)	0.30(0.24)
P	0.47(0.05)	0.24(0.26)	0.21(0.01)	0.34(0.11)	0.38(0.33)	0.24(0.18)	0.34(0.07)	0.17(0.13)	0.27(0.14)
R	0.42(0.34)	0.32(0.25)	0.17(0.11)	0.26(0.07)	0.14(0.19)	0.24(0.09)	0.09(0.10)	0.15(0.14)	0.22(0.16)
F1									
MCC	0.72(0.71)	0.62(0.31)	0.57(0.34)	0.66(0.56)	0.71(0.59)	0.68(0.38)	0.62(0.31)	0.66(0.23)	0.66(0.43)
P	0.82(0.68)	0.61(0.41)	0.71(0.35)	0.73(0.46)	0.66(0.62)	0.67(0.47)	0.65(0.47)	0.67(0.41)	0.69(0.49)
R	0.91(0.47)	0.65(0.43)	0.59(0.42)	0.66(0.40)	0.75(0.49)	0.59(0.23)	0.67(0.26)	0.47(0.51)	0.66(0.40)
F1	0.85(0.64)	0.74(0.65)	0.60(0.36)	0.71(0.38)	0.64(0.58)	0.72(0.34)	0.64(0.35)	0.63(0.53)	0.69(0.48)
MCC	0.80(0.56)	0.63(0.36)	0.58(0.38)	0.66(0.46)	0.73(0.53)	0.63(0.28)	0.64(0.28)	0.55(0.36)	0.65(0.40)
	0.83(0.66)	0.67(0.50)	0.65(0.35)	0.72(0.41)	0.65(0.60)	0.69(0.40)	0.64(0.40)	0.65(0.46)	0.69(0.47)
	0.78(0.53)	0.58(0.26)	0.52(0.28)	0.61(0.41)	0.69(0.47)	0.58(0.22)	0.59(0.19)	0.50(0.25)	0.61(0.33)
	0.81(0.61)	0.62(0.43)	0.61(0.26)	0.68(0.34)	0.60(0.54)	0.65(0.33)	0.59(0.33)	0.60(0.38)	0.65(0.40)

Table A7: String-based multi-class classification results for patterns extracted using TKS(CM-SPAM)

Cl	Adware	Backdoor	Downloader	Dropper	Virus	Worms	Trojan	Spyware	W. Avg.
NBMT									
P	1(0.67)	0.38(0.37)	0.32(0.36)	0(0.42)	0.42(0.29)	0.29(0.36)	0.21(0.31)	0.52(0.38)	0.39(0.40)
R	0.62(0.50)	0.58(0.66)	0.22(0.25)	0.28(0.42)	0.41(0.42)	0.23(0.26)	0.44(0.23)	0.37(0.59)	0.39(0.42)
F1	0.91(0.41)	0.75(0.53)	0.66(0.41)	0(0.09)	0.60(0.64)	0.14(0.20)	0.13(0.28)	0.30(0.40)	0.43(0.37)
MCC	0.57(0.43)	0.41(0.42)	0.47(0.29)	0.14(0.09)	0.50(0.75)	0.45(0.19)	0.10(0.60)	0.13(0.06)	0.34(0.35)
	0.95(0.50)	0.50(0.44)	0.43(0.38)	0(0.14)	0.49(0.40)	0.19(0.25)	0.16(0.29)	0.38(0.39)	0.39(0.35)
	0.60(0.46)	0.48(0.51)	0.30(0.26)	0.18(0.15)	0.45(0.54)	0.30(0.22)	0.17(0.34)	0.19(0.11)	0.33(0.32)
	0.94(0.47)	0.44(0.35)	0.35(0.28)	−0.01(0.15)	0.41(0.31)	0.13(0.19)	0.07(0.20)	0.33(0.30)	0.33(0.28)
	0.54(0.40)	0.43(0.47)	0.17(0.15)	0.12(0.15)	0.37(0.48)	0.18(0.13)	0.16(0.23)	0.15(0.16)	0.27(0.27)

Table A8: String-based multi-class classification results for patterns extracted using VMSP(CloFast)

Cl	Adware	Backdoor	Downloader	Dropper	Virus	Worms	Trojan	Spyware	W. Avg.
NBMT									
P	0.52(0.44)	0.49(0.32)	0.20(0.36)	0.44(0.01)	0.40(0.29)	0.25(0.24)	0.47(0.62)	0.32(0.47)	0.38(0.41)
R	0.62(0.57)	0.58(0.56)	0.25(0.21)	0.39(0.50)	0.50(0.43)	0.30(0.25)	0.39(0.25)	0.37(0.42)	0.41(0.40)
F1	0.38(0.35)	0.28(0.40)	0.16(0.21)	0.21(0.56)	0.53(0.61)	0.67(0.55)	0.37(0.10)	0.22(0.30)	0.35(0.33)
MCC	0.60(0.50)	0.44(0.36)	0.28(0.37)	0.36(0.14)	0.41(0.66)	0.56(0.48)	0.24(0.14)	0.31(0.18)	0.40(0.35)
	0.44(0.39)	0.34(0.35)	0.17(0.26)	0.28(0.17)	0.46(0.40)	0.36(0.33)	0.41(0.17)	0.26(0.36)	0.34(0.31)
	0.61(0.53)	0.46(0.44)	0.26(0.27)	0.38(0.21)	0.45(0.52)	0.39(0.33)	0.29(0.18)	0.33(0.25)	0.40(0.34)
	0.38(0.31)	0.29(0.25)	0.07(0.20)	0.24(0.26)	0.37(0.30)	0.27(0.22)	0.34(0.21)	0.18(0.31)	0.27(0.26)
	0.55(0.47)	0.39(0.39)	0.15(0.14)	0.29(0.21)	0.38(0.45)	0.30(0.21)	0.23(0.10)	0.25(0.21)	0.32(0.27)