Combining Machine and Automata Learning for Network Traffic Classification

Zeynab Sabahi-Kaviani, Fatemeh Ghassemi, Zahra Alimadadi

Combining Machine and Automata Learning for Network Traffic Classification

Zeynab Sabahi-Kaviani[1], Fatemeh Ghassemi[1,2], and Zahra Alimadadi[1]

[1] School of Electrical and Computer Engineering,
University of Tehran, Tehran, Iran
[2] School of Computer Science, Institute for Research in Fundamental Sciences,
PO.Box 19395-5746, Tehran, Iran

Abstract. Viewing the generated packets of an application as the words of a language, automata learning can be used to derive the behavioral packet-based model of applications. The alphabets of the learned automata, manually defined in terms of packets, may cause overfitting. As some packets always appear together, we apply machine learning techniques to automatically define the alphabet set based on the timing and statistical features of packets. Using the learned automata models, the classifier should detect the accepted words of the models in the input.To improve this time-consuming process, we present a framework, called NeTLang, that identifies the application model in terms of k-testable languages. The classification problem is reduced to observing only $\Theta(k)$ symbols from the input with the help of machine learning techniques. Our framework utilizes the two diverse automata learning and machine learning techniques to build on their strengths (to be fast and accurate) and to eliminate their weaknesses (i.e., ignoring temporal relations among packets). According to our results, NeTLang outperforms the state-of-the-art methods using each technique alone.

Keywords: Automata learning, Machine Learning, Traffic classification, Model Inference.

1 Introduction

Characterizing the network traffic and identifying running applications play an important role in several network administration tasks such as protecting against malicious behaviors, firewalling, and balancing bandwidth usage. Recently, dynamic port assignment and encryption protocol usage have considerably reduced the performance of the classic traffic classification methods, including port-based and deep packet inspection. This leads researchers to apply Machine-Learning techniques and behavioral pattern detection for traffic classification. Machine-Learning approaches classify network traffic based on statistical features with the granularity of flow [1, 2] or packet [3], and hence, they ignore temporal relations among flows, and as a result, their false positive rates are not negligible although they are fast. In behavioral classification methods [4–6], an expert extracts specific behavioral aspects of a particular application or application type

for the classification purpose. For instance, link establishment topology was used as the distinctive metric to classify P2P-TV traffic in [5].

Assuming a packet trace as a word of the language of an application, one can derive an automaton modeling the traffic behavior of that application. Automata learning approaches have been recently used to automatically derive the model of applications [7,8], network protocols [9,10], or Botnet behavior [8]. The alphabets of the learned automata are either manually defined by domain experts which is not straightforward, or in terms of packets which may cause overfitting. As some packets always appear together, we can consider a sequence of related packets together as a symbol of the alphabet. To this aim, we apply machine learning techniques to automatically define the alphabet set based on the timing and statistical features of packets.

The derived automata are used for traffic classification. A packet trace is classified into an application if the model of that application accepts it. Using automata learning methods, the classification problem is constrained to observe the complete trace of an application to verdict its acceptance/rejection. To tackle this challenge, inspired by [11], we upgrade the detection of an application based on partial observation of a trace, a window of size k, and derive a model that accepts a k-Testable language in the Strict Sense (k-TSS) [12]. K-TSS, a class of regular languages, also known as *window language*, allows to locally accept or reject a word by a sliding window parser of size k. We relax the acceptance condition of automata learning using machine learning by defining a proximity metric to be compatible with the local essence of the learned language. The proposed proximity metric is defined as a distance function. We have implemented our approach in a framework called Network Traffic Language learner, NeTLang. We evaluate the performance of our approach by applying it to real-world network traffic and compare it with machine and automata learning approaches. We achieved F1-Measure of 97% for both application identification and traffic characterization tasks.In summary, first, we learn the alphabet using a machine learning technique. Then, the network language is learned through an automata learning approach. Finally, the classifier identifies the classes based on our defined distance metrics on the input and the learned models. Our method makes these **contributions**: 1) Utilizing locally testable language learning in the traffic classification problem, 2) Extracting the domain-based alphabet automatically, 3) Upgrading the word acceptance by a new proximity metric. With these contributions, the following improvements are brought into traffic classification:

- Considering a sequence of related packets as the appropriate granularity of the problem, instead of per-packet detection which is too fine-grained or per-flow detection which is too coarse-grained,
- Providing highly accurate models for applications as our automata learning approach considers the temporal relation among flows and the way they are interleaved,
- Decreasing the classification time by considering only some first packets of a trace with a help of a novel distance function.

2 Background on Automata Learning

In this section, we provide some background on automata learning concepts used in our methodology. Learning a regular language from given positive samples (words belonging to the language) is a common problem in grammatical inference. To solve this problem, many algorithms were proposed to find the smallest deterministic finite automaton (DFA) that accepts the positive examples. In this paper, we focus on learning k-testable languages in the strict sense, a subset of a regular language, called k-TSS, initially was introduced by [12]. In such a language, words are determined by allowed three sets of prefixes and suffixes of length *k-1* and substrings of length k. It has been proven that it is possible to learn k-TSS languages in the limit [13]. To learn this language, the only effort is to scan the accepting words while simultaneously constructing the allowed three set. The locally testable feature of this language makes it appropriate for network traffic classification and other domains, such as pattern recognition [14] and DNA sequence analysis [15]. In the following, we provide the formal definition of k-TSS language taken from [16].

Definition 1 (k-test Vector). *Let $k > 0$, a k-test vector is determined by a 5-tuple $Z = \langle \Sigma, I, F, T, C \rangle$ where*

- *Σ is a finite alphabet,*
- *$I \subseteq \Sigma^{(k-1)}$ is a set of allowed prefixes of length less than k,*
- *$F \subseteq \Sigma^{(k-1)}$ is a set of allowed suffixes of length less than k,*
- *$T \subseteq \Sigma^{k}$ is a set of allowed segments, and*
- *$C \subseteq \Sigma^{<k}$ contains all strings of length less than k.*

Definition 2 (k-TSS Language). *Let $Z = \langle \Sigma, I, F, T, C \rangle$ be a k-test vector, for some $k > 0$. Then Language $\mathcal{L}(Z)$ in the strict sense (k-TSS) is computed as:*

$$\mathcal{L}(Z) = [(I\Sigma^* \cap \Sigma^* F) - \Sigma^*(\Sigma^k - T)\Sigma^*] \cup C$$

For instance, consider $k = 3$ and $Z = \langle \Sigma = \{a, b\}, I = \{ab\}, F = \{ab, ba\}, T = \{aba, abb, bba\}, C = \{ab\}\rangle$, then, $aba, abba \in \mathcal{L}(Z)$ since they are preserving the allowed sets of Z, while $bab, abb, abab, a$ do not belong to $\mathcal{L}(z)$ because, in order, they violate I ($ba \notin I$), F ($bb \notin F$), T ($bab \notin T$), and C ($a \notin C$). To construct the k-test vector of a language, we scan the accepted word by a k-size frame. By scanning the word $abba$ by a 3-size frame, ab, ba, and abb, bba are added to I, F, and T, respectively.

In our problem, we produce words such that their length is greater than or equal to k. It means that C always is empty. Hence, for simplicity, we eliminate C from the k-TSS vector for the rest of the paper.

3 Problem Statement and Basic Definitions

In this section, we first formally state the problem, and then we define some network concepts used in our methodology.

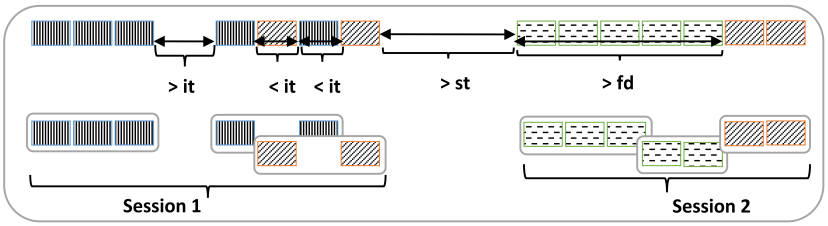


Fig. 1: Schematic representation of applying the timing parameters on a packet trace. Different backgrounds represent different network flows, and the floating boxes in the bottom represent derived network units.

Once a user runs an application, several *network connections* are established for transferring necessary data and control information, named network flow. A *network flow* consists of a sequence of packets, sent or received, having the same source IP, source port, destination IP, destination port, and protocol. Due to concurrency of the network, packet sequences of flows may be interleaved. In other words, the packet trace of an application is an interleaving of several flows.

We assume that for the applications set $A = \{App_1, App_2, ..., App_m\}$, the collection Π, consisting of the network traces, is given as an input, where $\pi_{App_i} \in \Pi$ is the captured traffic of App_i over a period of time. The goal is to distinguish which application(s) is running on a system by exploring the system traffic, named π. As the output, we label different parts of π with appropriate labels of application names.

To solve the problem, the main task is generating application models to be used by a classifier. Our intuition is that each application App_i has its network communicating language $\mathcal{L}(App_i)$ which differs from others. Since learning a language requires some of its positive examples, an important subproblem is to break down the input trace π_{App_i} of application App_i to a list of words $W_i = [w_1, w_2, ...w_n]$, where $w_j \in \mathcal{L}(App_i)$ is a successive subsequence of π_{App_i}. Instead of defining the alphabet in terms of packets, we consider a symbol for the sequence of packets that always appear together. To define the alphabet, we define some network concepts used in breaking down a trace:

Definition 3 (Session). *A session is a sequence of packets to/from the same user IP addresses where every two consecutive packets have a time interval of less than a* ***session threshold*** *(st). We have defined this concept to consider the latency when switching between two applications or within two different tasks of the execution of an application.*

Definition 4 (Flow-Session). *A Flow-Session is defined based on flows' inactive timeout. A flow is considered to be expired when it has been idle for a duration of more than a threshold called* ***inactive timeout*** *(it). Each flow in a session is split into shorter units called flow-session when two consecutive packets have a time interval of more than the inactive timeout. We have considered this concept to distinguish different phases of an application task.*

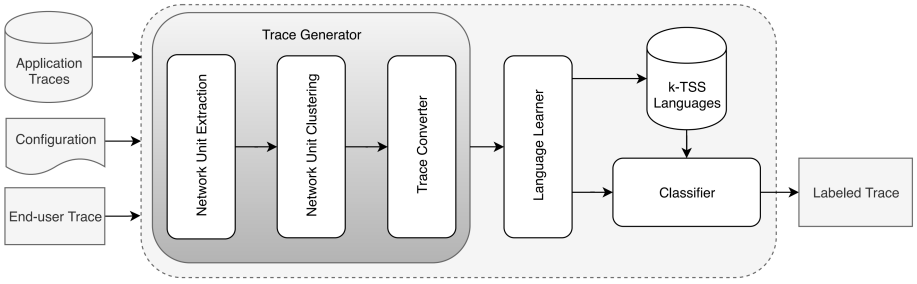


Fig. 2: The architectural view of NeTLang.

According to these definitions, for instance, a new *session* is started when switching from a Commit command in Git bash to an Add command, while each of these commands comprises some *flow-session*s corresponding to different steps of executing them such as initializing or transferring data.

Definition 5 (Network Unit). *With the interleaved feature of network flows, to be able to observe the relation among flows, each flow-session is chunked into shorter units having the duration of at most a* ***flow duration*** *(fd), named network unit. The flow duration criterion is another timing constraint of our problem expressing an upper bound time until when a flow can last.*

Figure 1 shows an example of applying the above concepts on a trace. The trace is split into two *sessions* based on the st. The *flow-sessions* are extracted based on the it metric, and also, the fd threshold leads to the creation of smaller flow parts defined as network units. From the above definitions, by using the three configuration variables naming st, fd, and it, we break an input into some sub-traces to identify the network units.

We define function *Map: Network Units* $\rightarrow \Sigma$ to map the derived network units to the *network alphabet* to be processable by the automata techniques. Each symbol $s \in \Sigma$ stands for an unbreakable packet sequence of a flow.

4 Methodology

Figure 2 illustrates the overall scheme of our methodology. There are three main modules *Trace Generator*, *Language Learner*, and *Classifier*. The trace generator module converts the packet traces of each application into a word list by first splitting the traces by using the timing parameters and then applying the Map function to identified network units. Then, the k-TSS network language of the application is learned from its words by the Language Learner module. By applying this process for all applications, a database of k-TSS languages is obtained. To classify traffic of a system, first, the Trace Generator module extracts its sessions and their network units using the timing parameters. Then, it converts the sessions to the symbolic words to prepare the inputs of the Language

Table 1: Flow Statistical Features: The f/b stands for forward/backward flow.

#	Feature	Description
1	TotalPktf/b	Count of packets sent/received within a network unit.
2	TotalLf/b	Sum of packets' length sent/received within a network unit.
3	MinLf/b	Minimum length of packet sent/received within a network unit.
4	MeanLf/b	Average length of packet sent/received within a network unit.
5	MaxLf/b	Maximum length of packet sent/received within a network unit.
6	StdLf/b	Standard deviation of packets length sent/received within a unit.
7	PktCntRf/b	Rate of TotalPktf/b to total number of packets within a network unit.
8	DtSizeRf/b	Rate of TotalLf/b to sum of all packets' length within a network unit.
9	AvgInvalf/b	Average of sent/received packets time interval within a network unit.

Learner module. Finally, the Classifier module compares the learned language of each session of the given traffic against the languages of the database to label them. We describe each module in detail in the following.

4.1 Trace Generator

The goal of the *Trace Generator* module is to transform the packet traces of each application into traces of smaller units, while preserving the relation among these units, as the letters of a language. To do so, at first, packet traces are split into smaller units, named *network unit*, by using the three timing parameters, introduced in Section 3. Then, these units are categorized based on their higher-layer protocol, and each category is clustered separately and named upon it. Consequently, the names of clusters constitute the *Network Alphabet*. Finally, the network alphabets are put in together to form our traces. Therefore, the input of this subproblem is Π containing m application traces and the timing parameters st, fd, and it, while the output is a set of word list $S = \{W_1, W_2, ..., W_m\}$, where W_i is a list including the generated words of language App_i.

Network Unit Extraction *Network Units* are extracted during three steps. At first, network traces are split into *sessions* based on the *session threshold*. Then, *flow-sessions* are extracted based on the *inactive timeout*. After that, *flow-sessions* are divided into smaller units according to the flow duration constraint. Figure 1 shows the result of applying this module on a given trace to produce sessions, flow-sessions, and network units.

Network Unit Clustering For naming network units, there is a need to group the similar ones and name them upon it. As there is no information on the number of units or their labels, an unsupervised machine learning algorithm named Kmeans++ is used for clustering these units. The only information available for each flow are those stored in the frame header of its packets. Therefore, the clustering algorithm could benefit from the knowledge of the unit higher-layer protocol. Consequently, flows having the same protocol are at worst clustered

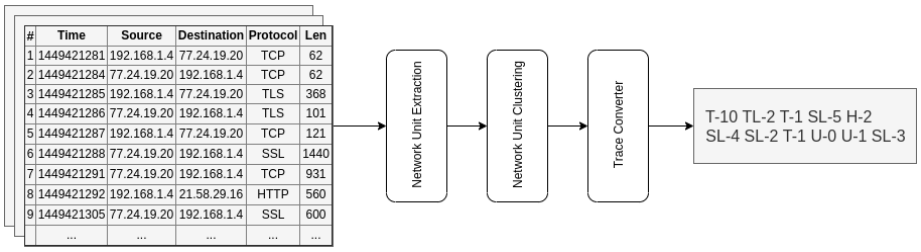


Fig. 3: Trace Generation at a glance.

together and named upon their protocol. The packets of each flow can be further clustered in terms of multiple statistical features listed in Table 1. Two groups of features are used for clustering *network unit*, the first group consists of features number one to six, and the second group includes features number three to nine. Regarding these two groups, the statistical features group (*stats*) is another configuration of this problem.

Moreover, it is required to set the number of clusters before performing the Kmeans++ algorithm. To do so, the number of clusters is automatically determined with the help of the elbow [17] method in our approach.

Our defined Map function returns a concatenation of the symbol abbreviating packet protocol name and a natural number indicating its cluster number (to distinguish network units). For example, `TL-5` corresponds to a packet sequence belonging to cluster 5 of the TLS protocol.

Trace Converter In this step, first, network units are sorted based on the timestamps of their first packet. This sorting enables us to preserve the ordering relation among these units, which is ignored in previous machine learning studies. Then, the list of words $W = [w_1, w_2, ...w_n]$ is created. Each w_j equals to $\alpha_1\ \alpha_2\ \alpha_3\ ...\ \alpha_{len(w_j)}$ where $len(w_j)$ shows the length of the jth word of application.

Figure 3 shows the result of applying the *Trace Generation* module where *st*, *ft*, and *fd* are set to 10, 5, and 5, respectively. For the given trace, its packets are grouped as $[(1,2),(3,4),(5,7),(6),(8),(9)]$, composing network units `T-10`, `TL-2`, `T-1`, `SL-5`, `H-2` and `SL-4`, respectively, named upon their clusters. This packet trace is split after packet 8, because of *st* constraint. Although packets 1, 2, 5, and 7 belong to a TCP flow, they are not grouped together, because of the flow duration limitation. This flow is broken between packet 2 and packet 5. Moreover, packets $(1,2)$, $(3,4)$, and $(5,7)$ are grouped together as the inactive timeout between packets is less than *ft*.

4.2 Language Learner

By applying the trace generator module on Π, the set of words list $S = \{W_1, W_2, ..., W_m\}$ is achieved, where $W_i = [w_1, w_2, ...w_n]$ corresponds to the list of the

words for the application $App_i \in A$. The word $w_j = \alpha_1\ \alpha_2\ \alpha_3\ ...\ \alpha_{len(w_j)}$ is a finite sequence of symbols of the network alphabet, where $len(w_j)$ determine the length of the sequence w_j. To learn language $\mathcal{L}(App_i)$ as a k-testable language, we should identify its k-test vector $Z = \langle \Sigma, I, F, T \rangle$. The value of k is the second input of this subproblem.

We perform a preparation task for each word list (W_i) to handle short words. Since words with the length shorter than k ($len(w) < k$) do not involve in producing I,T and, F sets, we attach them to the beginning of the next word in the list. Finally, $\mathcal{L}(App_i)$ is learned by scanning all words $w_j \in W_i$ with a k-window sliding parser and computing Σ, I, F, and T. As a result of executing the language learner module on word lists of all applications, the k-TSS languages database is provided.

Running Example: For the traces of Figure 3 let $k = 3$, then, we obtain:

- $\Sigma = \{$`H-2, SL-2, SL-3, SL-4, SL-5, T-1, T-10, TL-2, U-0, U-1`$\}$
- $I = \{$`SL-4 SL-2, T-10 TL-2`$\}$
- $F = \{$`SL-5 H-2, U-1 SL-3`$\}$
- $T = \{$`SL-2 T-1 U-0, SL-4 SL-2 T-1, T-1 SL-5 H-2, T-1 U-0 U-1, T-10 TL-2 T-1, TL-2 T-1 SL-5, U-0 U-1 SL-3`$\}$

4.3 Classifier

The classifier is responsible for characterizing the given packet trace π according to k-TSS languages. Since π tends to be an interleaving of execution of multiple applications, the classifier should partition that and label each part of the π with an appropriate application.

We use the trace generator module to split it to sub-traces of π and map them as a list of words $W = [w_1, w_2, ..., w_n]$. The goal of the classifier is to label the W members.

Preparation Tasks Some tasks are required to prepare the input trace concerning the value of k.

- Handle Short Words: this task is done similarly to what we mentioned at the Language Learner module based on the k value.
- Trim Words: Instead of sliding a k-window on a too lengthy word, we can only inspect its first symbols. Therefore, we break it down into a coefficient of k. In practice, using $4k$, we achieved acceptable accuracy (1.5 percent higher than when only observing $3k$ and considering $5k$ words does not increase its accuracy much more). This operation determines an upper bound time for detection.

After preparation tasks, we obtain the modified word list $W' = [w'_1, w'_2, ..., w'_n]$. Now, we learn a list of k-TSS languages for each $w' \in W'$ as $\mathcal{L}(w')$ by a k-size frame scanner. For instance, for the given $w' =$ `SL-4 SL-2 T-10 TL-2 T-1 U-2`, $\mathcal{L}(w')$ is obtained by k-TSS vector $Z(w')$ as $\langle \Sigma = \{$`SL-2, TL-2, T-1, U-2,`

SL-4, T-10}, I = {SL-4 SL-2,}, F = {T-1 U-2}, T = {SL-4 SL-2 T-10, SL-2 T-10 TL-2, T-10 TL-2 T-1,TL-2 T-1 U-2}⟩.

To specify each word's class, we observe its segments instead of exploring the whole word to be more noise tolerable. To this aim, we reduce the classification problem to find the similarity between the generated k-TTS languages of the trace and the ones in the database. For each k-TSS language in the database, we calculate its proximity with $\mathcal{L}(w')$ by a *distance* function defined as:

Definition 6 (Distance Function). *Distance function D measures the proximity metric between two k-TSS languages. Let $Z = \langle \Sigma, I, F, T \rangle$ be the k-test vector of $\mathcal{L}(w')$ and $Z_i = \langle \Sigma_i, I_i, F_i, T_i \rangle$ be the k-test vector of $\mathcal{L}(App_i)$. Then, $D(\mathcal{L}(w'), \mathcal{L}(App_i))$ is computed by five auxiliary variables, measuring the sets difference fraction: $\triangle T$, $\triangle T_i$, $\triangle \Sigma$, $\triangle I$ and $\triangle F$ (defined in Equation 1).*

$$\triangle T = \frac{T - T_i}{T}, \triangle T_i = \frac{T_i - T}{T_i}, \triangle \Sigma = \frac{\Sigma - \Sigma_i}{\Sigma}, \triangle I = \frac{I - I_i}{I}, \triangle F = \frac{F - F_i}{F}. \tag{1}$$

$$D(\mathcal{L}(w'), \mathcal{L}(App_i)) = \overline{\triangle' T}\ \overline{\triangle' T_i}\ \overline{\triangle' \Sigma}\ \overline{\triangle' I}\ \overline{\triangle' F} \tag{2}$$

We assume a priority among these delta metrics as $\triangle T$, $\triangle T_i$, $\triangle \Sigma$, $\triangle I$ and $\triangle F$. Since T carries more information of words than I and F, we assign the highest priority to $\triangle T$. Also, we give $\triangle T$ higher priority than $\triangle T_i$, as T is generated from a test trace in contrast to T_i which is generated from a number of traces of an application. We specify the next priority to $\triangle \Sigma$ to take into account the alphabet sets differences. We assign the next priorities to $\triangle I$ and then $\triangle F$, respectively, because the trimming operation of the preparation phase may impact on the end of the words while their initials are not modified. Finally, we convert the value of delta metrics from a float number in the range $[0, 1]$ to an integer number in the range $[0, 99]$, renaming them by a prime sign, for example, $\triangle' T$. By this priority, the distance function is defined as given by Equation 2.

A word w is categorized in class j if the k-TSS language of w has the minimum distance with the k-TSS language of App_j among all the applications:

$$Class(w') = j \iff \text{if } D(\mathcal{L}(w'), \mathcal{L}(App_j)) = argmin_{\forall App_i \in |A|}(D(\mathcal{L}(w'), \mathcal{L}(App_i))) \tag{3}$$

Considering the running example ($\mathcal{L}(App_i)$ in Section 4.2 and $\mathcal{L}(w')$), the defined delta metrics are obtained as $\triangle T = 0.75$ ($\triangle' T = 74$), $\triangle T_i = 0.85$ ($\triangle' T_i = 84$), $\triangle \Sigma = 0.16$ ($\triangle' \Sigma = 16$), $\triangle I = 0$ ($\triangle' I = 0$), and $\triangle F = 1$ ($\triangle' F = 99$) and finally, the distance is computed as $D(\mathcal{L}(w'), \mathcal{L}(App_i)) = 7484160099$.

5 Evaluation

To evaluate the proposed framework, we have fully implemented it and have run multiple experiments. We evaluate our method for two classification tasks:

traffic characterization and application identification. Traffic characterization deals with determining the application category of a trace such as *Chat*, and the goal of application identification is to distinguish the application of a trace such as *Skype*. To evaluate the performance of the proposed framework, we have used Precision (Pr), Recall (Rc), and F1-Measure (F1). The mathematical formula for these metrics is provided in Equation 4.

$$Recall = \frac{TP}{TP + FN}, Precision = \frac{TP}{TP + FP}, F1 = \frac{2 * Rc * Pr}{Rc + Pr} \tag{4}$$

5.1 Dataset

We have used the dataset of [7] to evaluate our method. This dataset is composed of captured traffic of eight applications, each executed for about 100 rounds and provided in pcap format. This set of applications comprises of Skype, VSee, TeamViewer, JoinMe, GIT, SVN, Psiphon, and Ultra, falling in four different traffic characterization categories: Chat, Remote Desktop Sharing, Version Control, and VPN. We have filtered out packets having useless information for traffic characterization and application identification tasks such as acknowledgment packets using the tshark tool[3]. To evaluate our framework, we first have randomly partitioned the dataset into three independent sets, naming train, validation, and test. These sets include 65%, 15%, and 20% of each application pcap files, respectively.

5.2 Implementation

We have fully implemented the framework with Python 3. After filtering out the unnecessary packets, 5-tuple which identifies each network flow along with timestamp and length are extracted for each packet of packet trace. Statistical features provided in Table 1 are extracted by Python Pandas libraries[4] in the *Network Unit Extraction* module (Section 4.1). Kmeans++ algorithm is used for the clustering purpose from the scikit-learn library[5] and StandardScaler function is used for standardizing the feature sets from the same package in the *Network Unit Clustering* module (Section 4.1). To automate identifying the number of clusters of each protocol, we have leveraged KneeLocator from the Kneed library[6]. For implementing the *Language Learner* module (Section 4.2) we benefited from k-TSS language learner part of [11] implementation and modified it based on our purpose. Finally, we have used the grid search method to choose the best value for the framework's hyper-parameters, including st, it, fd, $stats$, and k.

[3] https://www.wireshark.org/docs/man-pages/tshark.html
[4] https://pandas.pydata.org/
[5] https://scikit-learn.org/
[6] https://pypi.org/project/kneed/

Table 2: Framework Performance

Application	Performance Metrics		
	Pr	Rc	F1
Skype	100	100	100
VSee	100	100	100
TeamViewer	87	100	93
JoinMe	95	95	95
GIT	100	100	100
SVN	100	100	100
Psiphon	95	86	90
Ultra	100	95	97
Wtd. Average	97	97	97

(a) Application Identification

Traffic Characterization	Performance Metrics		
	Pr	Rc	F1
Remote Desktop Sharing	90	97	94
VPN	97	90	94
VersionControl	100	100	100
Chat	100	100	100
Wtd. Average	97	96.5	97

(b) Traffic Characterization

5.3 Classification Results

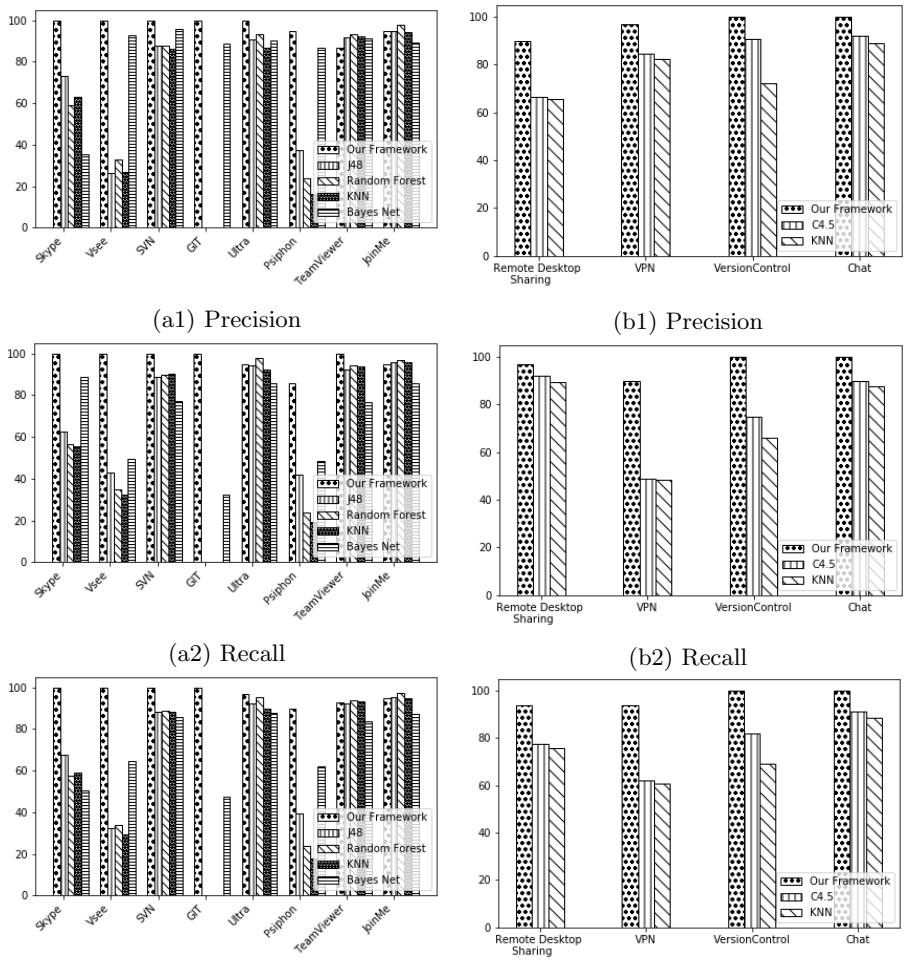
We have used a grid search to fine-tune the parameters st, fd, it, $stats$, and k, called hyper-parameters in machine learning. Based on the average F1-Measure, consequently, the best values for the application identification task are 15 Sec, 5 Sec, 10 Sec, 2, and 3 for the hyper-parameters st, fd, it, $stats$, and k, respectively, while for the traffic characterization task, they are 15 Sec, 15 Sec, 10 Sec, 2, and 3. Using the elbow method, the number of clusters for the most used protocol such as TCP is automatically set to 23 and for a less used protocol such as HTTP is set to six, resulting in 101 total number of clusters.

We have evaluated the framework on the test set with the chosen hyper-parameters. The framework has gained weighted average F1-Measure of 97% for both tasks. Table 2 provides the proposed framework performance in terms of Precision, Recall, and F1-Measure in detail.

5.4 Comparison with Other Methods

In this section, we provide a comparison of our framework with other flow-based methods used for network traffic classification. For the application identification task, we have compared our work with [2]. In this comparison, we first have extracted 44 most used statistical flow-based features from the dataset, as the 111 initial features are not publicly available. Then, we have converted these instances to arff format to be used as Weka[7] input. To have a fair comparison, we have used the ChiSquaredAttributeEval evaluator for feature selection and finally, applied all machine learning algorithms, including J48, Random Forest, k-NN (k = 1), and Bayes Net with 10 fold cross-validation used in this work. Fig. 4a compares the proposed framework performance with [2] in terms of Precision, Recall, and F1-Measure for application identification task. In

[7] http://www.cs.waikato.ac.nz/ml/weka/



(a1) Precision

(b1) Precision

(a2) Recall

(b2) Recall

(a3) F1-Measure

(b3) F1-Measure

(a) Application Identification

(b) Traffic Characterization

Fig. 4: Performance comparison of NeTLang and statistical classifiers.

most classes, our framework performed considerably better. Our framework gains much higher precision in identification of different applications in our dataset except for TeamViewer and JoinMe. In terms of recall, the work of [2] performed slightly better in detecting JoinMe and Ultra.

For the traffic characterization task, we made a comparison with the work of [1]. We have implemented a python code to extract time-related statistical flow-based features used by the authors. Using Weka, we have applied KNN and C4.5 in a 10-fold cross-validation setting which is the same as the one used in the paper. Fig. 4b compares the proposed framework performance with these

Table 3: Automata learning frameworks performance and time complexity comparison. (*: Multi-thread programming will reduce the time even more)

Application	Performance Metrics						Time Complexity			
	NeTLang			Sabahi [7]			NeTLang		Sabahi [7]	
	Pr	Rc	F1	Pr	Rc	F1	Train	Test	Train	Test
Skype	100	100	100	100	81	90				
VSee	100	100	100	100	95	97				
TeamViewer	87	100	93	100	93	94				
JoinMe	95	95	95	100	81	90				
GIT	100	100	100	100	99	99	<40* min	<milisec	<20 hr	<sec
SVN	100	100	100	100	99	99				
Psiphon	95	86	90	100	32	48.5				
Ultra	100	95	97	100	88	94				
Wtd. Average	97	97	97	100	83	89				

machine learning algorithms in terms of Precision, Recall, and F1-Measure for the traffic characterization task, showing that our work significantly outperforms the state-of-the-art work in application identification task.

As the proposed method in [7] is the only framework that has leveraged automata learning methods for application identification, we compare our work with it in terms of performance and time complexity (Table 3). Although the performance of [7] was computed in a two-class classification setting, NeTLang almost outperforms it in terms of Recall and F1-Measure. However, [7] has achieved higher overall precision because of its automata learning nature but owns time-consuming training and testing processes, while NeTLang considerably reduces the training time to the scale of minutes and it identifies application by observing only 4k of a network trace.

6 Related Work

Machine Learning based Methods in Traffic Classification. Many researchers applied the machine learning techniques in traffic classification. Time-related statistical flow-based features are used in [1] to characterize traffic in the presence/absence of a virtual private network (VPN). In this work, C4.5 decision tree and K-Nearest Neighbors are applied to the extracted features to classify flows. According to their result, the C4.5 algorithm performed better with the precision of 90% for traffic characterization.

Application identification from network flows is another traffic classification task performed in [2]. They used the CfsSubsetEval and ChiSquaredAttributeEval methods in Weka to optimize the number of features and supervised learning algorithms, including J48, Random Forest, k-NN, and Bayes Net. For the dataset they used, Random Forest and k-NN had better performance in terms of accuracy (90.87% and 93.94%, respectively).

Deep Packet [3] is a recently proposed packet-level deep-learning-based framework that automatically extracts features for traffic classification. They used

stacked autoencoders and one-dimensional CNN in their framework. This framework achieved a recall of 94% and 98% for traffic characterization and application identification tasks, respectively. The drawback of the deep-learning-based method is their time-consuming training process.

Automata Learning based Methods in Traffic Classification. The most related work to ours is [7] in terms of utilizing a passive automata learning technique to derive the behavioral packet-level network models of applications.They provided a multi-steps algorithm consists of building an initial automaton and generalizing it by a state merging condition based on the behavior of well-known network protocols. Their fine granularity leads to produce large models, which make some steps of generalizing fulfill slowly for large models. Furthermore, the detection of an application was constrained to observing a complete trace of an application. Our method rectifies these shortcomings by the idea of learning a k-TSS language, which was inspired by [11]. Botnet detection has been studied in [8]. They passively learned automata for clusters of the dataset to characterize a communication profile. Other related work mainly use the automata learning for protocol modeling such as [9, 10] by active automata learning, while using the desired protocol implementation as their oracle system and alphabet of the automaton derived manually by an expert.

7 Conclusion

In this paper, we have presented NeTLang, a framework which classifies network traffic by automatically extracting the network alphabet and learning the correspondent languages. To this aim, NeTLang combines an unsupervised learning algorithm and an automata learning technique to build on their strengths, to be fast and accurate and to eliminate their weaknesses, including ignoring the flow temporal relation. Moreover, we proposed a classifier by enhancing the acceptance condition of automata by machine learning with a new proximity metric. According to the experimental result, NeTLang has outperformed state-of-the-art methods used either machine learning or automata learning in traffic classification tasks by achieving the average F1-Measure of 97%. Furthermore, it has drastically decreased the learning time by approximately 96.5% in comparison with when using passive automata learning algorithm alone.

We plan to evaluate NeTLang using a public dataset. Additionally, to improve the approach performance, we will take advantage of protocols phases, naming initialization, data transmission, and finalization when extracting network units to define the alphabet more accurately.

Acknowledgments

The authors would like to thank Frits Vaandrager for his fruitful discussion on k-TSS languages and automata learning.

References

1. G. Draper-Gil, A. Habibi Lashkari, M. Saiful Islam Mamun, and A. A. Ghorbani, "Characterization of encrypted and vpn traffic using time-related features," in *ICISSP*, 2016.
2. B. Yamansavascilar, M. A. Guvensan, A. G. Yavuz, and M. E. Karsligil, "Application identification via network traffic classification," in *2017 International Conference on Computing, Networking and Communications (ICNC)*, Jan 2017, pp. 843–848.
3. M. Lotfollahi, M. X Jafari Siavoshani, R. Shirali Hossein Zade, and M. Saberian, "Deep packet: a novel approach for encrypted traffic classification using deep learning," *Soft Computing 2019*, pp. 1433–7479, 2019.
4. K. Xu, Z. Zhang, and S. Bhattacharyya, "Profiling internet backbone traffic: Behavior models and applications," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pp. 169–180, Aug 2005.
5. P. Bermolen, M. Mellia, M. Meo, D. Rossi, and S. Valenti, "Abacus: Accurate behavioral classification of p2p-tv traffic," *Computer Networks*, vol. 55, no. 6, pp. 1394–1411, 2011.
6. J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in Computer Virology*, vol. 7, no. 4, pp. 233–245, Nov 2011.
7. Z. Sabahi-Kaviani and F. Ghassemi, "Behavioral model identification and classification of multi-component systems," *Sci. Comput. Program.*, vol. 177, pp. 41–66, 2019.
8. C. Hammerschmidt, S. Marchal, R. State, and S. Verwer, "Behavioral clustering of non-stationary ip flow record data," in *2016 12th International Conference on Network and Service Management (CNSM)*. IEEE, 2016, pp. 297–301.
9. P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining model learning and model checking to analyze tcp implementations," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 454–471.
10. P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of ssh implementations," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, 2017, pp. 142–151.
11. A. Linard, C. de la Higuera, and F. Vaandrager, "Learning unions of k-testable languages," in *Language and Automata Theory and Applications*, C. Martín-Vide, A. Okhotin, and D. Shapira, Eds. Cham: Springer International Publishing, 2019, pp. 328–339.
12. R. McNaughton and S. A. Papert, *Counter-Free Automata (M.I.T. Research Monograph No. 65)*. The MIT Press, 1971.
13. P. Garcia, E. Vidal, and J. Oncina, "Learning locally testable languages in the strict sense." in *ALT*, 1990, pp. 325–338.
14. P. Garcia and E. Vidal, "Inference of k-testable languages in the strict sense and application to syntactic pattern recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 9, pp. 920–925, 1990.
15. T. Yokomori and S. Kobayashi, "Learning local languages and their application to dna sequence analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 10, pp. 1067–1079, Oct 1998.
16. C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.
17. R. L. Thorndike, "Who belongs in the family?" *Psychometrika*, pp. 267–276, 1953.