Mobile App Fingerprinting through Automata Learning and Machine Learning

Fatemeh Marzani*, Fatemeh Ghassemi*, Zeynab Sabahi-Kaviani*, Thijs van Ede†, and Maarten van Steen†
*University of Tehran,
Email: {marzani.f76, fghassemi, z.sabahi}@ut.ac.ir
†University of Twente,
Email: {t.s.vanede, m.r.vansteen}@utwente.nl

***Abstract*—Application fingerprinting is crucial in network management and security to provide the best Quality of Service (QoS). To generate fingerprints for applications, we use an automata learning algorithm to observe the temporal order among destination-related features of network traffic and create a language as a fingerprint. We label fingerprints through machine learning classifiers. We propose our approach in a framework called ML-NetLang for fingerprinting mobile applications from encrypted network traffic. Our evaluation achieves an average accuracy of 95% for Android and iOS applications. ML-NetLang outperforms comparable state-of-the-art techniques using behavioral-based, correlation-based, and machine-learning solutions.**

***Index Terms*—Fingerprinting, Traffic Classification, Automata Learning, Machine Learning**

I. Introduction

Identifying active applications on a (desktop or mobile) device is currently a central topic in computer science [1]. It is essential in network management, security, and intrusion detection to provide high Quality of Service (QoS) and prevent unusual behavior [2]. Several methods exist to identify active applications without installing intrusive monitoring agents on each device that observes network traffic. Traditional approaches are mainly based on port-based classification [3] and payload-based classification [4]. They are inefficient because of the wide usage of dynamic port assignment and encrypted traffic, respectively. Researchers apply machine learning, correlation-based, and behavioral classification to tackle these limitations. Machine Learning (ML) classification methods rely on statistical features of traffic on either a packet-level [5] or flow-level [6]. ML methods generally use too many redundant features [2]. Correlation-based classification methods focus on finding the correlation between flows. Although correlation-based methods avoid feature redundancy as observed in ML approaches, they have high computational complexity [2]. FLOWPRINT [7] is a correlation-based tool that finds temporal correlations among destination-related features and extracts maximal cliques from these correlations, using them as application fingerprints. Behavioral methods [8, 9] observe behavioral aspects in the traffic (such as IP address, used protocol, and port number) to identify active applications. Although behavioral methods are robust against encryption, their classification results are unsatisfactory [2]. We propose an approach combining the behavioral method based on automata learning and machine learning.

ISBN 978-3-903176-57-7

Our approach leverages the framework of NetLang [10] to generate and classify automata models of applications based on the destination features of network flows. Netlang uses automata learning techniques to derive behavioral models of each application's network traffic as k-Testable languages in the Strict Sense (k-TSS) [11]. K-TSS are a class of regular languages characterized by the sets of all prefixes and suffixes of length $k-1$ and substrings of length k appearing in the words of the language. It models applications as a formal language, where traffic traces produced by the application are interpreted as words whose letters consist of individual traffic flows. Using these observed words, it derives the formal language, which, in turn, is used to identify applications. In other words, it transforms the application identification problem by classifying traffic traces into an application based on a distance function between the application's language and the language derived from a single trace. The symbols of the languages are a set of related sequential packets that are mostly observed together in a flow. By using flows as the basis for symbols instead of packets, the approach is more robust to noise.

In NetLang, the set of alphabets are derived through a time-consuming pre-processing step by breaking up flows using timing and statistical features of packets. Instead, to speed up the approach and improve its accuracy, we modify and redesign NetLang's component for extracting automata alphabets. To extract alphabets of the learned languages, we consider destination-related features (such as IP and port number, or TLS certificate) as symbols of alphabets, inspired by FLOWPRINT. In this way, we consider the temporal order of communication flows as the fingerprint of an application, instead of the ordering of packets. This temporal ordering distinguishes our approach from FLOWPRINT which uses the static maximal clique among destinations. We also redesign the machine-learning models of NetLang's classification component with features that are compatible with the new symbols of the learned language. This redesign significantly improves detection accuracy and reduces training time (see Table II in Section IV-F1). Our approach is applicable to datasets containing only a single sample trace for each application, which means it can split the sample for training and testing.

We extract our classifier features based on the learned formal language model of applications.

Our experiments show that a well-designed combination of automata and machine learning, outperforms NetLang even though the symbols are selected in terms of minimal information of flows. We have implemented our approach in a framework called Machine Learning Network Traffic Language learner, ML-NetLang. We evaluate the performance of our approach using different public datasets under various conditions, and show that it outperforms correlation-based and statistics-based approaches. In summary, we make the following contributions:

- We leverage both behavioral and machine learning approaches by using k-TSS and ML algorithms in application identification.
- We implemented our method in a framework named ML-NetLang, which is robust in identifying encrypted network traffic.
- We show that our method identifies applications with an accuracy of 95% for both Android and iOS applications, while outperforming the state-of-the-art tools FLOWPRINT [7], AppScanner [12], and NetLang [10]. It also reduces time complexity by generating a single fingerprint for each application.

II. Background on Automata Learning

The aim of automata learning, also called grammar inference or regular inference, is to find the smallest automaton that accepts positive samples (in our case traces from a specific application) and rejects negative samples (i.e., traces from other applications) [13]. In our method, we benefit from algorithms for learning k-Testable Languages in the Strict Sense, called k-TSS [14]. The purpose of k-TSS is to find the smallest deterministic finite automata (DFA) by observing only positive samples. A k-TSS language is defined by a finite set of substrings, each with a length of k, that are allowed to appear in the words of the language [15].

If a finite set Σ of symbols $\{\alpha_1, \ldots, \alpha_n\}$ is fixed, then a word $w = \alpha_{i_1} \cdots \alpha_{i_k}$ is a finite sequence of symbols. A k-TSS language of a list of words is defined by all prefixes and suffixes of length $k-1$ of words and all substrings of words of length k [15]. The set of prefixes, suffixes, and substrings allowed to appear in words is listed in what McNaughton and Papert [14] called a **k-test vector**. The following definitions are taken from [14]:

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

- Σ is a finite alphabet,
- $I \subseteq \Sigma^{(k-1)}$ is a set of allowed prefixes of length $k-1$,
- $F \subseteq \Sigma^{(k-1)}$ is a set of allowed suffixes of length $k-1$,
- $T \subseteq \Sigma^k$ 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 L(Z) in the strict sense (k-TSS) is computed as:

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

In our problem, words are produced with a minimum length of k. This means that C always is empty. Hence, we can ignore C. A k-test vector of a language is constructed by scanning the accepted word through a sliding window of size k. For instance, if $w = abba$ and window size equals 3, then $\Sigma = \{a, b\}$, $I = \{ab\}$, $F = \{ba\}$, and $T = \{abb, bba\}$.

III. Methods

The goal of ML-NetLang is to fingerprint a mobile application using supervised methods based on the application's encrypted network traffic. FLOWPRINT [7] observed that a mobile application is generally composed of various modules that each communicate with a set of network destinations. We focus on finding distinctive communication orders in different mobile applications. Our fingerprints are based on (1) the temporal order among network flows of monitored devices and (2) the destinations these devices interact with. Exploiting the Netlang framework, Figure 1 shows the overview of the ML-NetLang methodology. We take network traffic as input and create fingerprints that map to applications, which are subsequently used for labelling new network traffic. We introduce a novel Trace Generator module that extracts features of traffic destinations from network traces, and then uses these features to convert traces into a list of words. Next, we feed these words into Netlang's Language Learner, which produces k-TSS languages. Finally, our classifier uses these learned languages to generate fingerprints, and then label network traffic with the help of machine-learning models and classifiers.

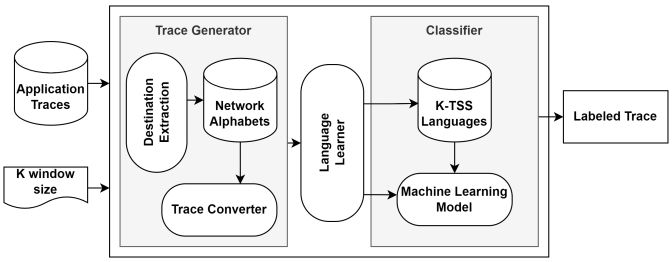


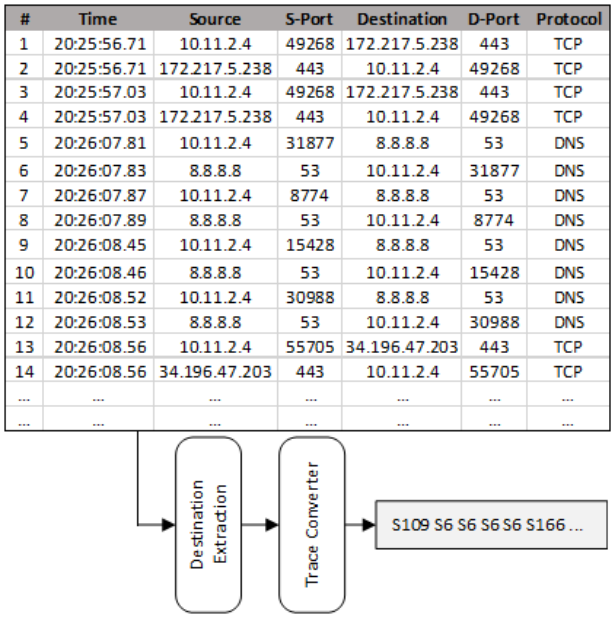
Fig. 1. Overview of ML-NetLang

A. Trace Generator

The trace generator converts the network traffic of each application into a list of words that are in the language of that application. Let Π be the input for the trace generator, containing network traffic of n applications $\{App_1, \ldots, App_n\}$. Assume that each application has m associated traces. Then the output is a set of word lists $S = \{W_1, W_2, W_3, \ldots, W_n\}$ where W_i is a list of words belonging to App_i. We remark that $W_i = [w_1, w_2, w_3, \ldots, w_m]$, where w_j is the word generated for the j^{th} trace in App_i. To generate a word for each trace *from a single application*, we proceed as follows:

1) We extract its network traffic flows.

2) For each flow, we extract the features timestamps, destination IP, destination port number, and TLS certificate[1].
3) Flows are sorted by the timestamp of their first packet.
4) Flows are categorized based on either tuple of destinations (IP, Port) or TLS certificates and given a symbol based on their category.
5) We define a mapping function to attach symbols to destination categories, concatenating "S" as the abbreviation of the symbols and a natural number j, which shows this destination is the j^{th} unique destination seen. For instance, $S6$ represents the 6^{th} destination seen among all applications traces by the trace generator. The symbols of categories of flows are used as the network alphabets.
6) Flows are replaced by network alphabets to create their corresponding symbolic traces. These traces represent the temporal order among the destination addresses of the various flows from a single application.
7) We can then create the list of words $W_i = [w_1, \ldots, w_m]$



#	Time	Source	S-Port	Destination	D-Port	Protocol
1	20:25:56.71	10.11.2.4	49268	172.217.5.238	443	TCP
2	20:25:56.71	172.217.5.238	443	10.11.2.4	49268	TCP
3	20:25:57.03	10.11.2.4	49268	172.217.5.238	443	TCP
4	20:25:57.03	172.217.5.238	443	10.11.2.4	49268	TCP
5	20:26:07.81	10.11.2.4	31877	8.8.8.8	53	DNS
6	20:26:07.83	8.8.8.8	53	10.11.2.4	31877	DNS
7	20:26:07.87	10.11.2.4	8774	8.8.8.8	53	DNS
8	20:26:07.89	8.8.8.8	53	10.11.2.4	8774	DNS
9	20:26:08.45	10.11.2.4	15428	8.8.8.8	53	DNS
10	20:26:08.46	8.8.8.8	53	10.11.2.4	15428	DNS
11	20:26:08.52	10.11.2.4	30988	8.8.8.8	53	DNS
12	20:26:08.53	8.8.8.8	53	10.11.2.4	30988	DNS
13	20:26:08.56	10.11.2.4	55705	34.196.47.203	443	TCP
14	20:26:08.56	34.196.47.203	443	10.11.2.4	55705	TCP
...	...	...	...	...	...	...
...	...	...	...	...	...	...

Fig. 2. Application of the Trace Generator module

To illustrate, consider Figure 2, showing a total of 14 packets. Since packets 1, 2, 3, and 4 are transmitted between the same source and destination in terms of IP, port, and protocol, they are grouped into one flow. Likewise, the other packets are grouped into flows leading to the trace $[(1,2,3,4),(5,6),(7,8),(9,10),(11,12),(13,14)]$ after which these flows are sorted by their first packet timestamp. Each flow is then replaced by the network alphabet based on its network destination. Thus, this trace is transformed into *S109 S6 S6 S6 S6 S166*. All flows $(5,6)$, $(7,8)$, $(9,10)$, and $(11,12)$ share the same destination address, so their alphabet is the same.

[1]For this paper, we assume that feature extraction is possible, implying, for example, that we exclude the situation of traffic flowing through a VPN tunnel, or redirected to a network proxy.

B. Language Learner

The network traffic of an application is converted into a list of words that are in the language of that application by the trace generator. The k-testable language is then derived by the language learner, taking the list of words as input. Our language learner comes from the one used in NetLang [10]. A language is learned by sliding a window of length k over an application's trace, thereby identifying the k-test vector $Z = \langle \Sigma, I, F, T \rangle$. In this work, we set $k = 3$, which is the recommended value in NetLang. A learned language is saved in the k-TSS languages database. For the application trace from Figure 2, the k-test vector is learned to be:

$$\begin{aligned} \text{application word} = \;& S109\ S6\ S6\ S6\ S6\ S166 \\ \Sigma = \;& \{S109,\ S6,\ S166\ \} \\ I = \;& \{S109\ S6\ \} \\ F = \;& \{S6\ S166\ \} \\ T = \;& \{S109\ S6\ S6,\ S6\ S6\ S6,\ S6\ S6\ S166\ \} \end{aligned}$$

C. Classifier

The language learner module learns the k-TSS language of an application App_i by its associated words W_i as $L(App_i) = L(W_i) = [L(w_1), \ldots, L(w_m)]$. To derive this language, we identify k-test vector for each word of $L(W_i)$. The k-test vector of $L(W_i)$, i.e., $Z(W_i)$, is represented by $[\langle \Sigma_1, I_1, F_1, T_1 \rangle, \ldots, \langle \Sigma_m, I_m, F_m, T_m \rangle]$. We compute the **union k-test vector** $UZ(App_i)$ of an application App_i as the union of learned vectors for its words:

$$UZ(App_i) = \langle \bigcup_{k=1}^{m} \Sigma_k, \bigcup_{k=1}^{m} I_k, \bigcup_{k=1}^{m} F_k, \bigcup_{k=1}^{m} T_k \rangle.$$

We use the **union k-test vector** of an application as its fingerprint as it gives a compact representation that considers all the sample words. The classifier aims to label given network traffic according to learned fingerprints. To do so, we use the trace generator to convert new traffic containing ℓ traces to a list of words $W = [w_1, \ldots, w_\ell]$. The classifier will then need to label each member of W with application names. To that end, for each $w \in W$, the language learner generates its k-test vector $Z(w) = \langle \Sigma, I, F, T \rangle$. Next, we train a machine-learning classifier that identifies each word class from features computed by our feature computation function. The feature computation function (FC) computes the proximity between the given k-test vector and the union k-test of all applications as the feature vector.

Definition 3. (Feature Computation Function): Given $Z(w) = \langle \Sigma, I, F, T \rangle$ and $UZ(App_i) = \langle \Sigma_i, I_i, F_i, T_i \rangle$, $FC(Z(w), UZ(App_1), \ldots, UZ(App_n)) = (f_1, \ldots, f_n)$, where f_i is the proximity between $Z(w)$ and $UZ(App_i)$:

$$f_i = (\Delta T_i, \Delta T_i', \Delta \Sigma_i, SIM\Sigma_i)$$

with

$$\Delta T_i = \frac{|T - T_i|}{|T|} \qquad \Delta T'_i = \frac{|T_i - T|}{|T_i|}$$
$$\Delta \Sigma_i = \frac{|\Sigma - \Sigma_i|}{|\Sigma|} \qquad SIM\Sigma_i = \frac{|\Sigma \cap \Sigma_i|}{|\Sigma \cup \Sigma_i|} \tag{1}$$

We compute the feature vector of training traces using the Feature Computation Function. If we have n applications, then the number of features computed for each trace equals $4 \times n$. We train the classifier model with the computed feature vectors and the names of the applications that are used as labels. To predict the application label for a given test trace w, we first compute its feature vector by applying the FC to its k-test vector and the union k-test vectors learned for all applications, i.e., $FC(Z(w), UZ(App_1), \ldots, UZ(App_n))$. Finally, the model predicts the label of a given trace by the name of an application.

IV. Evaluation

We implemented our methodology, named ML-NetLang[2] in Python 3 using PyShark[3] to read captured network traffic and extract destinations and timing features in the trace generator. We applied the k-TSS language learner implemented in NetLang [10] in the Language Learner module to learn the union k-test vectors of applications as fingerprints. Finally, we used Scikit-learn[4] to create the machine learning model used for classification. We evaluated the performance of our methods in application identification in terms of precision, recall, and F1-Score.

A. Dataset

We used two public encrypted network traffic datasets to evaluate our method under different conditions. These datasets have different properties, like being generated automatically or by user interaction, containing different versions of applications, installed on iOS and Android operating systems, and collected from different countries and application stores. As network traffic behavior relies on user interactions, the results of the synthetic datasets may differ from user-generated datasets. Furthermore, users tend to update applications with newer versions on average monthly [16], stressing the need for fingerprints that are robust against updates. ML-NetLang showed robust results within user-generated and synthetic data, iOS and Android, and identified applications precisely when applications were updated.

1) ReCon: ReCon Android App Versions [17, 18] stores multiple versions of 512 popular Android apps from Google Play Store, covering 7,665 app releases over eight years of app version history.The selection was made from the top 50 ranking apps from each category like games, social media, etc, among the 600 most popular free apps available on the Google Play Store. The network traffic was captured by automated and scripted interaction with apps on real mobile devices. This dataset enables us to evaluate our method against different versions of apps.

2) Cross Platform: Cross Platform [19] stores user-generated executions of overall 215 top Android apps from Google Play Store in the US and India, and Tencent MyApps and 360 Mobile Assistant stores in China. It also contains 196 iOS apps from the official Apple Store in the US, India, and China. The network traffic was captured by manual interaction for five minutes while testing all the main features of the applications. This dataset enables us to evaluate our method with user-generated network traffic and different mobile operating systems.

B. Application Identification

We evaluate our approach on 100 randomly chosen Android and iOS apps from the Cross-Platform dataset. We also randomly chose 100 applications with ten random versions of each from ReCon dataset. Some machine-learning methods randomly split network traffic flows into training and testing sets, but this technique does not preserve the temporal order among flows. As our method depends on this temporal order, we used partial-window splitting to preserve partial order among flows. First, we split network traffic into windows with a maximum size of k, which means each window contains k or fewer flows. Then we randomly split the windows of each application 50:50 into training and testing sets without any overlap. We assume the window size equals k in the language learner, which in our case is 3. Figure 3 shows the result of applying the partial-window splitting on the given network traffic in Figure 2 when the window size equals 3. The last window of network traffic may have fewer flows than the window size based on network traffic length. To evaluate our method, we created union k-test vectors of labeled training data. Then trained the model by computing features by applying Feature Computation Function in the k-TSS language and union k-test vectors of labeled training traffic. After training our model, we predicted the application name of the test data using the *distance function* proposed in NetLang and machine-learning algorithms Logistic Regression, SVM, Random Forest, Gradient Boosting, and Decision Tree. NetLang's *distance function* computes the distance between the test k-TSS language and trained k-TSS languages of applications, and labels the trace with the application name that has the minimum distance. Each classifier was evaluated by a 10-fold cross-validation.

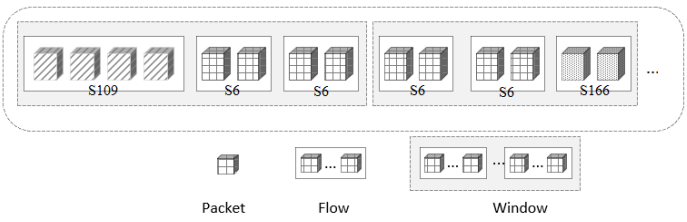


Fig. 3. Example of applying the partial-window splitting on the given network traffic in Figure 2. Window size equals 3. Different backgrounds of packets represent different flows, categorized based on the destination address.

[2]Available at: https://github.com/mlnetlang/
[3]https://pyshark.com/
[4]https://scikit-learn.org

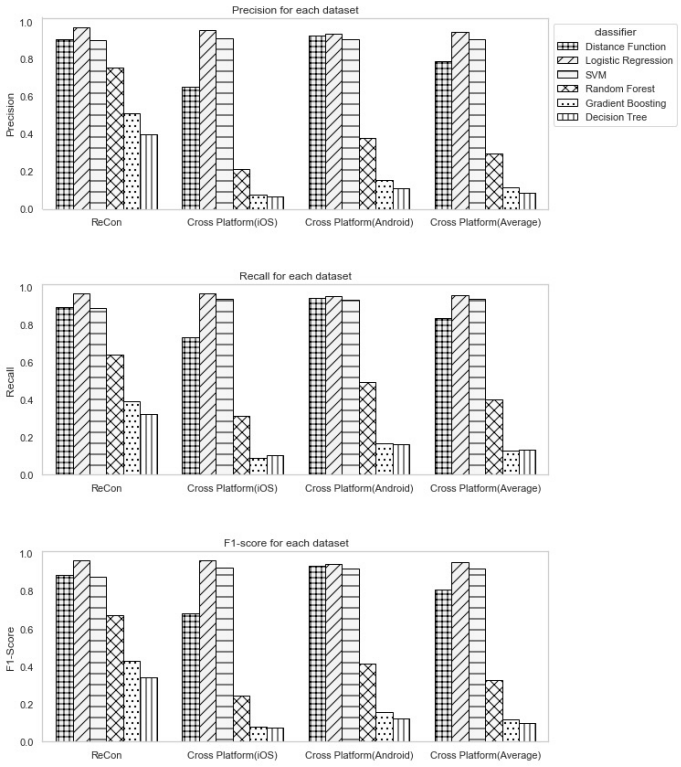


Fig. 4. Performance of application detection with different classifiers on each dataset.

Figure 4 shows the performance of the application detection using different classifiers on each dataset in terms of Precision, Recall, and F1-Score. Logistic Regression gives the best results in all datasets. Generally, SVM and *distance function* had slightly lower results than Logistic Regression. Random Forest, Gradient Boosting, and Decision Tree showed lower F1-Score compared to the other algorithms. More detailed performance results for Logistic Regression, SVM and the *distance function* can be found in Table I. The accuracy and recall levels are equivalent in our experiments, as a result of calculating the micro-average metrics for the individual apps. The superiority of LR and SVM can be explained as follows:

- Linear relationship: Linear Regression (LR) assumes a linear relationship between the input features and the target variable. Our results indicate that the relationship between the features and the application names in our data is approximately linear, so LR is able to capture this relationship well, resulting in good performance metrics such as precision, recall, and F1-score.
- Linear Separability: SVM is a binary classification algorithm that aims to find a hyperplane that can best separate the data points of different classes. Our results indicate that the features between the applications allow for a clear linear separation between the classes (i.e., the names of the applications used as class labels), So SVM performs well. SVM is particularly effective in cases where the data points are linearly separable.

C. Training Size

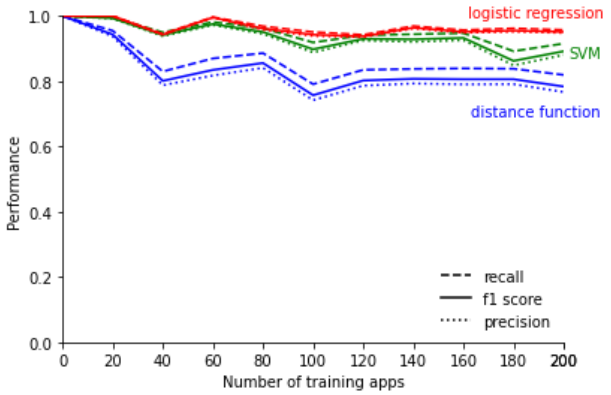


Fig. 5. Application identification performance vs training size (Cross-Platform)

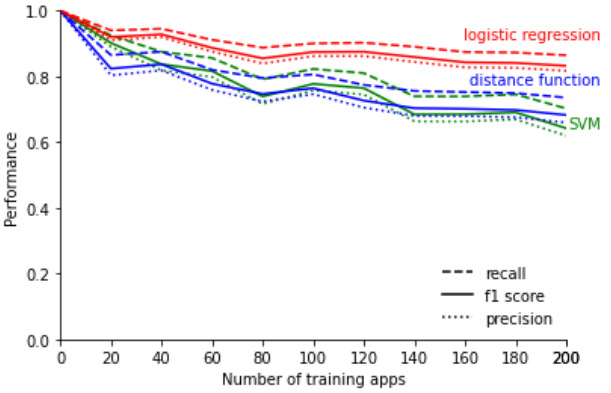


Fig. 6. Application identification performance vs training size (ReCon)

We have assumed that each device in the network has 100 applications installed. If this number is changed, ML-NetLang may perform better or worse because the number of training features has a direct and linear relationship with the number of training programs. To evaluate the impact of the number of installed applications, we train the model by varying the number of applications in the training data. Hence, we trained the model to evaluate application identification performance by randomly choosing N applications from 0 to 200 from the ReCon and Cross-Platform datasets. The three top algorithms, Logistic Regression, SVM, and *distance function*, are applied with a 10-fold cross-validation in each experiment. Figures 5 and 6 show the performance of ML-NetLang trained by different numbers of applications. As seen in both datasets, all metrics decline, but after a point, they become rather stable, especially for the Cross-Platform dataset. This shows the robustness of ML-NetLang against larger application sets. Furthermore, it is noticeable that Logistic Regression has the best performance compared to SVM and *distance function*. When 200 applications are installed, the F1-Score of Logistic Regression in the ReCon dataset is 83% and 95% on Cross Platform.

TABLE I
ML-NETLANG PERFORMANCE USING OF THREE BEST PERFORMING ALGORITHMS.

Dataset	Logistic Regression				SVM				Distance Function			
	Precision	Recall	F1-Score	Accuracy	Precision	Recall	F1-Score	Accuracy	Precision	Recall	F1-Score	Accuracy
Cross Platform (Android)	0.94	0.95	0.94	0.95	0.91	0.93	0.92	0.93	0.93	0.94	0.93	0.94
Cross Platform (iOS)	0.96	0.97	0.96	0.97	0.91	0.94	0.92	0.94	0.66	0.73	0.68	0.73
Cross Platform (Average)	0.95	0.96	0.95	0.96	0.91	0.94	0.92	0.94	0.79	0.84	0.81	0.84
ReCon	0.97	0.97	0.96	0.97	0.90	0.89	0.88	0.89	0.91	0.89	0.88	0.89

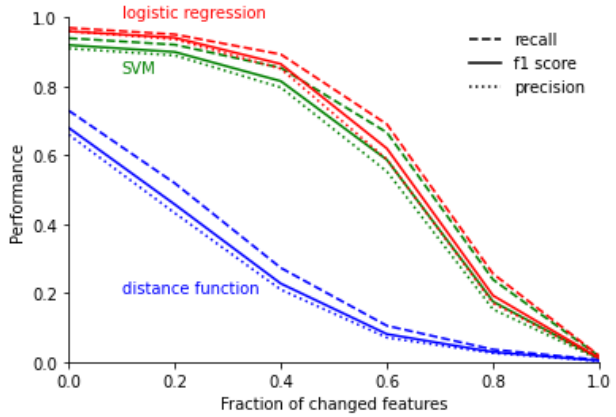


Fig. 7. Application identification performance vs changes in symbols of traffic trace (Cross-Platform-iOS). The x-axis demonstrates the % of changed features (in both IP and certificate features)

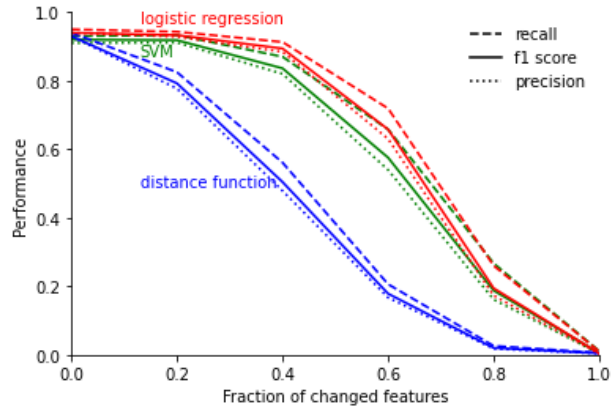


Fig. 8. Application identification performance vs changes in symbols of traffic trace (Cross-Platform-Android).

D. Mutated Traffic

So far, we have assumed that flows do not really change. However, in reality and because of the dynamic nature of network traffic, the destination-related features of applications may change over time, for instance, modifying (IP address, port) and TLS certificate. These changes can be the result of server migration or certificate renewals. To observe the effect of these changes on the ML-NetLang performance, we replaced a percentage of random symbols of generated traces in the trace generator module with random symbols (which we coin "noise"). We evaluated this experiment on 100 randomly chosen applications of the Cross-Platform dataset by applying Logistic Regression, SVM, and *distance function* using, again, a 10-fold cross-validation. Figures 7 and 8 show the results of ML-NetLang performance while the percentage of network destinations changed. As the figure shows, the application detection performance decreases mainly with the increase in the ratio of noise to traffic. However, in the case of 50% noise in the network traffic, Logistic Regression, and SVM still achieve a precision, recall and F1-Score of more than 80%, showing their robustness against high levels of noise. Increasing more than 50% of traffic changes leads to a steeper fall until all the symbols are changed when it cannot identify any application. In addition, it can be observed that Logistic Regression and SVM outperform the *distance function*.

E. Window size of k-TSS language

In the original NetLang paper [10], the optimal values for the window size k of k-TSS languages have been proposed as 3, 4, and 5, respectively, and therefore we have used $k = 3$ in all experiments. To observe the effect of the window size on ML-NetLang performance, we repeated the experiments detailed in Section IV-D with k equal to 3, 4, and 5, respectively. For this purpose, 100 random applications of the Cross-Platform iOS and Android datasets were chosen. We measured the experiment results using a 10-fold cross-validation and F1-Score metric to compare how different window sizes affect Logistic Regression, SVM, and *distance function*. Figures 9 and 10 show the results of ML-NetLang performance with different values of the window size while the percentage of network destinations changed in Cross-Platform datasets (iOS and Android). As the figures demonstrate, the decline in performance when using a larger window size is minimal for Logistic Regression and SVM. However, the performance of the distance function is notably affected by the choice of larger values for k and thus is less robust.

F. Comparison with other methods

We compare our work with the state-of-the-art tool NetLang which is the most related work to our approach. We also compare against the state-of-the-art methods FLOWPRINT [7] and AppScanner [12], which are using traffic correlation and machine-learning algorithms to identify applications, respectively.

1) Comparison with NetLang: NetLang and ML-NetLang both have three main modules: a trace generator, a language learner, and a classifier. We completely redesigned the trace generator and classifier and used NetLang's language learner.

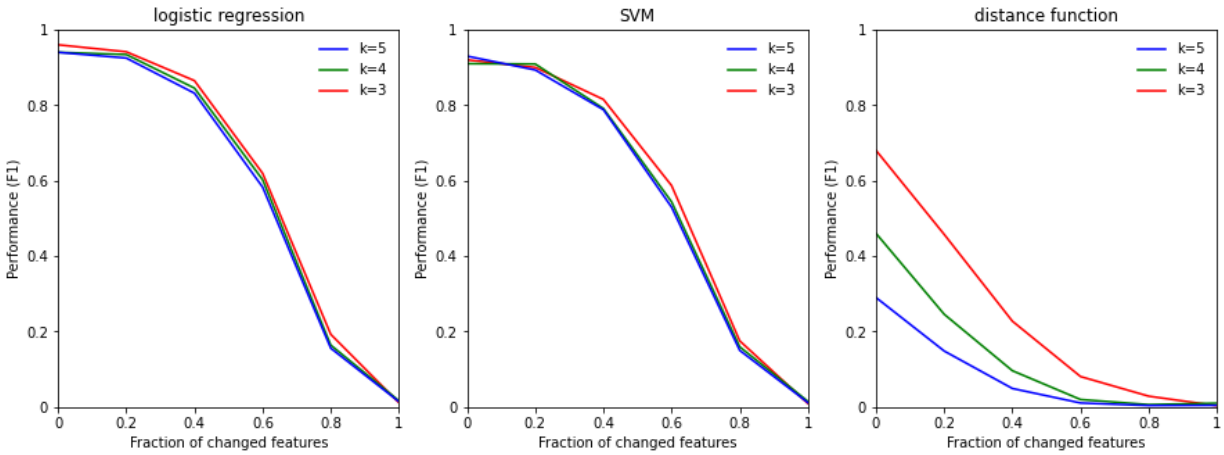


Fig. 9. Application identification performance vs changes in symbols of traffic trace(Cross-Platform-iOS) using different window sizes.

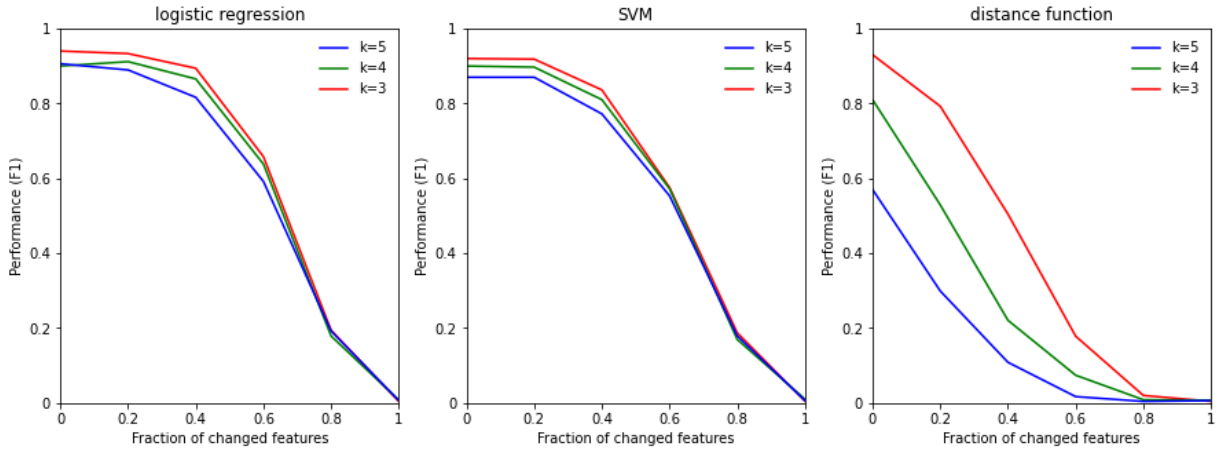


Fig. 10. Application identification performance vs changes in symbols of traffic trace(Cross-Platform-Android) using different window sizes.

Our method simplifies the trace generator by using destination-related features to extract the alphabets of learned automata (Section III-A), which is complicated and time-consuming in NetLang. NetLang splits network traffic into smaller units by different timing parameters, then categorizes units based on the highest layer protocol (observed in the corresponding packets), and finally, each category is clustered and labeled. We evaluate ML-NetLang against NetLang by running a 10-fold cross-validation on the same datasets and using partial window splitting to split the dataset into training and testing parts. In addition, NetLang requires timing parameters. We used the optimal values for these parameter values and window size of $k = 3$ suggested by the original paper. Table II compares our work and NetLang in terms of performance and time complexity. ML-NetLang remarkably outperforms and reduces the training time to the scale of seconds, which means it may be able to support online traffic classification, something NetLang can not. Our classification results are more accurate than NetLang due to machine-learning algorithms instead of NetLang's *distance function*. This *distance function* computes the distance of test k-TSS language and trained k-TSS languages of applications and classifies the traffic with the application name that has the minimum distance. One of the disadvantages of NetLang is that its timing parameters have to be tuned for each dataset. As our ML-NetLang approach does not include any timing parameters, it is independent of the underlying dataset. In addition, in the Sections IV-D and IV-E, we have shown that using Logistic Regression and SVM are both more robust than the *distance function* against changing destinations as well as the window size.

2) Comparison with FLOWPRINT and AppScanner: To compare fairly, we repeated the experiment setup mentioned in FLOWPRINT, choosing 100 applications randomly from the datasets and running a 10-fold cross-validation. Table III shows the performance of ML-NetLang, FLOWPRINT, and AppScanner in application identification. The accuracy and F1-Score are equal because of computing macro-average metrics for each application. Our method outperforms both FLOWPRINT and AppScanner in terms of recall and F1-Score. In other words, by focusing on the temporal order between flows rather than the various features used by FlowPrint and AppScanner, we observe a drop in false negatives. On

TABLE II
PERFORMANCE OF OUR FRAMEWORK COMPARED TO NETLANG IN APPLICATION IDENTIFICATION.

Dataset	ML-NetLang				NetLang			
	Precision	Recall	F1-Score	Accuracy	Precision	Recall	F1-Score	Accuracy
Cross Platform(Android)	0.94	0.95	0.94	0.95	0.26	0.36	0.29	0.36
Cross Platform(iOS)	0.96	0.97	0.96	0.97	0.20	0.27	0.22	0.27
Cross Platform(Average)	0.95	0.96	0.95	0.96	0.23	0.32	0.26	0.32
ReCon	0.97	0.97	0.96	0.97	0.29	0.28	0.26	0.30
Time Training	< sec				< 1hr			
Time Testing	< sec				< milisec			

TABLE III
PERFORMANCE OF OUR FRAMEWORK COMPARED TO FLOWPRINT AND APPSCANNER IN APPLICATION IDENTIFICATION.

Dataset	ML-NetLang(Logistic Regression)				FLOWPRINT(Jaccard Similarity)				AppScanner(Random Forest)			
	Precision	Recall	F1-Score	Accuracy	Precision	Recall	F1-Score	Accuracy	Precision	Recall	F1-Score	Accuracy
Cross Platform(Android)	0.94	0.95	0.94	0.95	0.90	0.87	0.87	0.87	0.91	0.89	0.87	0.89
Cross Platform(iOS)	0.96	0.97	0.96	0.97	0.94	0.93	0.93	0.93	0.85	0.15	0.24	0.15
Cross Platform(Average)	0.95	0.96	0.95	0.96	0.92	0.89	0.89	0.89	0.88	0.50	0.58	0.50
ReCon	0.97	0.97	0.96	0.97	0.95	0.94	0.95	0.94	0.90	0.43	0.58	0.43

the other hand, FLOWPRINT generates multiple fingerprints per app and uses the Jaccard similarity to label a fingerprint; its complexity order is $O(n^2)$. ML-NetLang reduces this complexity by generating a single union k-test vector as a fingerprint and uses Logistic Regression for classification. Having just a single fingerprint per application simplifies matters, although we still need to compare new traces to all fingerprints.

V. RELATED WORK

We categorize the related work into three classes based on the characteristics of their features used in classification, inspired by [2]. The statistics- based classification methods rely on statistical features extracted from the traffic. The correlation-based approaches depend on the correlation of flows in their classifications. The behavior-based methods consider the temporal order of packets or flows in communication with hosts.

A. Statistics-based methods

AppScanner [12] used statistical features of TCP flows to generate a model using Support Vector Machine and Random Forest algorithms. They achieved an accuracy of 96% on the 110 most popular applications in Google Play Store. Deep Packet [20] can identify encrypted traffic with granularity at the packet level. They used a stacked autoencoder and convolution neural network for traffic classification. Deep Packet achieved a recall of 98% on the ISCX dataset [21], a dataset of network applications with/without VPN traffic, using CNN as a classifier. In contrast to ML-NetLang, this method training phase is time-consuming.

B. Correlation-based methods

FLOWPRINT [7] is a tool to identify encrypted network traffic of mobile applications. They defined Adjusted Mutual Information (AMI) to rank statistical features of network traffic. They selected source/destination IP address, and TLS certificated features as the top ranks. The authors observed that mobile applications are composed of various modules that individually communicate with a set of network destinations. Based on this observation, FLOWPRINT clusters traffic flows based on destinations, finds temporal correlations among destinations, and extracts maximal cliques from these correlations, using them as application fingerprints. The classification accuracy of FLOWPRINT can reach 89.2%. Its granularity is at the flow level.

C. Behavior-based methods

NetLang [10] is the most similar framework to ours. It uses statistical features and timing to generate the alphabets of automata and then classifies the learned automata models with a *distance function*. However, the alphabet-generating method in NetLang is slow and cannot be used for online classification. Furthermore, this approach is not appropriate for mobile applications. BIND [22], uses statistical features of TCP streams to create application fingerprints. BIND also uses temporal features to observe application behavior. It gained an average accuracy of 92.6% on their own dataset collected for Android applications. It requires retraining the system periodically to avoid decreasing performance.

VI. CONCLUSION AND FUTURE WORK

We propose a framework, called ML-NetLang, for identifying applications from encrypted network traffic by combining automata learning and machine learning. The method is dataset independent, which means it is applicable to datasets with one traffic trace for each application and performs well on both synthetic (ReCon) and dynamic (Cross Platform) traffics. Furthermore, the results from the Cross-Platform dataset show

ML-NetLang can be used to identify iOS and Android applications. The evaluation indicates that ML-NetLang achieved an accuracy of 95%. The novelty of symbol generation based on destination addresses makes it more accurate and faster than other state-of-the-art techniques.

At the moment, our approach has shown to work when assuming that network traffic comes from just a single application. More challenging, obviously, is when network traffic comes from multiple, simultaneously operating applications. In that case, we may need to look at additional features to separate different behaviors, and thus applications. We see this as the next, logical step, to bringing our approach to more realistic settings.

REFERENCES

[1] M. Shafiq, X. Yu, A. A. Laghari, L. Yao, N. K. Karn, and F. Abdessamia, "Network traffic classification techniques and comparative analysis using machine learning algorithms," in *Proc. 2nd IEEE International Conference on Computer and Communications*, pp. 2451–2455, IEEE, 2016.

[2] J. Zhao, X. Jing, Z. Yan, and W. Pedrycz, "Network traffic classification for data fusion: A survey," *Information Fusion*, vol. 72, pp. 22–47, 2021.

[3] T. Karagiannis, A. Broido, M. Faloutsos, and K. C. Claffy, "Transport layer identification of p2p traffic," in *Proc. 4th ACM SIGCOMM conference on Internet measurement*, pp. 121–134, ACM, 2004.

[4] Y. Wang, Y. Xiang, and S. Yu, "Automatic application signature construction from unknown traffic," in *24th IEEE International Conference on Advanced Information Networking and Applications*, pp. 1115–1120, 2010.

[5] M. Lotfollahi, M. J. Siavoshani, R. S. H. Zade, and M. Saberian, "Deep packet: a novel approach for encrypted traffic classification using deep learning," *Soft Computing*, vol. 24, no. 3, pp. 1999–2012, 2020.

[6] B. Yamansavascilar, M. A. Güvensan, A. G. Yavuz, and M. E. Karsligil, "Application identification via network traffic classification," in *Proc. International Conference on Computing, Networking and Communications*, pp. 843–848, IEEE, 2017.

[7] T. van Ede, R. Bortolameotti, A. Continella, J. Ren, D. J. Dubois, M. Lindorfer, D. R. Choffnes, M. van Steen, and A. Peter, "Flowprint: Semi-supervised mobile-app fingerprinting on encrypted network traffic," in *Proc. 27th Annual Network and Distributed System Security Symposium*, vol. 27, The Internet Society, 2020.

[8] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in Computer Virology*, vol. 7, no. 4, pp. 233–245, 2011.

[9] L. Grimaudo, M. Mellia, and E. Baralis, "Hierarchical learning for fine grained internet traffic classification," in *8th International Wireless Communications and Mobile Computing Conference*, pp. 463–468, 2012.

[10] Z. Sabahi-Kaviani, F. Ghassemi, and Z. Alimadadi, "Combining machine and automata learning for network traffic classification," in *Proc. 3rd IFIP WG 1.8 International Conference on Topics in Theoretical Computer Science*, pp. 17–31, Springer, 2020.

[11] P. García, E. Vidal, and J. Oncina, "Learning locally testable languages in the strict sense," in *Proc. the 1st International Workshop on Algorithmic Learning Theory*, pp. 325–338, 1990.

[12] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Robust smartphone app identification via encrypted network traffic analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 1, pp. 63–78, 2018.

[13] Z. Sabahi-Kaviani and F. Ghassemi, "Behavioral model identification and classification of multi-component systems," *Science of Computer Programming*, vol. 177, pp. 41–66, 2019.

[14] R. McNaughton and S. A. Papert, *Counter-free automata*. The MIT Press, 1971.

[15] A. Linard, C. de la Higuera, and F. W. Vaandrager, "Learning unions of k-testable languages," in *Proc. 13th International Conference on Language and Automata Theory and Applications*, pp. 328–339, Springer, 2019.

[16] S. Comino, F. M. Manenti, and F. Mariuzzo, "Updates management in mobile applications: itunes versus google play," *Journal of Economics & Management Strategy*, vol. 28, no. 3, pp. 392–419, 2019.

[17] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. R. Choffnes, "Demo: Recon: Revealing and controlling PII leaks in mobile network traffic," in *Proc. the 14th Annual International Conference on Mobile Systems, Applications, and Services Companion*, p. 117, 2016.

[18] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. R. Choffnes, and N. Vallina-Rodriguez, "Bug fixes, improvements, ... and privacy leaks - A longitudinal study of PII leaks across android app versions," in *Proc. 25th Annual Network and Distributed System Security Symposium*, 2018.

[19] D. J. D. Jingjing Ren David Choffnes, "An International View of Privacy Risks for Mobile Apps," *Technical Report*, 2019.

[20] M. Lotfollahi, M. J. Siavoshani, R. S. H. Zade, and M. Saberian, "Deep packet: a novel approach for encrypted traffic classification using deep learning," *Soft Comput.*, vol. 24, no. 3, pp. 1999–2012, 2020.

[21] 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 *Proc. of ICISSP*, 2016.

[22] K. Al-Naami, S. Chandra, A. M. Mustafa, L. Khan, Z. Lin, K. W. Hamlen, and B. Thuraisingham, "Adaptive encrypted traffic fingerprinting with bi-directional dependence," in *Proc. the 32nd Annual Conference on Computer Security Applications*, pp. 177–188, 2016.