

# Predicting Domain Generation Algorithms with Long Short-Term Memory Networks

Jonathan Woodbridge, Hyrum S. Anderson, Anjum Ahuja, and Daniel Grant

{jwoodbridge,hyrum,aahuja,dgrant}@endgame.com

Endgame, Inc.

Arlington, VA 22201

*Abstract*—Various families of malware use domain generation algorithms (DGAs) to generate a large number of pseudo-random domain names to connect to a command and control (C2) server. In order to block DGA C2 traffic, security organizations must first discover the algorithm by reverse engineering malware samples, then generate a list of domains for a given seed. The domains are then either preregistered, sink-holed or published in a DNS blacklist. This process is not only tedious, but can be readily circumvented by malware authors. An alternative approach to stop malware from using DGAs is to intercept DNS queries on a network and predict whether domains are DGA generated. Much of the previous work in DGA detection is based on finding groupings of like domains and using their statistical properties to determine if they are DGA generated. However, these techniques are run over large time windows and cannot be used for real-time detection and prevention. In addition, many of these techniques also use contextual information such as passive DNS and aggregations of all NXDomains throughout a network. Such requirements are not only costly to integrate, they may not be possible due to real-world constraints of many systems (such as endpoint detection). An alternative to these systems is a much harder problem: detect DGA generation on a per domain basis with no information except for the domain name. Previous work to solve this harder problem exhibits poor performance and many of these systems rely heavily on manual creation of features; a time consuming process that can easily be circumvented by malware authors. This paper presents a DGA classifier that leverages long short-term memory (LSTM) networks for real-time prediction of DGAs without the need for contextual information or manually created features. In addition, the presented technique can accurately perform multiclass classification giving the ability to attribute a DGA generated domain to a specific malware family. The technique is extremely easy to implement using open source tools allowing the technique to be deployed in almost any setting. Results are significantly better than all state-of-the-art techniques, providing 0.9993 area under the receiver operating characteristic curve for binary classification and a micro-averaged F1 score of 0.9906. In other terms, the LSTM technique can provide a 90% detection rate with a 1:10000 false positive (FP) rate—a twenty times FP improvement over the next best method. Experiments in this paper are run on open datasets and code snippets are provided to reproduce the results.

## I. INTRODUCTION

Many malware families contain domain generation algorithms (DGAs) to make preemptive defenses difficult. Domains are generated pseudo-randomly in bulk (hundreds to tens-of-thousands per day) by a malware sample. The malware then attempts to connect to all or a portion of these generated domains in hopes of finding a command and control (C2)

server from which it can update, upload gathered intelligence, or pursue other malicious activities. The malicious actor only needs to register a small number of these domains to be successful. However, all the domains must be sinkholed, registered, or blacklisted before they go into use in order to preemptively defeat such an attack. This defense becomes increasingly difficult as the rate of dynamically generated domains increases.

Authors in [1] presented a thorough review of the efficacy of blacklists. As a part of this review, authors analyzed both public and private blacklists for DGA coverage, (i.e., how many domains generated by DGAs were contained in blacklists). Public blacklists were surprisingly lacking in terms of DGA coverage with less than 1.2% of DGAs analyzed by the authors being contained in any of the blacklists. Vendor provided blacklists fared better, but had mixed results over malware families with coverage varying from 0% to 99.5%. These results suggest that blacklists are useful, but must be supplemented by other techniques to provide a more adequate level of protection.

Another approach to combating malware using DGAs is to build a DGA classifier. This classifier can live in the network sniffing out DNS requests and looking for DGAs. When DGAs are detected, the classifier notifies other automated tools or network administrators to further investigate the origin of a DGA. Previous work in DGA detection can be broken down into two categories: retrospective detection and real-time detection. Retrospective detection makes bulk predictions on large sets of domains and are designed as a reactionary system that cannot be used for real-time detection and prevention [2], [3], [4]. In these systems, sets of domains are broken down into groupings using clustering with the intent to generate statistical properties of each grouping. Classification is accomplished by generating templates during training and using statistical tests (e.g., Kullback-Leibler divergence) to classify groups of potential DGAs. In addition, these techniques incorporate contextual information such as HTTP headers, NXDomains across a network, and passive DNS to further improve performance. Much of the previous work in DGA detection falls in the former category and, unfortunately, does not meet the needs of many real-world security applications that require real-time detection and prevention [5]. In addition, it is often unrealistic for many security applications to use contextual information. For example, endpoint detection and response (EDR) systems run on endpoints and hosts and have strict performance requirements on processing, network, and mem-

ory usage. Aggregating such contextual information from the network to each endpoint requires far too much overhead and is not practical for a real-world deployment.

Real-time detection techniques attempts to classify domains as DGA generated on a per domain basis using only the domains' names (i.e., no additional contextual information). Real-time detection is a considerably harder problem than retrospective techniques and techniques often exhibit performance far too low for a real-world deployment. (Suprisingly, authors in [5] found that retropective techniques had similarly bad performance!) Many of the previous real-time approaches use hand picked features (e.g., entropy, string length, vowel to consonant ratio, etc.) that are fed into a machine learning model, such as a random forest classifier. Using hand-crafted features have two major drawbacks. First, hand-crafted features are easy to circumvent. Second, deriving hand-crafted features is a time consuming process. If, and when, a malicious actor derives a new DGA family around beating a set of features, security professionals will need to spend considerable time creating new features. To the best of our knowledge, authors in [2] presented the first (and only until this paper) featureless real-time technique by using Hidden Markov Models (HMMs). However, as shown later in the paper, HMMs perform quite poorly on detecting DGAs. To note, the HMMs in [2] were part of a much larger retrospective detection system.

This paper presents a feature-less real-time technique using Long Short-Term Memory networks (LSTMs) to classify DGAs. This technique has four significant advantages over other techniques in the literature. First, the LSTM DGA classifier is featureless, in that it operates on raw domain names (e.g., google.com, facebook.com, etc.). If a new family of DGA appears, then the classifier can be retrained without the tedious step of hand picking features. LSTMs work largely as a black box making it very difficult for adversaries to reverse engineer and beat a classifier without the same training set. Second, the presented technique has a significantly better true positive rate/false positive rate over previously published retrospective and real-time approaches. Third, the technique also works in a multiclass classification setting. Therefore, the algorithm not only provides a binary decision of whether a domain is DGA or not, but can accurately fingerprint a unique DGA's structure. Fourth, the presented algorithm can classify in real-time using absolutely no contextual information. Classification of a domain takes 20 ms on commodity hardware.<sup>1</sup> The technique is trivial to implement and can run on virtually any security environment. In fact, all the code required to implement this system is provided in this paper demonstrating its ease of deployment.

In this paper, we make the following contributions. We

- 1) introduce an LSTM network to predict DGA generated domains, which to our knowledge, is the first application and in-depth analysis of deep learning to this domain;
- 2) present complete experimental results showing significant improvements over previous techniques (both real-time and retrospective) in the literature using open datasets; and
- 3) provide source code to reproduce results.

To allow for easily reproducible results, Python source code built on the open source framework Keras [6] is provided. Experiments were run on GPU hardware, but it's possible to run all experiments on commodity desktop or laptop hardware. An overview of LSTMs and previous work is discussed in Section II. Details of reproducing the results are given in Sections III and IV. Full results are given in Section V with suggestions for future work in Section VI.

## II. BACKGROUND

Domain fluxing is a technique used by botnets and command-and-control (C2) servers to create many domains using a Domain Generation Algorithm (DGA) [7], [8]. All botnets and C2 servers in the same infrastructure use the same seeded algorithm such that they all create the same pseudorandomly generated domains. A subset of these domains are registered by the C2 servers while each botnet iterates through the DGA generated domains until it finds one that is registered. To further complicate the process, C2 servers continually switch to new DGA generated domains making blacklist creation and take down efforts difficult.

One approach to combating domain fluxing is to reverse engineer a piece of malware and its respective DGA [8]. Once a DGA and its respective seed is known, future domains can be registered and used as an impostor C2 server to hijack botnets (a process known as sinkholing). Once a campaign has been hijacked, adversaries must redeploy new botnets with updated seeds to continue.

Blacklisting is another approach to combat domain fluxing [1]. DGA generated domains are added to a blacklist that can be used by a network administrator to block connections to potential C2 servers. However, both blacklists and sinkholing are only effective when both the algorithm and seed used by a campaign is known.

### A. Domain Generation Algorithms

This paper evaluates the ability to classify DGA generated domains from 30 different types of malware. Malware families include ransomware, such as Cryptolocker [9], [10] and Cryptowall [11], banking trojans, such as Hesperbot [12], and general information-stealing tactics, such as ramnit [13].

DGA techniques vary in complexity from simple uniformly generated domain names to those that attempt to model distributions that are seen in real domains. ramnit, for example, creates domains with a series of divides, multiplies and modulus computed on a seed [13] while supobox creates domains by concatenating two random strings (typically taken from the English language) [14].

Predicting DGA generated domains from such algorithms as supobox is extremely difficult without using contextual information. In fact, the LSTM technique presented in this paper was the only real-time technique able to classify such domains.

### B. DGA Classification

DGA classification can be a useful component of a domain reputation system. Domain reputation systems have the task of

<sup>1</sup>Apple MacBook Pro with a 2.2 GHz Intel Core i7 and 16GB of memory

assigning a trustworthy score of a domain. This score typically varies from 0 (most benign) to 1 (most malicious). Domain reputation systems typically incorporate many pieces of heterogeneous data, such as passive DNS (pDNS), to make decisions on a domain's reputation [15], [16], [17]. DGA classification is one piece of information that can help assign a reputation to a domain. Previous approaches to DGA classification can be roughly broken down into two categories:

- 1) *Retrospective*: classifying domains in groups to take advantage of bulk statistical properties or common contextual information; and
- 2) *Real-time*: classifying domains individually with no additional contextual information.

Authors in [3], [4] detect DGAs by using both unigram and bigram statistics of domain clusters. The training set is separated into two subsets: those generated by a DGA and those not generated by a DGA. The distributions of both unigrams and bigrams are calculated for both the subsets. Classification occurs in batches. Each batch of unknown domains is clustered by shared second level domain and domains sharing the same IP address. The unigram and bigram distributions are calculated for each cluster and compared to the two known (labeled) subsets using the Kullback-Leibler (KL) distance. In addition, the authors use the Jaccard distance to compare bigrams between clusters and the known (labeled) sets as well.

Authors in [2] apply a similar clustering process to classify domains with unsuccessful DNS resolutions. To train, statistical features are calculated for each subset of labeled DGA generated domains, such as Bobax, Torpig, and Conficker.C. Unknown domains are clustered by statistical characteristics such as length, entropy, and character frequency distribution, as well as shared hosts requesting the domain (i.e., cluster two domains together if the same host made a DNS query for both domains). Next, statistical features are calculated for each cluster and compared to the training subsets to classify the clusters as formed by a known DGA. If a cluster is classified as belonging to a known DGA, the host is deemed to be infected.

Once a host is deemed to be infected with a DGA-bot, the authors attempt to identify the bots active C2 server. This stage of the process uses a Hidden Markov Model trained on each known family of DGA and applied to single domains (i.e., this technique follows the same assumptions as the LSTM technique proposed by this paper). Each domain with a successful DNS request is fed through each HMM. If a domain receives an adequate score (i.e., greater than some threshold  $\theta$ ), the domain is labeled as a DGA. The threshold is learned at training time and set to a maximum false positive rate of 1%. We use this HMM technique as one of our comparisons to previous work.

The aforementioned techniques (with exception to the HMM technique in [2]) are accomplished retrospectively. Authors in [5] perform an in-depth comparison of these techniques and discuss two important findings. First, retrospective techniques are too slow for most real-world deployments and often take hours to detect malicious domains. Second, the performance of these systems are quite poor in terms of false positives and true positives. These authors present their own technique that overlaps both retrospective and real-time

techniques. They apply an online form of sequential hypothesis testing to NXDomains only. Clients in a network are given an evolving score based on the number and maliciousness of NX-Domains. A client can be labeled as malicious or benign once its score goes above or below predefined thresholds. While this system is a big improvement over retrospective systems, it has three main drawbacks. First, detection is not always in real-time as a client takes time to build an appropriate score. Authors reported that only 83% of domains were detected in time to prevent a connection. Second, performance of their system is considerably less than most real-time solutions as we show in section V. Third, their system cannot perform multiclass classification as their system bases classification solely on the presence of NXDomains.

Authors in [18] present a real-time DGA classifier that uses two basic linguistic features named *meaningful characters ratio* and *n-gram normality score*. The *meaningful characters ratio* calculates the ratio of characters in a domain that comprise of a meaningful word. For example, *facebook* has a ratio of 1 as all character in the domain are covered by the words *face* and *book* while *face1234* has a ratio of 0.5 as only half of its character are covered by the word *face*. The *n-gram normality score* is calculated by finding n-grams with  $n \in 1, 2, 3$  within a domain and calculating their count in the English language. The mean and covariance of these four features are calculated from a benign set (*Alexa* top 100,000). Unknown domains are then classified by their Mahalanobis distance to the benign set (i.e. a larger distance is indicative of a DGA generated domain).

The approach in [18] is used as a filter step. Once domains have been classified as a DGA they are fed to a clustering technique (similar to those described above) to further classify the domains.

Section V shows a comparison of our technique to both retrospective and real-time systems. Our technique significantly outperforms retrospective techniques and the comparison is brief and compares findings to those in [5]. An in depth comparison is performed between our technique and the aforementioned real-time systems. More specifically, we compare our technique to the HMM defined by [2] as well as a Random Forest Classifier trained on features defined in [2], [3], [4], [18]. We do not perform an in depth comparison on the full systems as defined in [2], [3], [4] as they are retrospective systems and have already been shown to perform far worse than our system [5].

### C. LSTM Networks

In a variety of natural language tasks, recurrent neural networks (RNNs) have been used to capture meaningful temporal relationships among tokens in a sequence [19], [20], [21], [22]. The key benefit of RNNs is that they incorporate contextual (state) information in their mapping from input to output. That is, the output of a single RNN cell is a function of the input layer and previous RNN activations. Due to long chains of operations that are introduced by including self-recurrent connections, the output of a traditional RNN may decay exponentially (or, more rarely but catastrophically explode) for a given input, leading to the well-known *vanishing gradients* problem. This makes learning long-term dependencies in an RNN difficult to achieve.

The problem of vanishing gradients is a key motivation behind the application of the Long Short-Term Memory (LSTM) cell [23], [24], [25], which consists of a state that can be read, written or reset via a set of programmable gates. The cell’s state has a self-recurrent connection that allows the cell to exactly retain state between time steps. However, that state may be modulated by a new input via an input gate, which effectively multiplies the input by a number that ranges between 0 and 1 (sigmoid activation) or -1 and 1 (tanh activation). Likewise, a forget gate modulates the self-recurrent state connection by a number between 0 and 1. Thus, if the input gate modulates the input with 0, and the forget gate modulates the recurrent connection with 1, the cell ignores the input and perfectly retains state. On the other hand, a 1 (input) and a 0 (forget) causes the cell’s state to be overwritten by the input. And in the case of a 0 (input) and 0 (forget), the state is reset to 0. Finally, an output gate modulates the contribution of the cell’s state to the output, which propagates to the input gates of LSTM cells across the layer, as well as to subsequent layers of the network.

The LSTM cell’s design with multiplicative gates allows a network to store and access state over long sequences, thereby mitigating the vanishing gradients problem. For our use with domain names, the state space is intended to capture combinations of letters that are important to discriminating DGA domains from non-DGA domains. This flexible architecture generalizes manual feature extraction via bigrams, for example, but instead learns dependencies of one or multiple characters, whether in succession or with arbitrary separation.

### III. METHOD

We employ an LSTM network for detecting DGAs. The model has the following advantages:

- the model accepts variable-length character sequences as input, so that there is no auxiliary requirement for feature extraction<sup>2</sup>;
- the model is very compact, comprised simply of an embedding layer, an LSTM network layer, and a fully connected output layer that is simple logistic (or for multiclass, multinomial logistic) regression; and
- although training on a large dataset is computationally intensive, the shallow structure allows for very fast query times.

A graphical depiction of our model is shown in Fig. 1. To prevent overfitting when training neural networks, it is common practice to employ dropout. Dropout consists of randomly removing a random subset of edges between layers of a network during each iteration of training, but restoring their contribution at test time. We apply dropout after the LSTM layer prior to logistic regression.

The embedding layer projects  $\ell$ -length sequences of input characters from the input domain  $\mathcal{S} \subset \mathcal{Z}^\ell$  to a sequence of vectors  $\mathcal{R}^{d \times \ell}$ , where  $\ell$  is an upper bounded length determined from the training set. The input domain consists of non-redundant valid domain name characters (lowercase alphanumeric, period, dash and underscore), and the output dimension

<sup>2</sup>In experiments, we employ a trivial pre-processing step to remove top-level domains and convert all characters to lowercase.

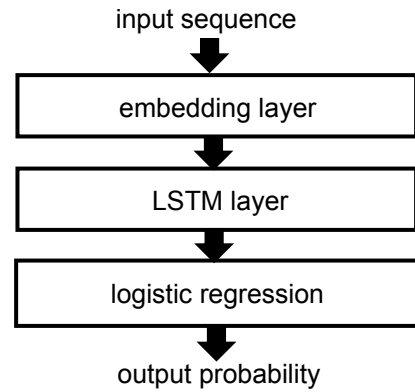


Fig. 1: Our model consists of an embedding layer, an LSTM layer that serves essentially as a feature extractor, and a logistic regression classifier.

$d$  is a tunable parameter that represents an embedding. In our model, we choose  $d = 128 > |\mathcal{S}|$  to provide additional degrees of freedom to the model, but preliminary experiments showed that results are relatively insensitive to the particular choice of  $d$ .

The LSTM layer can be thought of as implicit feature extraction, as opposed to explicit feature extraction (e.g.,  $n$ -grams) used in other approaches. Rather than represent domain names explicitly as a bag of bigrams, for example, the LSTM learns patterns of characters (or in our case, embedded vectors) that maximize the performance of the second classification layer. In our experiments we compare the LSTM model to an explicit bigram logistic regression model.

All LSTM code was written in Python using the Keras framework [6]. Two models are generated: one for a binary classification and one for a multiclass classification. Code for the binary classification is shown in Fig. 2 and the multiclass classification in Fig. 3.

The two code examples have a few small differences. The final dense layer goes from an output of one value in the binary classifier (line 15) to `nb_classes` in the multiclass classifier (line 17). A binary decision only requires a single value from  $[0, 1]$  where 0 is the most benign and 1 is the most DGA. The multiclass model produces `nb_classes` scores, one for each family known by the classifier, where multinomial logistic regression is employed on *softmaxed* activations on line 18 to encode a distribution that sums to unity.

### IV. EXPERIMENTAL SETUP

In the following section, we describe details of our experimental setup in evaluating DGA classifiers in a binary experiment (DGA vs. non-DGA) and multiclass experiment (which DGA?) using publically available domain names and DGA data.

#### A. Evaluation Metrics

Precision, Recall,  $F_1$  score, and Receiver Operating Characteristic (ROC) are the four evaluation metrics used to compare the LSTM classification technique to other state-of-the-art techniques. Precision is defined as

```

1 from keras.preprocessing import pad_sequences
2 from keras.models import Sequential
3 from keras.layers.core import Dense
4 from keras.layers.core import Dropout
5 from keras.layers.core import Activation
6 from keras.layers.embeddings import Embedding
7 from keras.layers.recurrent import LSTM
8
9 model=Sequential()
10 model.add(Embedding(max_features,
11                    128,
12                    input_length=75))
13 model.add(LSTM(128))
14 model.add(Dropout(0.5))
15 model.add(Dense(1))
16 model.add(Activation('sigmoid'))
17
18 model.compile(loss='binary_crossentropy',
19              optimizer='rmsprop')
20
21 # Pad sequence where sequences are case
22 # insensitive characters encoded to
23 # integers from 0 to number of valid
24 # characters
25 X_train=sequence.pad_sequences(X_train,
26                               maxlen=75)
27
28 # Train where y_train is 0-1
29 model.fit(X_train, y_train,
30          batch_size=batch_size, nb_epoch=1)

```

Fig. 2: Binary LSTM Code

```

1 from keras.preprocessing import pad_sequences
2 from keras.models import Sequential
3 from keras.layers.core import Dense
4 from keras.layers.core import Dropout
5 from keras.layers.core import Activation
6 from keras.layers.embeddings import Embedding
7 from keras.layers.recurrent import LSTM
8
9 model=Sequential()
10 model.add(Embedding(max_features,
11                    128,
12                    input_length=75))
13 model.add(LSTM(128))
14 model.add(Dropout(0.5))
15 # nb_classes is the number of classes in
16 # the training set
17 model.add(Dense(nb_classes))
18 model.add(Activation('softmax'))
19
20 model.compile(loss='categorical_crossentropy',
21              optimizer='rmsprop')
22
23 # Pad sequence where sequences are case
24 # insensitive characters encoded to
25 # integers from 0 to number of valid
26 # characters
27 X_train=sequence.pad_sequences(X_train,
28                               maxlen=75)
29
30 # Train where y_train is one-hot encoded for
31 # each class
32 model.fit(X_train, y_train,
33          batch_size=batch_size, nb_epoch=1)

```

Fig. 3: Multiclass LSTM Code

$$\text{Precision} = \frac{\sum \text{True Positive}}{\sum \text{True Positive} + \sum \text{False Positive}},$$

and measures the purity of all positively labeled instances (i.e., the ratio of correct positively labeled instances to all positively labeled instances). Recall is defined as

$$\text{Recall} = \frac{\sum \text{True Positive}}{\sum \text{True Positive} + \sum \text{False Negative}},$$

and measures the completeness of positively labeled instances (i.e., the ratio of correct positively labeled instances to all instances that should have been labeled positive).  $F_1$  score is the harmonic mean of Precision and Recall:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

ROC measures the trade-off of the true positive rate (TPR) to false positive rate (FPR) where

$$\text{TPR} = \frac{\sum \text{True Positive}}{\sum \text{True Positive} + \sum \text{False Negative}},$$

and

$$\text{FPR} = \frac{\sum \text{False Positive}}{\sum \text{False Positive} + \sum \text{True Negative}}.$$

The ROC is generated by evaluating the TPR and FPR at all thresholds of score returned by a classifier. For example, the ROC is calculated for a probabilistic classifier by varying a threshold from 0.0 to 1.0 and calculating FPR and TPR for each value in the range. Area under the curve (AUC) is a common single metric to compare ROC curves, and as the name implies, is just the area under the ROC curve. An AUC of 1 is perfect, and an AUC of 0.5 is the same as chance in a binary classifier.

Averaging results over classes is done using both a micro and macro average. Micro averaging takes into account the number of elements in the test set. This means that smaller classes will account for less in the average than larger classes. Macro, on the other hand, averages over all classes regardless of the number of elements in each individual class. For this paper, macro averaging is probably a better predictor of performance as the distributions of classes in our dataset may not accurately represent the true distributions in the wild. However, both measures are provided for completeness.

## B. Experimental Designs

The proposed technique is evaluated using three different experimental designs:

- 1) binary classification with random holdout test sets to measure the general ability to detect DGA vs. non-DGA,
- 2) binary classification with holdout DGA algorithm families to measure the ability to detect new DGAs, and
- 3) multiclass classification to measure the ability to distinguish one DGA algorithm from another.

The binary classification experimental design tests each DGA classifier for its ability to make an accurate binary decision: *DGA* or *not DGA*. The DGA class consists of domains from all thirty families in our training set. This experiment is run using  $n$ -fold cross validation with ten folds. Evaluation is accomplished with both an ROC as well as a detailed Precision, Recall and  $F_1$  score broken down by each class. Both the micro and macro averages of Precision, Recall and  $F_1$  score are also given.

In the second experiment, we test each classifier’s ability to discover new DGA families not used in the training set. The ten smallest DGA families are removed from the dataset and each classifier is trained on all samples from the remaining classes. Precision, Recall and  $F_1$  score is calculated on the test set. In addition, we find both the micro and macro average of these scores over all classes for each algorithm.

The multiclass classification design tests each DGA classifier for its ability to make an accurate decision on the family of DGA. The random forest DGA classifier (using manual features) uses a One vs. Rest while the LSTM and Bigram classifiers do a direct multiclass classification. We display a class breakdown of Precision, Recall and  $F_1$  score for each class as well as the micro and macro average.

## C. Data

This paper uses open datasets for reproducibility. A real-world system should use an expanded dataset to make it more difficult for an adversary to reverse engineer and defeat the classifier. The experimental designs use data from two sources.

- 1) The Alexa top 1 million domains [26] are used for training domains that are not DGAs.
- 2) The OSINT DGA feed from Bambenek Consulting [27] is used for DGA domains.

The OSINT DGA feed consists of thirty families of DGAs with a varying number of examples from each class. This feed contains approximately 750,000 DGA examples.

## D. Comparison to state of the art

For each experiment, we compare the featureless LSTM DGA classifier to

- a featureless HMM model<sup>3</sup> defined in [2],

- logistic regression on character bigrams (simple features), and
- a random forest DGA classifier using manually-crafted domain features defined in [2], [3], [4], [18].

In particular, the manually crafted features of the random forest DGA classifier include the following:

- length of domain name,
- entropy of character distribution in domain name,
- vowel to consonant ratio,
- Alexa 1M  $n$ -gram frequency distribution co-occurrence count, where  $n = 3, 4$  or  $5$ ,
- $n$ -gram normality score, and
- meaningful characters ratio.

Note that for the  $n$ -gram normality score, we use  $n = 3$ ,  $n = 4$  and  $n = 5$  as three distinct features as opposed to  $n = 1$ ,  $n = 2$  and  $n = 3$  as in [18] since the larger  $n$ -gram size performed better in preliminary experiments. In addition, features were trained in a random forest DGA classifier as opposed to a Mahalanobis distance classifier as used in [18] as the random forest DGA classifier produced better results.

Four separate HMMs are trained with one trained on the non-DGA class, and three trained on the three largest DGA classes in terms of support (`Post`, `banjori`, and `ramnit`). The number of hidden states is set to the average length of the domain names in the training set. We use the Neyman-Pearson likelihood ratio test to classify a domain as DGA generated if

$$\log P_{i^*} - \log P_0 \geq \eta,$$

where

$$i^* = \arg \max_{i \in \{\text{banjori, ramnit, Post}\}} P_i,$$

$P_0$  is the probability of being a non-DGA, and  $\eta$  is a user specified threshold. There are a few key differences from the HMM presented in [2]. Authors in [2] use a distinct HMM for each family of DGA, while we only create an HMM for the three largest classes of DGAs in the training set. In addition, we use the Neyman-Pearson likelihood ratio test as opposed to a threshold directly on the maximum HMM score from the DGA HMMs. Preliminary results showed a significant improvement in ROC over the algorithm presented in [2] when using these updates.

Even with the improved algorithm, the HMM performed worse than other techniques evaluated in this paper. This is especially true for the multiclass experiment. The original HMM algorithm in [2] was presented on only four classes, each with a significant support. This is unlike our setup that has thirty classes with varying degrees of support. For this reason we omit HMM results for the multiclass experiment.

We also compare our results with those of retrospective techniques as reported in [5]. This comparison is only done for the binary classification as our dataset only contains

<sup>3</sup>HMM is excluded from the multiclass experiment due to poor performance.

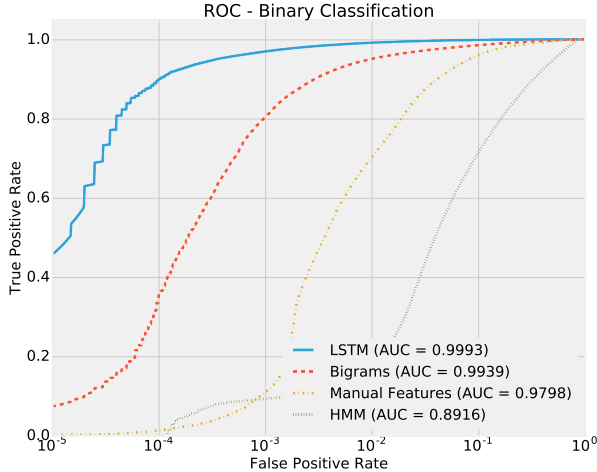


Fig. 4: ROC curves for binary classification of DGA and non-DGA generated domains using the LSTM model, logistic regression with bigram features, random forest classifier with manual features, and HMM classifier.

TABLE I: True Positive Rates of LSTM compared to Retrospective techniques

Technique	True Positive Rate	False Positive Rate
KL Divergence [3], [4]	< 0.5	0.05
NXDomains [5]	0.94	0.002
LSTM	<b>0.98</b>	<b>0.001</b>

domain names without any contextual information. In addition, retrospective techniques perform far worse than real-time techniques for binary classification and, therefore, will likely degrade even further for multiclass classification.

## V. RESULTS

Results for the three experiments and an interpretation of model performance are presented in this section.

### A. Binary Classification

The ROC curves for the HMM, random forest classifier with manually-crafted features (Manual Features), logistic regression classifier on character bigrams (Bigrams), and LSTM DGA classifier (LSTM) are presented in Fig. 4. Note that the abscissa (false positive rate) is on a log scale to highlight the differences in the algorithms. LSTM provides the best performance with an AUC of 0.9993 with the bigram model at 0.9939. The difference between the two algorithms may seem small, but are actually quite significant in a production system. As an example, the LSTM model can classify 90% of all DGAs with a 1 in 10,000 false positive rate. On the other hand, a Bigram model will classify the same percentage of DGA's with a 1 in 550 false positive rate (i.e., the Bigram model produces a false positive rate that is 20× that of the LSTM model).

TABLE III: Recall for all leave-out classes

Domain Type	HMM	Features	Bigram	LSTM	Support
bedep	0.83	<b>0.99</b>	<b>0.99</b>	<b>0.99</b>	172
beebone	0.00	<b>1.00</b>	0.00	0.00	210
corebot	0.59	<b>1.00</b>	0.71	0.77	280
cryptowall	<b>0.30</b>	0.20	0.18	0.20	94
dircrypt	0.94	0.91	0.94	<b>0.97</b>	510
fobber	0.93	0.93	0.95	<b>0.99</b>	600
hesperbot	0.90	0.76	0.86	<b>0.92</b>	192
matsnu	0.00	0.02	<b>0.04</b>	0.0	48
symmi	0.00	<b>1.00</b>	0.11	0.06	64
tempedreve	0.81	<b>0.61</b>	0.80	<b>0.84</b>	249
micro	0.78	<b>0.90</b>	0.80	0.81	
macro	0.53	<b>0.74</b>	0.558	0.642	

The breakdown of Precision, Recall, and  $F_1$  for each class as classified by the binary classifiers is given in Table II. The support (size of test set) is given in the last column. In general, classes that are the most difficult to detect have smaller support. This is expected as they have a smaller contribution to model updates during training than larger classes. In addition matsnu was undetectable by all algorithms. matsnu is a dictionary-based DGA, meaning it is created by randomly selecting and concatenating multiple words from a dictionary. Interestingly, suppbobx is also a dictionary based DGA, but was detectable (to some extent) by the LSTM. The size of the suppbobx training was about twenty times that of matsnu allowing for repeats of randomly selected dictionary words. These repeats allow the LSTM to learn the dictionaries of such DGAs. We leave an in-depth analysis of dictionary based DGA to future work.

The HMM performed worse than expected. The results presented in [2] only used a small number of homogenous DGA families (Conficker, Murofet, Bobax and, Sinowal) while the experiments in this paper use over 30 different families. Some of these families in this paper are related, but overall, our results were generated from a larger/more rich dataset. As discussed later in this paper, the letter distributions are very different across the 30 DGA families used in this paper. For example, DGA families such as Cryptolocker and ramnit have near uniform distributions over letters, dyre has a uniform distribution over hexadecimal characters with a dictionary word as a prefix, and suppbobx and matsnu use English words to create domains giving a distribution very similar to english based domains. In contrast, Conficker [28], Murofet [29], Bobax [30] and Sinowal [31] all use a generator that gives a uniform distribution over letters similar to Cryptolocker and ramnit.

Table I displays the true positive rate and false positive rate for retrospective techniques as compared to the LSTM technique presented by this paper. As can be seen, the LSTM technique significantly outperforms the best retrospective techniques.

### B. Leave-Class-Out Binary Classification

The binary leave-one-out classifier is interesting as it tests each algorithm's robustness to DGA families not seen during

TABLE II: Precision, Recall and  $F_1$  Score for Binary Classifiers

Domain Type	Precision				Recall				$F_1$ Score				Support
	HMM	Features	Bigram	LSTM	HMM	Features	Bigram	LSTM	HMM	Features	Bigram	LSTM	
Alexa	0.8300	0.9400	0.9700	0.9900	1.0000	1.0000	1.0000	1.0000	0.9100	0.9700	0.9900	0.9900	300064
Cryptolocker	1.0000	1.0000	1.0000	1.0000	0.9000	0.9800	0.9700	0.9900	0.9500	0.9900	0.9900	0.9900	1799
P2P Gameover Zeus	1.0000	1.0000	1.0000	1.0000	0.9900	1.0000	1.0000	1.0000	0.9900	1.0000	1.0000	1.0000	298
Post Tovar GOZ	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	19863
Volatile Cedar / Explosive	0.0000	1.0000	1.0000	1.0000	0.0000	0.4600	0.4900	0.9900	0.0000	0.6300	0.6600	1.0000	294
banjori	1.0000	1.0000	1.0000	1.0000	0.5900	0.9400	1.0000	1.0000	0.7400	0.9700	1.0000	1.0000	121678
bedep	1.0000	1.0000	1.0000	1.0000	0.8100	1.0000	1.0000	1.0000	0.8900	1.0000	1.0000	1.0000	53
beebone	0.0000	1.0000	1.0000	1.0000	0.0000	1.0000	0.9700	1.0000	0.0000	1.0000	0.9900	1.0000	65
corebot	1.0000	1.0000	1.0000	1.0000	0.5900	1.0000	1.0000	0.9600	0.7400	1.0000	1.0000	0.9800	81
cryptowall	1.0000	1.0000	1.0000	1.0000	0.1100	0.0600	0.1400	0.1200	0.1900	0.1100	0.2500	0.2100	29
dircrypt	1.0000	1.0000	1.0000	1.0000	0.9100	0.9200	0.9600	0.9600	0.9500	0.9600	0.9800	0.9800	150
dyre	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9900	1.0000	1.0000	1.0000	0.9900	1.0000	2389
fobber	1.0000	1.0000	1.0000	1.0000	0.8900	0.9600	0.9700	0.9700	0.9400	0.9800	0.9800	0.9900	181
geodo	1.0000	1.0000	1.0000	1.0000	0.9100	1.0000	0.9900	0.9900	0.9500	1.0000	1.0000	1.0000	173
hesperbot	1.0000	1.0000	1.0000	1.0000	0.8300	0.7700	0.8500	0.9700	0.9100	0.8700	0.9200	0.9800	58
matsnu	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	14
murofet	1.0000	1.0000	1.0000	1.0000	0.9200	1.0000	0.9900	1.0000	0.9600	1.0000	1.0000	1.0000	4292
necurs	1.0000	1.0000	1.0000	1.0000	0.8800	0.8400	0.9400	0.9600	0.9400	0.9100	0.9700	0.9800	1232
nymaim	1.0000	1.0000	1.0000	1.0000	0.8000	0.5600	0.7300	0.8000	0.8900	0.7200	0.8500	0.8900	1815
pushdo	1.0000	1.0000	1.0000	1.0000	0.6600	0.4700	0.5600	0.6000	0.7900	0.6400	0.7200	0.7500	507
pykspa	1.0000	1.0000	1.0000	1.0000	0.7200	0.5400	0.7700	0.9000	0.8400	0.7000	0.8700	0.9500	4250
qakbot	1.0000	1.0000	1.0000	1.0000	0.9100	0.9600	0.9600	0.9800	0.9500	0.9800	0.9800	0.9900	1517
ramnit	1.0000	1.0000	1.0000	1.0000	0.8800	0.9100	0.9400	0.9600	0.9400	0.9500	0.9700	0.9800	27439
ranbyus	1.0000	1.0000	1.0000	1.0000	0.9000	1.0000	0.9800	0.9800	0.9500	1.0000	0.9900	0.9900	2625
shifu	1.0000	1.0000	1.0000	1.0000	0.7200	0.2100	0.6600	0.7700	0.8400	0.3500	0.8000	0.8700	697
shiotob/urlzone/bebloh	1.0000	1.0000	1.0000	1.0000	0.9000	0.9700	0.9500	0.9800	0.9500	0.9900	0.9700	0.9900	3031
smda	1.0000	1.0000	1.0000	1.0000	0.5600	0.0800	0.4000	0.9200	0.7100	0.1400	0.5800	0.9600	4449
suppobox	1.0000	0.0000	1.0000	1.0000	0.0100	0.0000	0.0000	0.3200	0.0200	0.0000	0.0100	0.4800	298
symmi	0.0000	1.0000	1.0000	1.0000	0.0000	1.0000	0.7900	0.6900	0.0000	1.0000	0.8800	0.8200	18
tempedreve	1.0000	1.0000	1.0000	1.0000	0.7600	0.5700	0.8500	0.7700	0.8600	0.7300	0.9200	0.8700	74
tinba	1.0000	1.0000	1.0000	1.0000	0.8900	0.9800	0.9700	0.9900	0.9400	0.9900	0.9900	0.9900	18505
Micro Average	0.9008	0.9647	0.9826	<b>0.9942</b>	0.8815	0.9639	0.9848	<b>0.9937</b>	0.8739	0.9593	0.9851	<b>0.9906</b>	16708
Macro Average	0.8655	0.9335	0.9668	<b>0.9674</b>	0.6787	0.7477	0.8006	<b>0.8571</b>	0.7335	0.7929	0.8468	<b>0.8913</b>	16708

training. Only Recall is presented for this experiment as there are no non-DGA generated domains in this test set. The results for this experiment are shown in Table III.

The manual features random forest classifier performs best in terms of both micro and macro average. On the other hand, the LSTM classifier has the most families that it performs best on (five in total as opposed to four in total for the manual features classifier). The biggest discrepancy between manual features and LSTM was with *beebone*. In particular, the manual features classifier identifies all of the *beebone* samples, while the LSTM model recovers none. The domain names from *beebone* have a rigid structure, like `ns1.backdates13.biz` and `ns1.backdates0.biz`, so that the LSTM model was unable to learn the structure that included the word `backdates` without training data. The results are nearly as dramatic for *symmi*, which produces nearly-pronounceable domain names like `hakueshoubar.ddns.net`, by drawing a random vowel or a random consonant at each even-numbered index, then drawing a random character of the opposite class (vowel/consonant) in the subsequent index location. These examples highlight blind spots in the LSTM classifier. However, these blind spots can be easily fixed through training with the use of an adversarial network (i.e., train a generator network that creates domains that confuses our classifier).

Apparently, the structure of some DGA families—even if not elaborately designed—are peculiar enough to necessitate their inclusion in the training set. As evident in the results for Experiment 1 in Table II, the LSTM readily detects these families with distinct structure when accounted for in the training set with sufficient support. The manual features

appear to be generic enough to detect these families with high recall. However, its important to note that manual features were designed specifically for known DGA families and all of our DGAs in our test set are known (i.e., our dataset is known and labeled) making this experiment biased to a feature based classifier. Even with this bias, the LSTM classifier still performs best in terms of the number of DGA families it detects.

### C. Multiclass

The HMM results were omitted from the multiclass experiments due to poor performance. As stated previously, the HMM algorithm was designed for few DGAs, whereas our experiments include over 30 classes. Precision, Recall, and  $F_1$  is displayed in Table IV for the random forest classifier with manual features (Manual Features), multinomial logistic regression on character bigrams (Bigram) and the LSTM classifier. The LSTM classifier significantly outperforms the other two algorithms in both the micro and macro averaged Precision, Recall, and  $F_1$  score. In general, poor performance resulted from classes with small representation. One exception was *Cryptolocker*, which no multiclass classifier was able to detect. However, all the binary classifiers were able to distinguish *Cryptolocker* from other families.

Fig. 5 shows the confusion matrix for the LSTM multiclass classifier. A large number of the incorrectly classified *Cryptolocker* DGAs are classified as *ramnit*. To further investigate, the unigram distributions for four DGA families and *Alexa* are shown in Fig. 6. The distributions for *Cryptolocker* and *ramnit* are both uniform over the same range. This is expected as they are both generated



TABLE IV: Precision, Recall and  $F_1$  Score for Multiclass Classifiers

Domain Type	Precision			Recall			$F_1$ Score			Support
	Features	Bigram	LSTM	Features	Bigram	LSTM	Features	Bigram	LSTM	
Alexa	0.914	0.980	<b>0.990</b>	0.960	0.990	<b>1.000</b>	0.940	0.988	<b>0.990</b>	199978
Cryptolocker	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1189
P2P Gameover Zeus	0.000	<b>0.343</b>	0.327	0.000	<b>0.288</b>	0.217	0.000	<b>0.308</b>	0.247	196
Post Tovar GOZ	0.941	<b>1.000</b>	<b>1.000</b>	<b>1.000</b>	<b>1.000</b>	<b>1.000</b>	0.970	<b>1.000</b>	<b>1.000</b>	13185
Volatile Cedar / Explosive	0.000	<b>1.000</b>	0.987	0.000	<b>1.000</b>	0.980	0.000	<b>1.000</b>	0.980	200
banjori	0.900	0.990	<b>1.000</b>	0.938	<b>1.000</b>	<b>1.000</b>	0.920	<b>1.000</b>	<b>1.000</b>	81281
bedep	0.000	0.000	<b>0.943</b>	0.000	0.000	<b>0.107</b>	0.000	0.000	<b>0.187</b>	34
beebone	<b>1.000</b>	<b>1.000</b>	<b>1.000</b>	0.560	<b>1.000</b>	<b>1.000</b>	0.713	<b>1.000</b>	<b>1.000</b>	42
corebot	0.000	<b>1.000</b>	<b>1.000</b>	0.000	0.980	<b>0.990</b>	0.000	0.990	<b>0.993</b>	54
cryptowall	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	15
dircrypt	0.000	<b>0.083</b>	0.000	0.000	<b>0.010</b>	0.000	0.000	<b>0.020</b>	0.000	100
dyre	0.985	0.988	<b>1.000</b>	<b>1.000</b>	0.988	<b>1.000</b>	0.991	0.988	<b>1.000</b>	1600
fobber	0.000	0.000	<b>0.177</b>	0.000	0.000	<b>0.023</b>	0.000	0.000	<b>0.040</b>	121
geodo	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	114
hesperbot	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	36
matsnu	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	9
murofet	<b>0.883</b>	0.643	0.783	0.066	0.542	<b>0.700</b>	0.122	0.590	<b>0.737</b>	2845
necurs	0.000	0.000	<b>0.643</b>	0.000	0.000	<b>0.093</b>	0.000	0.000	<b>0.160</b>	827
nymaim	0.000	0.390	<b>0.477</b>	0.000	0.113	<b>0.190</b>	0.000	0.175	<b>0.267</b>	1222
pushdo	0.000	0.770	<b>0.853</b>	0.000	0.588	<b>0.640</b>	0.000	0.665	<b>0.730</b>	339
pykspa	0.000	0.788	<b>0.910</b>	0.000	0.593	<b>0.713</b>	0.000	0.675	<b>0.800</b>	2827
qakbot	0.000	<b>0.590</b>	0.590	0.000	0.232	<b>0.387</b>	0.000	0.338	<b>0.463</b>	993
ramnit	0.566	0.637	<b>0.770</b>	0.654	0.763	<b>0.850</b>	0.605	0.690	<b>0.810</b>	18308
ranbyus	0.439	0.000	<b>0.450</b>	0.000	0.000	<b>0.517</b>	0.001	0.000	<b>0.460</b>	1736
shifu	0.000	0.037	<b>0.560</b>	0.000	0.003	<b>0.570</b>	0.000	0.007	<b>0.553</b>	465
shiotob/urlzone/bebloh	0.000	0.965	<b>0.973</b>	0.000	0.853	<b>0.907</b>	0.000	0.907	<b>0.940</b>	2016
simda	0.000	0.840	<b>0.930</b>	0.000	0.750	<b>0.977</b>	0.000	0.792	<b>0.950</b>	2955
suppobox	0.000	0.392	<b>0.833</b>	0.000	0.062	<b>0.517</b>	0.000	0.112	<b>0.627</b>	197
symmi	0.000	0.625	<b>0.913</b>	0.000	0.117	<b>0.857</b>	0.000	0.200	<b>0.883</b>	11
tempedreve	0.000	<b>0.043</b>	0.000	0.000	<b>0.010</b>	0.000	0.000	<b>0.018</b>	0.000	50
tinba	0.821	0.735	<b>0.910</b>	0.923	0.802	<b>0.990</b>	0.869	0.767	<b>0.950</b>	12332
Micro Average	0.851	0.933	<b>0.963</b>	0.888	0.944	<b>0.970</b>	0.867	0.940	<b>0.963</b>	11138
Macro Average	0.240	0.479	<b>0.614</b>	0.197	0.409	<b>0.523</b>	0.198	0.427	<b>0.541</b>	11138

using a series of multiplies, divisions and modulus based on a single seed [13], [10]. On the other hand, `suppobox` is interesting as it generates unigrams similar to distributions seen by the Alexa top one million domains and is often confused with the benign set. As discussed earlier, `suppobox` is an English dictionary-based DGA, meaning domains are constructed by concatenating multiple, randomly chosen words from the English dictionary. Interestingly, only the LSTM classifier was able to consistently detect `suppobox` (as seen in Table II). This shows LSTM’s ability to extract some deep understanding that is lost by other classifiers. Specifically, the LSTM actually learns the dictionary used by `suppobox` to construct domains.

Fig. 7 shows the all-to-all cosine distance of the unigram distribution between all DGA families and the Alexa top one million domains. `dyre` stands out as it is extremely dissimilar to other algorithms. This is not surprising when comparing this figure to Table 6. `dyre` has a nearly uniform distribution over primarily hexadecimal numbers (non-hexadecimal letters exist, but are rare).

When comparing both Fig. 5, Fig. 7, and Table II, some correlation can be seen between the unigram distribution and DGA algorithms that are often misclassified. This suggests that it’s not only the lack of representation of these algorithms in the training set, but also the distribution of letters that is causing much of the misclassification. More specifically, many

DGAs produce domains that look nearly identical in terms of their character distributions making multiclass classification difficult if not impossible. To test this, we performed agglomerative clustering on each DGA’s family unigram distribution using cosine distance. We set a threshold of 0.2 to define super families (the threshold was chosen using domain knowledge of DGA families). These super families are shown in Table V. Interesting super families include Super Family 4 (dictionary-based DGAs), Super Family 5 (randomly selected character DGAs), and Super Family 7 (randomly selected characters with near equal vowels and consonants).

The same multiclass classification experiment was run on these super families and the results are shown in VI. As expected, all three classifiers performed much better on super families. Results demonstrate that an actual deployment of a multiclass DGA classification would be best run on super families, often alerting on groups of DGAs instead of alerting on a single family. Again, the LSTM classifier performs significantly better than other algorithms.

#### D. Model Interpretability

We analyze the binary LSTM classifier in order to provide some intuition about the function of the various layers. It is important to note that in the LSTM model, each layer in Fig. 1 is jointly optimized for the binary classification task.

TABLE VI: Precision, Recall and  $F_1$  Score for Multiclass Classifiers

Domain Type	Precision			Recall			$F_1$ Score			Support
	Features	Bigram	LSTM	Features	Bigram	LSTM	Features	Bigram	LSTM	
Alexa	0.930	0.980	<b>0.990</b>	0.960	0.990	<b>1.000</b>	0.940	<b>0.990</b>	<b>0.990</b>	199906
Super Family 0	0.980	0.990	<b>1.000</b>	<b>1.000</b>	0.990	<b>1.000</b>	0.990	0.990	<b>1.000</b>	1603
Super Family 1	<b>1.000</b>	<b>1.000</b>	<b>1.000</b>	0.590	<b>1.000</b>	<b>1.000</b>	0.740	<b>1.000</b>	<b>1.000</b>	43
Super Family 2	0.000	<b>1.000</b>	<b>1.000</b>	0.000	<b>1.000</b>	0.970	0.000	<b>1.000</b>	0.990	203
Super Family 3	0.000	0.950	<b>0.980</b>	0.000	0.810	<b>0.900</b>	0.000	0.870	<b>0.940</b>	1998
Super Family 4	0.910	0.990	<b>1.000</b>	0.920	<b>1.000</b>	<b>1.000</b>	0.910	0.990	<b>1.000</b>	81559
Super Family 5	0.870	0.950	<b>0.970</b>	0.880	0.940	<b>0.970</b>	0.870	0.950	<b>0.970</b>	40450
Super Family 6	0.000	0.840	<b>0.960</b>	0.000	0.550	<b>0.670</b>	0.000	0.670	<b>0.790</b>	2877
Super Family 7	0.000	0.830	<b>0.940</b>	0.000	0.680	<b>0.910</b>	0.000	0.750	<b>0.920</b>	3326
Super Family 8	0.940	0.990	<b>1.000</b>	<b>1.000</b>	0.990	<b>1.000</b>	0.970	0.990	<b>1.000</b>	13267
Super Family 9	0.000	0.980	<b>1.000</b>	0.000	0.910	<b>1.000</b>	0.000	0.940	<b>1.000</b>	52
Super Family 10	0.000	0.000	<b>0.910</b>	0.000	0.000	<b>0.830</b>	0.000	0.000	<b>0.870</b>	11
Micro Average	0.896	0.977	<b>0.990</b>	0.919	0.979	<b>0.992</b>	0.903	0.980	<b>0.988</b>	28774
Macro Average	0.469	0.875	<b>0.979</b>	0.446	0.822	<b>0.938</b>	0.452	0.845	<b>0.956</b>	28774

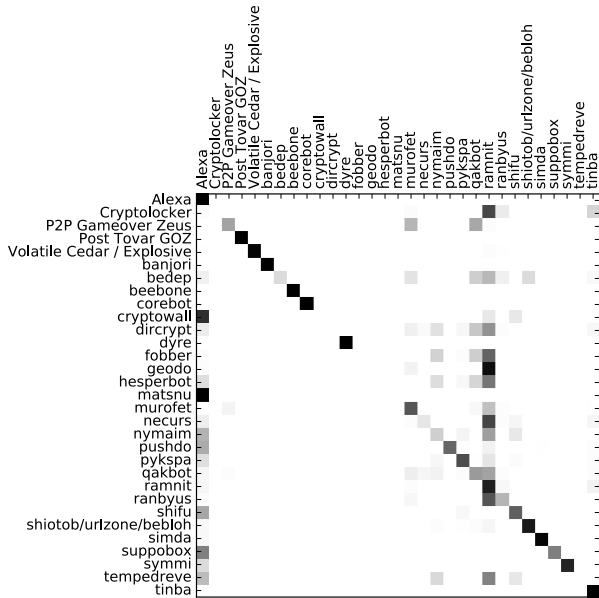


Fig. 5: Confusion matrix for the LSTM multiclass model. Blocks represent the fraction of DGA families on the vertical axis classified as DGA families on the horizontal axis, where 0 is depicted as white and 1 depicted as black. A perfect classifier would produce an identity matrix composed of black blocks.

Nevertheless, analyzing each layer independently does provide some intuition about the model’s operation and performance.

The embedding layer in Fig. 1 learns a 128-dimensional vector representation for each character in the set of valid domain characters. A two-dimensional linear projection (via PCA) of the character embeddings is shown in Fig. 8. It is clear that the learned embedding consists of non-orthogonal vectors for each character. This is in contrast to the orthonormal one-hot encoding of bigrams used in the logistic regression character bigram model. The placement of vectors in the

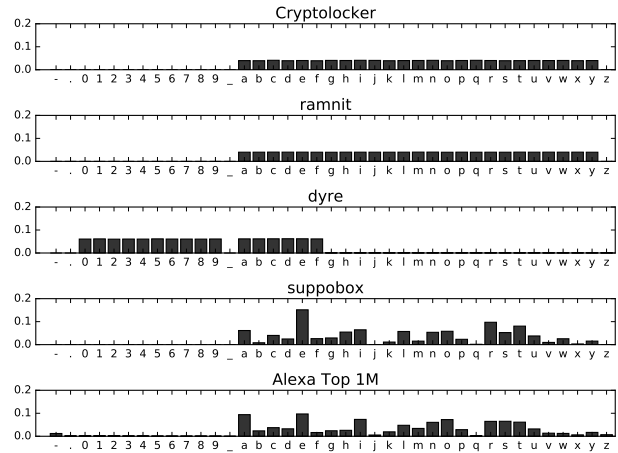


Fig. 6: Unigram distributions for Cryptolocker, ramnit, dyre, suppobox and the Alexa top one million.

TABLE V: DGA Super Families

Super Family	Member Families
Super Family 0	dyre
Super Family 1	beebone
Super Family 2	Volatile Cedar / Explosive
Super Family 3	shioto/ur/zone/bebloh
Super Family 4	banjori, cryptowall, matsnu, suppobox
Super Family 5	murefet, tinba, shifu, geodo, necurs, Cryptolocker, ramnit, ranbyus, bedep, hesperbot, tempedreve, fobber, nymaim, qakbot, P2P Gameover Zeus, dircrypt
Super Family 6	pykspa
Super Family 7	pushdo, simda
Super Family 8	Post Tovar GOZ
Super Family 9	corebot
Super Family 10	symmi

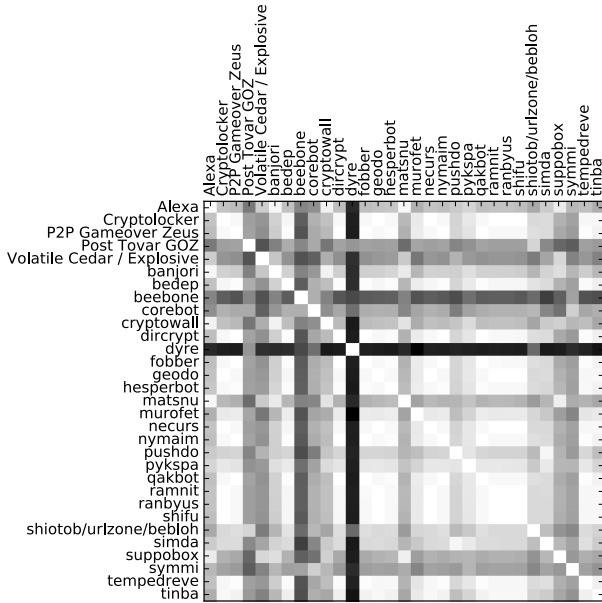


Fig. 7: All-to-all cosine distance comparison of unigram distributions of all DGA families and the Alexa top one million. Distances range from 0 to 1 with 0 depicted as white and 1 depicted as black.

embedding space (and subsequently, the two-dimensional plot) relates to the similarity or interchangeability of characters for the DGA vs. non-DGA discrimination task. For example, one would infer from the plot that replacing “9” with “5” would have much less effect on the score of the DGA classifier than would replacing “9” with “w”. The plot shows that there are obvious clusters of numeric digits and alphabetic characters (and underscore), while the less-common hyphen and period are fairly dissimilar to every other character.

Next, we investigate the *state* (or *memory*) of several LSTM cells in the second layer of the LSTM model in Fig. 1. The state of an LSTM cell has an initial value that is updated as each character of a domain is fed through the model. It is a function of the current input (embedded character vector) and the previous emission of the LSTM cell. In turn, the LSTM’s emission is a function of the current state, current input, and previous emission. In our model, the final emission (corresponding to the last character in the domain) from each of 128 LSTM cells is fed to the final logistic regression layer of the model to produce the DGA score.

Each LSTM cell acts somewhat as an optimized feature extractor on the sequences of embedded character vectors produced from the previous embedding layer, and the cell’s state provides an indication of what the cell is tracking. Similar to [32], Fig. 9 shows the tanh of a particular LSTM cell’s state (called *memory* in [32]) as it is updated character-by-character during a prediction task. As shown in Fig. 9, some states in our model have a tendency to track common characteristics of domain names in the dataset. For example, Fig. 9(a) shows a state that seems to trend with domain name length, with soft resets on periods and hyphens. The LSTM cell state depicted in Fig. 9(b) appears to accumulate large

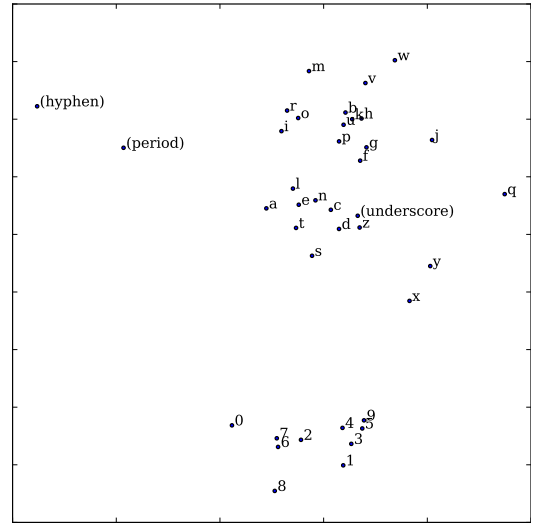


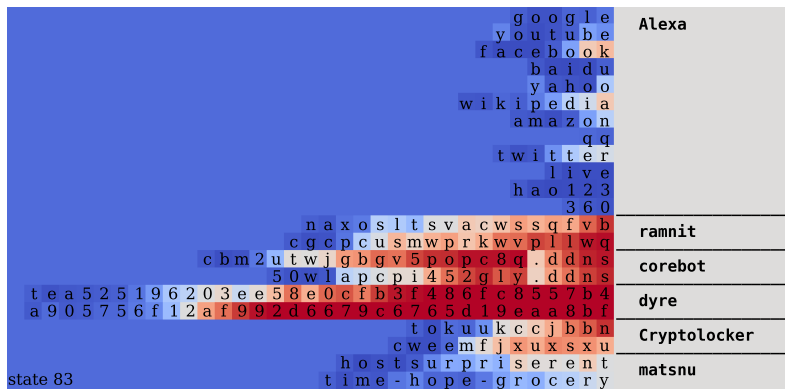
Fig. 8: Two-dimensional linear projection (PCA) of the embedded character vectors learned by the LSTM binary classifier. Note that the model groups characters by similar effect on the LSTM layer’s states and the subsequent model loss.

values for long sequences of random alphanumeric characters. The state in Fig. 9(c) seems to accumulate value on sequences of hexadecimal characters, as is the predominant pattern in *dyre*. Finally, Fig. 9(d) depicts the most common scenario we encountered while inspecting states: it’s generally very difficult to determine precisely what the state is tracking. We note that our application of LSTMs for DGA classification does not yield quite as clearly the distinctive purpose of states as has been demonstrated for natural language models [32].

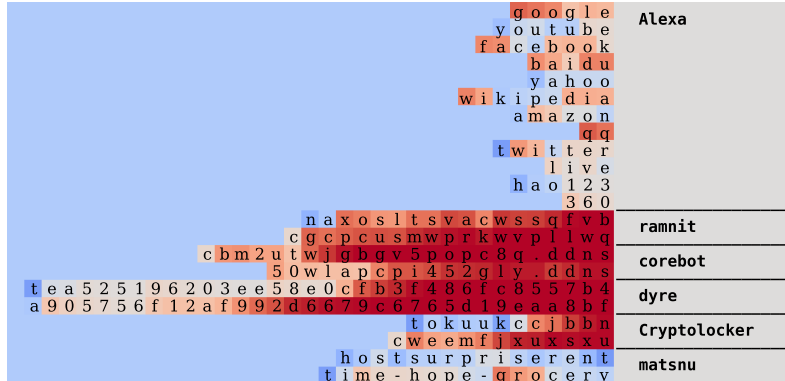
## VI. CONCLUSION

This paper presented an approach using LSTM networks to classify DGA generated domains. LSTMs are advantageous over other techniques as they are featureless, using raw domain names as its input. There is no need to manually create features that are difficult to maintain and can be rendered useless in an adversarial machine learning setting. In addition, an LSTM classifier can be run in real-time on single domains on standard commodity hardware making it trivial to deploy in virtually all security settings. Experiments on publicly-available datasets showed that the LSTM classifier performed significantly better than other techniques (both real-time and retrospective), with the ability to classify 90% of DGAs with a false positive rate of  $10^{-4}$ . In addition, the LSTM classifier may be trivially modified for multiclass classification, which can provide context about the origin and intent of the domain-generating malware.

An in-depth analysis of results showed that the most difficult algorithms to classify are, intuitively, those that are modeled from a similar character distribution as domains in the Alexa top one million. Some of these DGA families concatenate randomly selected words from (typically) English dictionaries. However, the LSTM classifier was able to distinguish those DGA families when the amount of training examples were significant and the families were grouped together in super families.



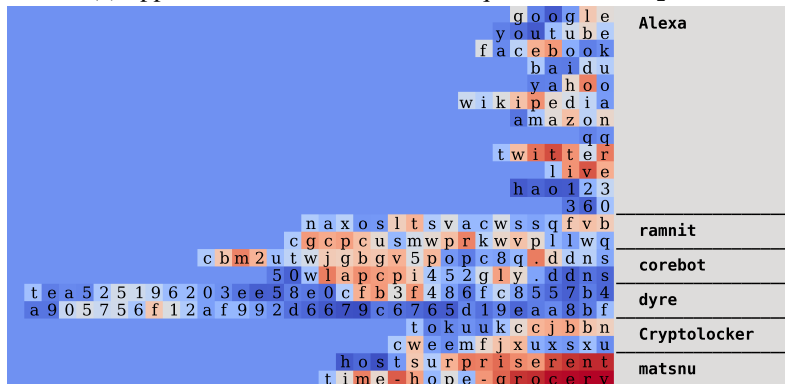
(a) approximately tracks long domain names, with a soft reset on period and hyphen



(b) appears to track random alphanumeric sequences, as in ramnit, corebot, dyre and Cryptolocker



(c) appears to track hexadecimal sequences, as in dyre



(d) as in this example, it is difficult to ascribe an intuitive function of most states

Fig. 9: Examples of LSTM cell state values as domain characters are fed into the model. Color corresponds to the tanh of the state, and does not necessarily denote DGA or non-DGA. Color preceding a domain name denotes the cell's initial state. Our model correctly identifies DGA or non-DGA for all examples shown except for the final two matsnu examples.

We also provided an in-depth analysis of the functional interpretability of each layer in the LSTM DGA classifier. Our analysis revealed that the model optimized vector embeddings for each character in a somewhat intuitive way, with distinct clusters for alphabetic and numeric digits. Our analysis of the LSTM layer revealed the existence of LSTM cells that track a few somewhat interpretable features such as a hexadecimal and random character sequences. However, we found that most states did not provide clear interpretable evidence of function, in contrast to other applications of LSTMs, e.g., [32].

Like all models, experiments show that our model is sensitive to class imbalance, which limits its ability to detect families with very little support in the training set (e.g., `matsnu`, `symmi` and `cryptowall`). In the extreme case of zero training support, it was found that the LSTM model does not generalize well for detecting all families with very distinctive structure. Manually-engineered features were able to detect some of those families that an LSTM classifier missed, and we hypothesize that this is directly a result of expert-tuned bias in the feature set that cannot be represented in the featureless LSTM model.

All relevant source code and suggestions on deploying a real-world LSTM DGA classifier were provided by this paper. In addition, we reference open datasets to create an equal classifier to that presented in this paper. To the best of our knowledge, the presented system is by far the best performing DGA classification system as well as one of the easiest to deploy.

## REFERENCES

- [1] M. Kührer, C. Rossow, and T. Holz, "Paint it black: Evaluating the effectiveness of malware blacklists," in *Research in Attacks, Intrusions and Defenses*, pp. 1–21, Springer, 2014.
- [2] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon, "From throw-away traffic to bots: detecting the rise of DGA-based malware," in *P21st USENIX Security Symposium (USENIX Security 12)*, pp. 491–506, 2012.
- [3] S. Yadav, A. K. K. Reddy, A. Reddy, and S. Ranjan, "Detecting algorithmically generated malicious domain names," in *Proc. 10th ACM SIGCOMM conference on Internet measurement*, pp. 48–61, ACM, 2010.
- [4] S. Yadav, A. K. K. Reddy, A. N. Reddy, and S. Ranjan, "Detecting algorithmically generated domain-flux attacks with DNS traffic analysis," *Networking, IEEE/ACM Transactions on*, vol. 20, no. 5, pp. 1663–1677, 2012.
- [5] S. Krishnan, T. Taylor, F. Monrose, and J. McHugh, "Crossing the threshold: Detecting network malfeasance via sequential hypothesis testing," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, IEEE, 2013.
- [6] F. Chollet, "keras." <https://github.com/fchollet/keras>, 2016.
- [7] M. Knysz, X. Hu, and K. G. Shin, "Good guys vs. bot guise: Mimicry attacks against fast-flux detection systems," in *INFOCOM, 2011 Proceedings IEEE*, pp. 1844–1852, IEEE, 2011.
- [8] B. Stone-Gross, M. Cova, B. Gilbert, R. Kemmerer, C. Kruegel, and G. Vigna, "Analysis of a botnet takeover," *Security & Privacy, IEEE*, vol. 9, no. 1, pp. 64–72, 2011.
- [9] M. Ward, "Cryptolocker victims to get files back for free," *BBC News, August*, vol. 6, 2014.
- [10] "A closer look at cryptolocker's DGA." <https://blog.fortinet.com/post/a-closer-look-at-cryptolocker-s-dga>. Accessed: 2016-04-22.
- [11] N. Hampton and Z. A. Baig, "Ransomware: Emergence of the cyber-extortion menace," in *Australian Information Security Management Conference*, 2015.
- [12] A. Cherepanov and R. Lipovsky, "Hesperbot-A new, advanced banking trojan in the wild," 2013.
- [13] Symantec, *W32.Ramnit analysis*. 2015-02-24, Version 1.0.
- [14] J. Geffner, "End-to-end analysis of a domain generating algorithm malware family." Black Hat USA 2013, 2013.
- [15] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster, "Building a dynamic reputation system for DNS.," in *USENIX security symposium*, pp. 273–290, 2010.
- [16] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi, "Exposure: Finding malicious domains using passive analysis.," in *18th Annual Network and Distributed System Security Symposium*, 2011.
- [17] L. Bilge, S. Sen, D. Balzarotti, E. Kirda, and C. Kruegel, "Exposure: a passive DNS analysis service to detect and report malicious domains," *ACM Transactions on Information and System Security (TISSEC)*, vol. 16, no. 4, p. 14, 2014.
- [18] S. Schiavoni, F. Maggi, L. Cavallaro, and S. Zanero, "Phoenix: DGA-based botnet tracking and intelligence," in *Detection of intrusions and malware, and vulnerability assessment*, pp. 192–211, Springer, 2014.
- [19] A. J. Robinson, "An application of recurrent nets to phone probability estimation," *Neural Networks, IEEE Transactions on*, vol. 5, no. 2, pp. 298–305, 1994.
- [20] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur, "Recurrent neural network based language model.," in *INTERSPEECH*, vol. 2, p. 3, 2010.
- [21] A. Graves, "Sequence transduction with recurrent neural networks," *arXiv preprint arXiv:1211.3711*, 2012.
- [22] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, "Advances in optimizing recurrent networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 8624–8628, IEEE, 2013.
- [23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [24] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.
- [25] F. A. Gers, N. N. Schraudolph, and J. Schmidhuber, "Learning precise timing with LSTM recurrent networks," *J. Machine Learning Research*, vol. 3, pp. 115–143, 2003.
- [26] "Does Alexa have a list of its top-ranked websites?" <https://support.alexa.com/hc/en-us/articles/200449834-Does-Alexa-have-a-list-of-its-top-ranked-websites->. Accessed: 2016-04-06.
- [27] "Bambenek consulting - master feeds." <http://osint.bambenekconsulting.com/feeds/>. Accessed: 2016-04-06.
- [28] P. A. Porras, H. Saïdi, and V. Yegneswaran, "A foray into conficker's logic and rendezvous points.," in *LEET*, 2009.
- [29] D. Andriess, C. Rossow, B. Stone-Gross, D. Plohmann, and H. Bos, "Highly resilient peer-to-peer botnets are here: An analysis of gameover zeus," in *Malicious and Unwanted Software: The Americas (MALWARE), 2013 8th International Conference on*, pp. 116–123, IEEE, 2013.
- [30] P. Royal, "On the kraken and bobax botnets." [https://www.damballa.com/downloads/r\\_pubs/Kraken\\_Response.pdf](https://www.damballa.com/downloads/r_pubs/Kraken_Response.pdf), 2008.
- [31] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your botnet is my botnet: analysis of a botnet takeover," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 635–647, ACM, 2009.
- [32] A. Karpathy, J. Johnson, and F.-F. Li, "Visualizing and understanding recurrent networks," in *to appear in Proceedings of the International Conference on Learning Representations*, 2016. arXiv preprint arXiv:1506.02078.