Hardening DGA Classifiers Using Adversarial Attacks and IVAP

Charles Grumer

A thesis

submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE IN COMP SCIENCE & SYSTEMS

University of Washington

2019

Committee

Martine De Cock

Anderson Nascimento

Femi Olumofin

Jonathan Peck

Program Authorized to Offer Degree:

Computer Science and Systems

University of Washington

Hardening DGA Classifiers Using Adversarial Attacks and IVAP

Charles Grumer

Chair of the Supervisory Committee:

Dr. Martine De Cock

School of Engineering and Technology

Abstract

Domain generation algorithms (DGAs) are utilized by botmasters as a way to connect malware-infected machines with the botmaster's command-and-control center (C&C). Such a connection allows the botmaster to send and receive information between his machine and the infected machines at will, which has strong privacy, financial, and security implications at both an individual level and on a large scale. As such, the ability to identify DGA domains before users' machines have connected to them is of the utmost importance. A multitude of machine learning classifiers have been developed for the detection of DGAs, which are intended to be able to detect a specific DGA after training on domains generated by it. CharBot is an incredibly simple DGA that has been demonstrated to be very effective at fooling state-of-the-art classifiers; therefore, defensive measures must be taken against this specific technique. This thesis consists of two parts: the use of adversarial attacks to harden DGA classifiers against CharBot and the use of Inductive Venn-Abers Predictors (IVAP) to raise classifiers' predictive scores.

Contents

List of Algorithms

List of Tables

List of Figures

1 Introduction

Malicious actors have good reasons for wanting to connect unknowing users' machines to their command-and-control centers. Once a connection has been established, the malicious actor can not only transfer information to and from the controlled machine, but also use a group of infected machines to perform a large-scale attack.

Past methods for establishing such a connection revolved around hard-coding one or more domains into a piece of malware, which an infected machine would then query. However, once such domains were identified and blacklisted, the malware was effectively useless. Domain generation algorithms serve as a way to deterministically generate a list of domains on both the malicious actor's computer and the infected machines, which renders blacklisting infeasible [4].

Consequentially, cybersecurity practitioners and data scientists hoping to secure machines against such attacks need to be able to detect DGA domains with high accuracy. Machine learning is ideal for this task since various features, either extracted by the data scientist or performed on-the-fly by the machine learning algorithm, can be used to differentiate between benign domains and DGA domains.

A wide range of classifiers have been suggested and implemented for this task. Recurrent Neural Networks (RNNs) are one popular option, as exemplified in [3] and [12], both of which utilize long short-term memory (LSTM) networks. This type of RNN, originally proposed in [1], is ideally suited for sequential data classification such as strings. Other researchers have suggested the use of random-forest classifiers (RF), such as [11], as an alternative to RNN classifiers.

Another common classifier type for DGA detection is convolutional neural networks (CNN) for DGA detection, as was reviewed in [14]. This paper compared two CNNs, two RNNs, and one hybrid CNN/RNN, and found little difference between the various techniques regarding DGA detection rates. The authors have suggested following the principle of *Occam's Razor*, suggesting the simplest solution should be taken.

DGAs are plentiful in the wild, although that does not mean the source code for every DGA is available to data scientists. In order to achieve an acceptably high level of accuracy even the best state-of-the-art classifiers need to be trained on examples generated by each DGA that the model hopes to detect. However, the same training methods are not necessarily successful for all DGA types, and training on some DGAs has the potential to degrade a classifier's ability to recognize other DGAs.

As part of the preparation for this thesis, we developed CharBot, a character-based adversarial attack in which known, benign domains are perturbed at a character level as a means to generate domains that evade detection. In [17], we showed this technique to be successful, and given the method's simplicity, it is important to find ways to identify CharBot's domains, and those of other similarly working DGAs.

The main contributions of my thesis are:

1. Evidence that Inductive Venn-Aber's Predictors effectively harden DGA detection neural networks against a wide range of DGAs
2. Evidence that Inductive Venn-Aber's Predictors serve as an effective defense against perturbation based DGAs

3. Evidence that White Box Adversarial Training of DGA detection neural networks using gradients for loss estimation is too inaccurate to be effective

Part of the work presented in this thesis has been published as:

1. CharBot: A Simple and Effective Method for Evading DGA Classifiers [17]
   J. Peck, C. Nie, R. Sivaguru, **C. Grumer**, F. Olumofin, B. Yu, A. Nascimento, M. De Cock
   IEEE Access 7, p. 91759-91771, 2019

2. Hardening DGA Classifiers Utilizing IVAP [16]
   C. Grumer, J. Peck, F. Olumofin, A. Nascimento, M. De Cock
   Proceedings of IEEE BigData2019 (2019 IEEE International Conference on Big Data), 2019

2 Background

2.1 DNS Terminology

This thesis requires a working knowledge of the domain name system (DNS) and many of the common acronyms used in the field:

- Top-level domain (TLD): this refers to the topmost level of a domain. Common TLDs are *com*, *net*, *org*, et cetera.
- Second-level domain (SLD): this refers to the portion of the domain directly below the TLD, denoted by the presence of a period between the sections. In *google.com*, *google* is the SLD.
- Fully qualified domain name (FQDN): this describes the entire domain name and always features a trailing period. For example, both *google.com.* and *www.google.com.* are FQDNs, but *google.com* without the trailing period is not an FQDN.
- Name server: receives an FQDN and returns the associated IP address if known.
- DNS-valid characters: the set of characters that can make up a valid FQDN, namely the alphanumeric characters and the dash. FQDN's with DNS-*invalid* characters may still query a name server, but will always fail to return an IP address (NXDomain).

2.2 Data Sets

The experiments described in section 4 will require the use of several data sets:

- Alexa: a list of the top one million domain names as collected by Alexa. These domains are all considered to be benign.
- Bambenek: a list of one million DGA domain names collected by Bambenek Consulting. These domains are considered to be non-benign. The breakdown of the DGAs that compose this data set is available in table 1.
- AlexaBamb: a training data set consisting of both benign Alexa domains and non-benign Bambenek domains.
- Qname: a list of one million domain names collected from passive DNS traffic. All domains contained in this list meet the following requirements:
  1. Have existed for thirty or more days.
  2. Have been resolved two or more times.
  3. Have never resulted in an NXDomain response.
- QnameBamb: a training data set consisting of both benign Qname domains and non-benign Bambenek domains.
- TLD list: a list of common TLDs such as *com*, *net*, *org*, *ca*, *info*, et cetera.
- CharBot: a list of DGA domains generated by perturbing characters of benign domains.

For classification purposes, benign domains will be marked as *false* and DGA (non-

Family	Count
banjori	439,223
Post_Tovar_GOZ	143,000
ramnit	97,913
tinba	66,688
qakbot	60,000
murofet	59,100
necurs	49,152
pykspa	19,483
ranbyus	19,240
dyre	17,329
simda	14,755
Cryptolocker	13,000
nymaim	13,000
shiotob-urlzone-bebloh	12,521
kraken	9,688
locky	7,686
chinad	3,328

Family	Count
pushdo	2,760
suppobox	2,361
sphinx	1,663
shifu	2,331
ramdo	2,000
proslikefan	1,938
vawtrak	1500
proslikefan	1300
virut	1300
padcrypt	1248
geodo	1248
Volatile_Cedar	996
pizd	847
corebot	600
fobber	600
dircrypt	570
bedep	528

Family	Count
bamital	520
vidro	400
unknownjs	390
tempedreve	249
beebone	210
hesperbot	192
symmi	128
sisron	124
matsnu	102
cryptowall	94
gozi	72
pandabanker	70
unknowndropper	60
dromedan	4
madmax	3
g01	1

Table 1: Distribution of DGA families in the Bambenek data set

benign) domains will be marked as *true*. Data sets are split for training purposes, and the exact type of splits will be expanded upon in their relevant sections.

2.3 Domain Generation Algorithms (DGAs)

Domain Generation Algorithms (DGAs) are a family of algorithms, primarily utilized by malicious actors known as *botmasters*, which generate a deterministic series of domain names [4]. The botmaster will generate a list of domain names, using their DGA, and register one (or more) of them at that time. The same DGA will also be present in a piece of malware that has infected a series of machines and will generate the same series of domain names, then attempt to connect to each successive domain until a connection is made. Once a connection has been established, the botmaster's *command-and-control center* (C&C) may openly communicate with the infected machine, retrieving and transmitting data on request.

In order for a DGA to be successful from the perspective of the botmaster, several

attributes must be present. First, the algorithm must be deterministic so that the botmaster and all infected machines produce identical domain lists. Second, the DGA needs to be able to generate a very large number of domains; this can be accomplished by adding a pseudorandomizaton to the DGA and using a periodically changing (but easily accessible) starting seed, such as the date. These two attributes ensure that a botmaster's malware is not made obsolete by simply blocking all utilized domains.

2.4 DGA Classifiers

The term DGA classifier can be used to describe two types of machine learning models. The first type of classifier is used to determine whether or not a given domain name was generated by a DGA. The second type is used to determine what family of DGA generated a given DGA domain. For the purposes of this thesis, we will only be considering the former. Various neural networks were reviewed in [14], which found little difference in effectiveness between CNNs, RNNs, and hybrid architectures. I primarily reference three neural networks throughout this paper: Invincea to represent a CNN; LSTM.MI to represent an RNN; and MIT to represent a CNN/RNN hybrid.

LSTM.MI is a recurrent neural network proposed in [12] that utilizes LSTM for improved sequence recognition. It differs from other LSTM RNNs primarily due to the cost-sensitive algorithm introduced by the authors. The code can be seen in section 6.2.

Invincea is a convolutional neural network designed for short string classification introduced in [8]. As noted by the authors, the use of CNNs over RNNs for classification over sequences is arguably superior due to the reduced time required for both training and classification. As is noted in [14], the parallel layers of Invincea ensure that pooling occurs across the entire domain name rather than across some substring, as would be the case in many other CNN models. The code can be seen in section 6.1.

The MIT model was proposed in [5] but not for the purposes of DGA detection; its usefulness in this area of research was shown in [14]. The code can be seen in section 6.3.

2.5 Adversarial Machine Learning

Adversarial Machine Learning (AML) is a type of machine learning that focuses on attacking the accuracy and/or reliability of a machine learning model as well as protecting against such attacks. As described by [13], several different types of AML attacks exist. One type, known as an evasion attack, is focused on creating synthetic examples that are likely to be misclassified by machine learning models. Another type, known as a poisoning attack, works by modifying the training data in an attempt to prevent a model from learning to classify accurately. The methods I research are all related to protecting against evasion attacks, although poisoning attacks is a related area for future work.

A wide range of adversarial attacks for the purpose of evading DGA classifiers have

been proposed. In [17], CharBot's seminal work, black-box adversarial attacks are utilized by making small character perturbations on known benign domains; this technique is covered in more detail in section 2.6. The authors of [19] utilized a multi-factor, white-box approach for their DeceptionDGA, in which Alexa data was used to generate domains with properly distributed domain lengths, vowel ratios, character distributions, and character probabilities given the previous character; as was noted in the paper, this essentially normalizes the generated domains at a bigram level. DeepDGA, as described in [3], is arguably the most complicated method for adversarial example generation. It functions by being trained on benign domains and learning to output pseudorandom domains, which can be considered to be malicious for all intents and purposes; once this training is complete, the model is frozen and only requires a pseudorandom seed to produce synthetic domains.

All three of these methods have their advantages. The method I utilized, described in more detail in section 4.1, is related to each of these three techniques in some way. First, my method will be compared to the black-box method shown in CharBot [17]. Second, the domains generated by my proposed white-box adversarial can have their character distributions compared to the benign samples that are perturbed, and can thereby be compared to DeceptionDGA in [19]. Third, both my technique and the DeepDGA method described in [3] attempt to create the adversarial examples that are the most difficult to classify. Additionally, the paper [10] creates white-box adversarial examples for strings, although not specifically for DGA domains. My methodology, as described in section 4.1, is the model after this paper with modifications for the DGA domain of research. It attempts to approximate the loss

caused by character perturbations by using a gradient, which allows for adversarial generations with less computational overhead than many other techniques.

Both malicious actors and data scientists utilize adversarial techniques, albeit for different reasons. The former are attempting to circumvent the classifier, gain knowledge of how it works, and/or damage its ability to classify correctly in the future. The latter are generally trying to test how well their classifiers will hold-up to malicious adversarial attacks, determine any blind-spots that exist in their classifiers, and fill those blind spots with generated data.

The concept of robustness in regard to adversarial training is well-covered in literature [6], [15]. In [6] the authors make several key findings, most notably that powerful attacks, such as the ones described in section 4.1, are needed to ensure a model is sufficiently defensive, and that adversarial examples generated off a weak model fail to harden a more powerful model. This second point can potentially be quantified by comparing adversarial examples generated by the methodology outlined in section 4.1 at various iterations, although such work is beyond the scope of this thesis.

While the methods covered in [6] would require modification for use on DGA classifiers, the concepts are highly transferable. Similarly, [15] covers a range of methodological practices that help with experimental design, but also covers the importance in testing if a model is hardened against a specific attack or is generally robust. The methodologies covered in section 4 ensure that multiple data sets are used for testing, including examples both produced both by CharBot and available in the Bambenek

data set. This will allow us to confirm whether or not the WBAT is singularly strong or robustly strong.

Due to the fact that I am attempting to protect against a specific DGA, CharBot, I expect the white box adversarial techniques to result in improved metrics in regards to CharBot detection while suffering from nominal decreases in overall DGA detection metrics.

2.6 The CharBot Domain Generation Algorithm

CharBot, as published in [17], is a character-based DGA that is designed to be both simple and effective. I implemented it as part of [17], a paper written in collaboration between the University of Washington, Tacoma and Infoblox. CharBot requires three pieces of data in order to operate:

- A list of domain names. I use the second ten thousand Alexa domain names that have six or more characters in the SLD.
- A pseudorandom seed. I use the current date.
- A list of known TLDs.

The pseudorandom seed is used to select the following:

- A domain from the list of benign domains.
- Two characters from within the selected domain.
- Two DNS-valid characters from a uniform distribution.

- A TLD from the TLD list.

After the above selections have been made, CharBot replaces the characters selected from the input domain name with those selected from the uniform distribution. It then appends the selected TLD to the end of the newly perturbed domain. This process can be looped to create a list of new domains, as would be the case if it were to be implanted into a piece of malware.

Algorithm 1: CharBot

Data: a list of SLDs D, a list of TLDs T, a seed s
Result: a DGA domain

1 Initialize the pseudorandom generator with the seed s.
2 Randomly select an SLD d from D.
3 Randomly select two indices i and j so that $1 \leq i, j \leq |d|$.
4 Randomly select two replacement characters c_1 and c_2 from the set of DNS-valid characters.
5 Set $d[i] \leftarrow c_1$ and $d[j] \leftarrow c_2$.
6 Randomly select a TLD t from T.
7 **return** $d.t$

As was discussed in [17] the decision to utilize only two character perturbations was made to highlight two important features. First, fewer character perturbations allows generated domains to remain more similar to their original, input domain; this increases the likelihood that they will not be detected by DGA classifiers and helps preserve the overall character distribution. Second, a real-world DGA must generate domains that are unregistered for the algorithm to be of use to botmasters.

2.7 Inductive Venn-Abers Predictors

The concept of Inductive Venn-Abers Predictors was introduced in [2] as a way to measure the predictive reliability of machine learning models. In doing so, one can easily create probabilistic models that have better performance metrics than their non-calibrated counterparts. There are negatives associated with such methodologies, such as decreased data set sizes, a small number of rejected inferences, and additional run-time. The algorithms utilized to form these probabilistic models are described in depth in [20], a lengthened and up-to-date adaptation of [2].

3 Preliminary Work

I implemented CharBot in the context of a research collaboration between the University of Washington, Tacoma and Infoblox in an attempt to exemplify the relative ease by which DGA classifiers can be fooled. As was noted in CharBot's seminal paper [17], I was successful in several ways. First, CharBot successfully creates previously unregistered domain names (table 2), a necessity for a real-world DGA. Second, CharBot easily evades state-of-the-art classifiers that were not trained to identify CharBot's domains (table 3). Third, CharBot retains its ability to evade classifiers, albeit to a lesser extent, once trained on data sets that were supplemented with CharBot domains, as shown in table 3.

It is worth noting that the results in table 3 also show that DGA detection classifiers retained high true positive rates (TPR) at suitably low false positive rates (FPR) even after training on data sets that included CharBot data.

DGA	Seed	# Unique Synthetic Domains	# Unregistered Domains (out of 500 samples)
CharBot	2018-12-04	100,000	500 (100%)
	2019-01-01	10,000	500 (100%)

Table 2: CharBot data set's unregistered domains by seed.

For the experiments noted above, an 80%/10%/10% train/validation/test split was used. This split methodology involves training on the training data, minimizing model loss on the validation data, and reporting results as calculated using the testing data. An additional 100,000 pieces of CharBot data were added to the data sets where CharBot training was used, and testing for CharBot was performed on a

separate 100,000 piece data set.

Classifier	Data set	FPR=0.001 CB	FPR=0.01 CB
LSTM.MI	AlexaBamb	5.58%	15.50%
	AlexaBamb + CharBot	55.19%	81.08%
	QnameBamb	15.25%	31.90%
	QnameBamb + CharBot	52.67%	81.96%

Table 3: CharBot detection rates.

These initial findings are important for several reasons. Given that CharBot is such a simple DGA, its ability to fool state-of-the-art classifiers suggests several areas for further research. First, it is important to be able to train a classifier to detect CharBot with a high rate of accuracy. Second, achieving the first goal should not negatively affect a classifier's ability to detect other DGA families to any significant degree.

Classifier	Data set	FPR=0.001		FPR=0.01	
		TPR	AUC	TPR	AUC
LSTM.MI	AlexaBamb	96.79%	94.91%	99.27%	98.89%
	AlexaBamb + CharBot	95.50%	95.35%	98.89%	98.67%
	QnameBamb	81.98%	83.37%	98.98%	96.68%
	QnameBamb + CharBot	82.91%	84.98%	98.51%	96.48%

Table 4: Performance metrics of LSTM.MI on various datatsets.

4 Methodology

4.1 White-Box Adversarial Training

As noted in [17], adversarially training a machine learning classifier can be done in many different ways. [17] utilized black-box adversarial example generation, where malicious domains are generated and trained on, irrespective of the model's current weights. On the other hand, white-box adversarial training is often seen as a more powerful training technique, as it considers the model's trained parameters to generate examples that maximize the model's chance to misclassify. In this section, I outline how I converted our DGA training technique to utilize CharBot domains that were generated to the model's parameters. The hope was that this technique would increase the likelihood of the models to be able to detect CharBot generated domains without significantly harming their ability to detect other DGA domains.

The production of white-box adversarial examples was done in-between consecutive training epochs. I utilized a technique similar to the one described in [10], albeit with a few modifications, to better function in the domain of DGA detection. In essence, benign domain names were taken as inputs, all character perturbations generated, and surrogate loss values calculated as a way to quantify the most adversarial perturbations. For each benign input domain, the two most adversarial perturbations that occurred at different character indices were combined to create the CharBot domain that was the most difficult for the model to classify.

The training was completed by creating several generations of classifiers. The first generation will train only on its associated data set, namely AlexaBamb or Qname-

Bamb, which will be split 80% training, 10% validation, and 10% testing. For each successive generation, a subset of the benign domains was modified to create adversarial examples; these replaced malicious domains from the original data set. The model will be validated for accuracy and loss after each generation to determine the optimal number of adversarial training generations (i.e. where the validation loss is minimized). CharBot detection rates were tested after each epoch as well, but were not used to determine when to cease training.

This method for creating and training on white-box adversarial examples is mathematically similar to one outlined in [7] as a minimax problem. Essentially, the generation of white-box adversarial examples is the maximization of loss given the current model, and the training process is the minimization of loss given the adversarial examples. They describe this relationship with the following equation:

$$\theta^\star = \underset{\theta}{\operatorname{argmin}} \underset{(x,y)\sim\mathcal{D}}{\mathbb{E}} \left[\max_{\delta\in\Delta} L(\theta, x+\delta, y) \right].$$

It is worth noting that the question of when to stop training when adversarial examples are substituted is an open question. The validation metrics are calculated off of the standard, non-perturbed data set and are thereby reflective of the model's ability to classify between those benign/malicious domains rather than its ability to detect CharBot, which is what the adversarial examples are trying to improve.

4.1.1 Gradient Generation

TensorFlow[1], the back-end used for our neural networks, offers the ability to calculate model gradients automatically; however, this requires the ability to differentiate between all layers of the model. The neural networks used to classify domains utilize an embedding layer, which is non-differentiable. Therefore, a methodology had to be devised to estimate the model's gradient.

This was done in four steps: pre-training the machine learning model for a single epoch, testing m benign domains against the model, multiplying them by the one-hot representation of their inputs, and calculating an element-wise average without regards to zero; this methodology is expanded upon below. Similar gradient estimation methodologies have been implemented by FoolBox[2], but none are directly utilizable due to the presence of the embedding layer and because the white-box adversarial technique being implemented requires one-hot representations of character perturbations.

Neural network's weights are initialized to random values which prevents the generation of adversarials before the model has been trained. By performing an initial, single epoch of training we ensure the weights are partially tuned for the classification task being done.

The m domains taken are then used to estimate backpropagation through the neural network. Numerically embedded representations of these domains are passed through the neural network, and the resulting loss recorded for each. This scalar loss is then

[1] `https://github.com/tensorflow/tensorflow`
[2] `https://github.com/bethgelab/foolbox`

multiplied by the one-hot representation of the input domain, resulting in a two-dimensional array of zeros and the domain's loss value.

The next step is to compute the dot-product of the losses and the one-hot encoded domains, as shown in figure 1, which is followed by columnar averaging. However, due to the fact that domains are of varying lengths, only non-zero values are used for this averaging. This results in a two-dimensional array where the values represent the average loss for each character at each index regardless of the domain's overall input length. Domains are left-padded based on the maximum length, as can be seen in figure 2, and a failure to use this weighted average will result in lower values at smaller indices; this would be problematic as most domains have fewer characters than the maximum allowed by the DNS protocol. This weighted average is described in the summation shown in algorithm 2.

Algorithm 2: GradientGenerator

Data: a list of benign domains B, a trained model M, the number of possible characters C, a maximum domain length L

Result: An estimated gradient vector of length L

$X_{L,C} \leftarrow$ a matrix of size L x C
for *each benign domain* b_i *in* B **do**
    Set $M_{lb} \leftarrow M$'s scalar loss with respect to b
    Set $b_{oh} \leftarrow b$'s one-hot embedding of length L
    Set $b_{gv} \leftarrow M_{lb} b_{oh}$
    Set $X_{i,*} \leftarrow bgv$
end
$V \leftarrow \sum_{c=1}^{C} (X_{L,c}/(1 - \delta_{X_{L,c},0}))$
return V

Figure 1: Loss dot product

$$\begin{bmatrix} L_0, L_1, \ldots, L_d \end{bmatrix} \cdot \begin{bmatrix} oh_{i_0 j_0} & oh_{i_0 j_1} & \ldots & oh_{i_0 j_v} \\ oh_{i_1 j_0} & oh_{i_1 j_1} & \ldots & oh_{i_1 j_v} \\ \vdots & \vdots & \ddots & \vdots \\ oh_{i_d j_0} & oh_{i_d j_1} & \ldots & oh_{i_d j_v} \end{bmatrix}$$

where:
c = Max length of one-hot encoded domain
d = Number of domains
L = Scalar loss for each domain$\in$d

4.1.2 Perturbations and Vector Differences

Character perturbations are represented by the difference between the input's one-hot representation and the perturbed input's one-hot representation. This will result in a single value of 1, a single value of -1, and the remaining bits represented by zeros.

Algorithm 3: PerturbationGenerator

Data: an input domain D, a list of DNS-valid characters V
Result: a list of all valid domain perturbations P

```
P ← an empty list
for each character d_i in D do
    for each character v_j in V do
        if i ∉ {0, |D| − 1} then
            current ← D
            current_i = v_j
            Append current to P
        end
    end
end
return P
```

Figure 2: Numeric embedding of domains

washington.edu

$$\begin{bmatrix}12 & 1 & 9 & 5 & 6 & 7 & 4 & 10 & 8 & 7 & 13 & 3 & 2 & 11\end{bmatrix}$$

uw.edu

$$\begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 11 & 12 & 13 & 3 & 2 & 11\end{bmatrix}$$

Figure 3: One-hot embedding of domains

$$\begin{array}{c}w\\a\\s\\h\\i\\n\\g\\t\\o\\n\\.\\e\\d\\u\end{array}\begin{bmatrix}0&0&0&0&0&0&0&0&0&0&0&0&1&0\\0&1&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&1&0&0&0&0\\0&0&0&0&0&1&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&1&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1&0&0&0&0&0&0\\0&0&0&0&1&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&1&0&0&0\\0&0&0&0&0&0&0&0&1&0&0&0&0&0\\0&0&0&0&0&0&0&1&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0&1\\0&0&0&1&0&0&0&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&1&0&0\end{bmatrix}$$

$$\begin{array}{c}\\\\\\\\\\\\\\\\\\u\\w\\.\\e\\d\\u\end{array}\begin{bmatrix}1&0&0&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0&1\\0&0&0&1&0&0&0&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&1&0&0\end{bmatrix}$$

Algorithm 4: DifferenceGenerator

Data: an input domain D, a perturbed domain P
Result: a difference vector $\vec{V}$
1 $\vec{D_{oh}} \leftarrow$ one-hot embedding of D
2 $\vec{P_{oh}} \leftarrow$ one-hot embedding of P
3 $\vec{V} \leftarrow \vec{P_{oh}} - \vec{D_{oh}}$
4 **return** $\vec{V}$

4.1.3 Loss Estimation

The loss estimation for each perturbation can be estimated by calculating the cross multiplication between each difference vector and the estimated gradient. The largest resulting value is estimated to have the highest loss and considered to be the hardest for the model to classify, making it the best adversarial example. This method was suggested in [10] as a form of surrogate loss, as calculating the brute-force loss values for every perturbation (or pair of perturbations) for each domain would require an infeasible number of forward passes through the neural network.

As noted above, the most efficient way to estimate multiple perturbations to the same input is to select the best perturbation and then choose additional perturbations that occur at previously unused indices.

4.1.4 Adversarial Generation

In our methodology, the generation of adversarial examples has four steps: gradient generation, difference vector calculation, difference vector loss estimation, and adversarial selection. The last three steps are very similar to those outlined in [10], albeit in a different research domain, while the first step is only required due to the

embedding layer reasons outlined above.

Algorithm 5: AdversarialGenerator

Data: a gradient vector $\vec{G}_{1xm}$, a benign input domain D, a set of DNS valid characters C, a number of character perturbations N s.t. $N < |D|$

Result: an adversarial domain A with N perturbations of D

$A \leftarrow D$
$P \leftarrow$ a list of all valid perturbations of D as defined by Algorithm 3
$P_d \leftarrow$ a list of all difference vectors for P as defined by Algorithm 4
$P_l \leftarrow \vec{p_d} * \vec{G}$ for every $\vec{p_d}$ in P_d
for $n \leftarrow$ *0 to* N **do**
    $S_{c,i} \leftarrow max(P_l)_{character,index}$
    $A_i \leftarrow S_c$
    $P_l \leftarrow P_l$ where $P_i \neq S_c$
end
return A

4.1.5 Number of Perturbations

Representing multiple perturbations can be accomplished by including an additional pair of (1, -1) values in the difference vectors produced by Algorithm 4 for each additional perturbation performed. The runtime for calculating the number of perturbations is:

$$O(DC^S) \tag{1}$$

where:

D = Number of input domains

C = Number of valid characters

S = Number of character perturbations

However, by iteratively making perturbations – that is, by choosing the best perturbation followed by the next-best perturbation that occurs at a different character index - the runtime for generating difference vectors can be reduced to:

$$O(DCS) \tag{2}$$

The CharBot paper utilized two-character perturbations per input domain to maximize the likelihood that a domain "tricks" the classifier while minimizing the likelihood that a domain would already be registered. The latter is important as a DGA is functionally useless if all the domains it generates have already been registered by legitimate means.

After the publication of [18] some of our future experimentation increased the number of flips to $\lfloor \frac{D}{2} \rfloor$, as was shown successful when used in conjunction with saliency map attacks.

4.1.6 Training

We initialized training by performing a single epoch of training on the default data set. The neural network's weights are initially randomized, and by doing this, we achieve weights that can be used for semi-accurate inference and subsequently adversarial generations. Various number of domains, m, used in the gradient estimation process as outlined in 4.1.1, were tried; two values, 1,000 and 10,000, were utilized in our full experimentation. Increasing the sample size beyond 10,000 did not appear to improve the results.

The estimated gradient was then used to replace a subset of the malicious input domains, and training on this modified data set proceeded for a single epoch. The process was then repeated until validation loss was minimized.

4.1.7 Single vs. Multiple Gradients

The initial implementation of the adversarial generator utilized only a single gradient per epoch. Assuming a large enough sample is used to estimate the gradient, additional gradient generations will not be significantly different until the model's weights have been altered from additional training. However, it is possible to create additional gradients for each subsequent adversarial perturbation.

This is accomplished by recording the first perturbation's perturbation-index and perturbation-character followed by averaging a secondary gradient from input examples that contain the perturbation-character at the perturbation-index. In this way, the secondary gradient, and consequentially the second perturbation, will be formed in relation to the first perturbation rather than in relation to all possible inputs. This process can be extended for subsequent character perturbations, averaging a new gradient from only inputs that contain all previous perturbation values.

This process was accomplished by maintaining the matrix result of figure 1, taking the subset of rows that feature the correct character at the correct index, and computing the mean as done before.

4.1.8 Adversarial vs. Random Indices

The gradient methodologies above can be used to select the most adversarial character perturbations, but this method may result in the repetitive selection of the same perturbations. This can be avoided by choosing random indices to make perturbations while allowing the adversarial gradient to choose the perturbation character. This results in a larger array of different character perturbations, which should consequently help to keep the character distribution more similar to non-perturbation data sets and greatly reduces the runtime of the generation step.

4.1.9 Minibatch

Minibatch training involves splitting each training epoch into k sub-epochs and, in the case of adversarial attacks, generating $\frac{1}{k}$ adversarials per sub-epoch. This methodology allows an updated gradient to be generated after each sub-epoch in the hopes that it will more accurately reflect the portions of the neural network that are susceptible to CharBot style character perturbation evasions.

4.1.10 Blackbox

The method utilized in [17] to increase CharBot detection rates was to replace a subset of the negative training samples with CharBot domains. This method was utilized in conjunction with our adversarial examples in an attempt to bolster CharBot detection rates.

4.1.11 Contemporaneous Work

During the research phase of this thesis [18] was published online. The authors were successful in using a saliency map attack as implemented in the CleverHans[3] adversarial attack library to protect against a perturbation attack similar to that of CharBot.

Using this method in conjunction with our models would be difficult due to the inability to differentiate through the embedding layer, which was the main difficulty in our current model; consequently, some form of gradient estimation would again need to take place. The authors also noted the HotFlip methodology from [10] would be unlikely to work in the domain of DGA adversarial generation due to its focus on estimation and computational speed rather than accuracy.

4.2 Inductive Venn-Abers Predictors (IVAP)

As their name implies, binary classifiers offer inferences for whether an input belongs to one of two possible classes. Inductive Venn-Abers Predictors (IVAP) function as a wrapper for trained classifiers that offer an additional metric of prediction reliability. In essence, both positive and negative predictions of a binary classifier will also have to pass an additional level of statistical acceptance criteria. This results in four possible outcomes, where [0, 1] describe the inference labels: 0-accepted, 0-rejected, 1-accepted, and 1-rejected.

The expected benefit of this methodology is an increase in predictive measures at

[3] `https://github.com/tensorflow/cleverhans`

the cost of a decreased training data set and the small number of inputs being left unlabeled after the application of IVAP.

The IVAP training process uses data set splits that are atypical: 64% training, 4% validation, 16% calibration, and 16% testing. I trained these models for a maximum of 100 epochs, with early stopping being applied after 10 epochs based on loss as calculated against the validation data set. The four data splits are used as follows:

1. Training: to train the neural network
2. Validation: to stop neural network training on loss minimization; to tune the β value shown in equation (4)
3. Calibration: to calibrate the model's inference reliability as shown in equation (3)
4. Testing: to calculate results

Following training, the model is calibrated using the calibration data set such that any tested input will be associated with two prediction probabilities as outlined below:

$$p_0 \leq \Pr[Y = 1 \mid X = x] \leq p_1 \tag{3}$$

The difference between these lower and upper bound probabilities can be used to quantify whether or not a prediction should be rejected. However, to do so, we need to identify an acceptance threshold β such that:

$$p_1 - p_0 \leq \beta \tag{4}$$

This β value is calculated by using the calibrations calculated above and the validation data set that was previously used to prevent overfitting during the training process. In both [16] and this thesis the β value is tuned to maximize the difference between the true rejection rate and the false rejection rate, although other metrics may be preferable in other domains of research.

The IVAP process is done using five algorithms as outlined in [20], the unabridged and continuously updated version of [2].

4.2.1 IVAP Adversarial Training

The methodology and experiments conducted above were published in [16] but did not quantify the hardening of DGA classifiers against perturbation based adversarial attacks.

Additional experiments were run by combining the BlackBox methodology outlined in [17] with the IVAP methodologies utilized in [16]). The QnameBamb data was trained on using the 64% training, 4% validation, 16% calibration, and 16% testing protocol outlined above, but with 20% of the training data replaced with CharBot perturbed domains at each epoch.

Experiments were run using Invincea and LSTM.MI models over five fold cross validation with additional testing performed on CharBot data sets.

5 Conclusion and Future Work

5.1 White-Box Adversarial Generation

The techniques outlined in 4.1 were all implemented successfully but ultimately yielded little success, as [18] suggested would be the case in their contemporaneous work. The best results were generated with the following parameters:

1. Baseline:
    - 100,000 BlackBox as shown in [17]
2. Test i:
    - 100,000 BlackBox + 100,000 Adversarial
    - Adversarial selected indices and characters
    - Secondary gradient produced based on first perturbation
    - 10,000 sample first gradient
3. Test ii:
    - 100,000 BlackBox + 100,000 Adversarial
    - Adversarial selected indices and characters
    - 1,000 sample gradient
4. Test iii:
    - 100,000 BlackBox + 100,000 Adversarial

- Randomly selected indices, adversarially selected characters
- 1,000 sample gradient

Table 5: White Box Adversarial Training Results

Classifier	Parameters	FPR=0.01		FPR=0.001		
		TPR	AUC	TPR	AUC	CB Det
LSTM.MI	Baseline	98.83%	98.78%	92.26%	96.03%	82.70%
	Test i	98.81%	98.75%	95.74%	96.08%	87.16%
	Test ii	98.93%	98.83%	96.06%	96.27%	82.92%
	Test iii	98.78%	98.66%	95.55%	95.61%	82.52%

TPR=True Positive Rate
AUC=Area Under the ROC Curve
FPR=False Positive Rate
CB Det=CharBot Detection Rate

The methodologies described in [18] could serve as a good starting point for future work. Their work utilized one-hot encoding, as shown in section 4.1.2, which would increase the training runtime significantly over the numerical embedding we use as shown in figure 2; testing the results of the neural network architectures shown in section 6, when trained on one-hot embedded domains as well as replicating the results shown in [18], would be necessary initial steps. Following this, work could branch into two areas. First, an analysis of the performance of [18]'s gradient when used in conjunction with the rest of our methodology described in 4.1.4 would offer insight into whether our process has merit beyond its computational efficiency. Second, an analysis of how successful our gradient estimation, as described in 4.1.1, is when utilizing the saliency map attack implemented in CleverHans[4] which is what was utilized in [18]. This analysis would be non-trivial in nature, requiring extensive

[4] `https://github.com/tensorflow/cleverhans`

code refactoring in order to produce gradients usable by CleverHans.

An additional area of research would involve a quantification comparison between adversarial example's loss estimation, as described in 4.1.3, with the model's true loss. While the usage of the model's true loss for each adversarial generation is computationally prohibitive, using a subset of generated adversarials for comparisons could offer us insight into the loss estimation's success. It should be noted that the described surrogate loss estimation method is not intended to estimate the loss values calculated by the model, but rather to produce loss values whose order ranking is similar, as we are only using this value to choose the most adversarial example.

5.2 IVAP

The results for the implemented IVAP methodology were published in [16] and are shown in Table 6. Most of these results are in line with what was expected to be seen as per [2], although the MIT did have problematic results after the application of IVAP. In all other cases the model's true positive rate and accuracy increased while maintaining a suitably low false positive rate.

The differences in results are likely dependent on two factors: data set and model architecture, or some combination of the two.

As was discussed in 2.2, the AlexaBamb data set is likely to feature less data noise than the QnameBamb data set. While DGA classifiers have been successfully trained on noisy data, as was done in [9], additional work needs to be done to determine if data noise affects the IVAP process. The construction of the calibrations and the

Table 6: Comparisons of model performance with and without IVAP

Data Set	Classifier	TPR	FPR	ACC	FRR	TRR	REJ
QnameBamb	LSTM.MI	0.917	0.001	0.993	-	-	-
	LSTM.MI + IVAP	0.993	0.001	0.996	0.064	0.922	0.100
	Invincea	0.856	0.001	0.991	-	-	-
	Invincea + IVAP	0.986	0.001	0.993	0.126	0.916	0.183
	MIT	0.920	0.001	0.993	-	-	-
	MIT + IVAP	-	-	-	-	-	1.000
AlexaBamb	LSTM.MI	0.964	0.001	0.992	-	-	-
	LSTM.MI + IVAP	0.999	0.001	0.999	0.066	0.950	0.082
	Invincea	0.966	0.001	0.991	-	-	-
	Invincea + IVAP	0.999	<0.001	0.999	0.108	0.969	0.123
	MIT	0.909	0.001	0.992	-	-	-
	MIT + IVAP	0.973	<0.001	0.988	0.286	0.493	0.290

FPR=False Positive Rate, ACC=Accuracy, TPR=True Positive Rate
TRR=True Rejection Rate, FRR=False Rejection Rate, REJ=Rejection Rate
Published in [16]

Table 7: Model performance/CharBot detection with and without IVAP

Data Set	Classifier/Method	TPR	FPR	ACC	FRR	TRR	REJ
QnameBamb	LSTM.MI Baseline	0.898	0.001	0.986	-	-	-
	+ IVAP	0.999	0.001	0.999	0.112	0.981	0.156
	LSTM.MI CharBot	0.450	-	0.450	-	-	-
	+ IVAP	0.920	-	0.920	0.088	0.824	0.321
	Invincea Baseline	0.871	0.001	0.984	-	-	-
	+ IVAP	0.995	0.001	0.997	0.149	0.963	0.201
	Invincea CharBot	0.436	-	0.436	-	-	-
	+ IVAP	0.919	-	0.919	0.079	0.836	0.334

FPR=False Positive Rate, ACC=Accuracy, TPR=True Positive Rate
TRR=True Rejection Rate, FRR=False Rejection Rate, REJ=Rejection Rate

calculation of the β value as discussed in 4.2 are of particular interest.

The concept of data poisoning has been well studied in the field of adversarial machine learning, and many of these same methods could be used in such an exploration. Taking a more accurately marked data set, such as AlexaBamb, and swapping a small number of labels would be a good start. This method could be used to quantify at what point incorrectly labeled data begins to degrade the IVAP process, and since IVAP is a wrapper process, quantifications could be performed under otherwise identical circumstances.

These label swaps could be performed on both the calibration data set during the calibration step and on the validation data set after validation but before β calculation.

The results of IVAP on adversarial DGA detection using the methodology described in 4.2.1 can be seen in Table 7. The results show an inability for neural network

models to detect CharBot domains before the application of IVAP, but a reasonably high detection rate once IVAP was applied.

As was suggested in [2] and shown to be true in the domain of DGA classification in [16], the primary caveat for this methodology is the rejection rate. It is worth noting that CharBot domains were rejected at a significantly higher rate than QnameBamb domains, which logically follows due to their difficulty in being detected by traditionally trained DGA detection neural networks.

Future work in regards to the detection of perturbation based DGAs utilizing IVAP is fairly open-ended. Initial work could involve the training of additional neural network models and comparisons between various data sets. Since the training methodology described in 4.2.1 involves the utilization of BlackBox generated CharBot domains, it would also be useful to hyper-tune the ideal number of perturbed domains as well as increase the number of perturbations per domain as was done in [18].

6 Code Appendix

6.1 Invincea

As described in [8].

```
def getconvmodel(self, kernel_size, filters):
    model = Sequential()
    model.add(Conv1D(filters=filters, input_shape=(128, 128),
        kernel_size=kernel_size, padding='same', activation='relu', strides=1))
    model.add(Lambda(lambda x: K.sum(x, axis=1), output_shape=(filters, )))
    model.add(Dropout(0.5))
    return model


main_input = Input(shape=(75, ), dtype='int32', name='main_input')
embedding = Embedding(input_dim=128, output_dim=128,
conv1 = getconvmodel(2, 256)(embedding)
conv2 = getconvmodel(3, 256)(embedding)
conv3 = getconvmodel(4, 256)(embedding)
conv4 = getconvmodel(5, 256)(embedding)
merged = Concatenate()([conv1, conv2, conv3, conv4])
middle = Dense(1024, activation='relu')(merged)
middle = Dropout(0.5)(middle)
middle = Dense(1024, activation='relu')(middle)
middle = Dropout(0.5)(middle)
```

```
output = Dense(1, activation='sigmoid')(middle)
model = Model(inputs=main_input, outputs=output)
model.compile(loss='binary_crossentropy', optimizer='adam')
```

6.2 LSTM.MI

As described in [12].

```
model = Sequential()
model.add(Embedding(max_features, 128, input_length=maxlen))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='rmsprop')
```

6.3 MIT

As described in [5].

```
main_input = Input(shape=(75, ), dtype='int32', name ='main_input')
embedding = Embedding(input_dim=128, output_dim=128,
    input_length=75)(main_input)
conv = Conv1D(filters=128, kernel_size=3, padding='same',
    activation='relu', strides=1)(embedding)
max_pool = MaxPooling1D(pool_size=2, padding='same') (conv)
```

```
encode = LSTM(64, return_sequences=False)(max_pool)
output = Dense(1, activation='sigmoid')(encode)
model = Model(inputs=main_input, outputs=output)
model.compile(loss='binary_crossentropy', optimizer='adam')
```

References

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," eng, *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997, ISSN: 0899-7667.

[2] V. Vovk, I. Petej, and V. Fedorova, "Large-scale probabilistic predictors with and without guarantees of validity," in *Advances in Neural Information Processing Systems 28*, 2015, pp. 892–900.

[3] H. S. Anderson, J. Woodbridge, and B. Filar, "DeepDGA: Adversarially-tuned domain generation and detection," in *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, ser. AISec '16, Vienna, Austria: ACM, 2016, pp. 13–21, ISBN: 978-1-4503-4573-6. DOI: 10.1145/2996758.2996767. [Online]. Available: http://doi.acm.org/10.1145/2996758.2996767.

[4] D. Plohmann, K. Yakdan, M. Klatt, J. Bader, and E. Gerhards-Padilla, "A comprehensive measurement study of domain generating malware," Presented at USENIX Security Symposium, Austin, TX, USA, 2016. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_plohmann.pdf.

[5] S. Vosoughi, P. Vijayaraghavan, and D. Roy, "Tweet2Vec: Learning tweet embeddings using character-level CNN-LSTM encoder-decoder," in *Proceedings of*

the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval, 2016, pp. 1041–1044.

[6] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 39–57. DOI: 10.1109/SP.2017.49.

[7] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[8] J. Saxe and K. Berlin, "Expose: A character-level convolutional neural network with embeddings for detecting malicious urls, file paths and registry keys," *CoRR*, vol. abs/1702.08568, 2017. arXiv: 1702.08568. [Online]. Available: http://arxiv.org/abs/1702.08568.

[9] B. Yu, D. L. Gray, J. Pan, M. De Cock, and A. C. Nascimento, "Inline DGA detection with deep networks," in *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, IEEE, 2017, pp. 683–692.

[10] J. Ebrahimi, A. Rao, D. Lowd, and D. Dou, "Hotflip: White-box adversarial examples for text classification," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Melbourne, Australia: Association for Computational Linguistics, 2018, pp. 31–36. [Online]. Available: http://aclweb.org/anthology/P18-2006.

[11] S. Schüppen, D. Teubert, P. Herrmann, and U. Meyer, "FANCI: Feature-based automated NXDomain classification and intelligence," in *USENIX Security*

Symposium, 2018, pp. 1165–1181. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/schuppen.

[12] D. Tran, H. Mac, V. Tong, H. A. Tran, and L. G. Nguyen, "A LSTM based framework for handling multiclass imbalance in DGA botnet detection," *Neurocomputing*, vol. 275, pp. 2401 –2413, 2018, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2017.11.018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231217317320.

[13] Y. Vorobeychik and M. Kantarcioglu, "Adversarial machine learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 12, no. 3, pp. 1–169, 2018.

[14] B. Yu, J. Pan, J. Hu, A. Nascimento, and M. De Cock, "Character level based detection of DGA domain names," in *2018 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2018, pp. 4168–4175.

[15] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. Goodfellow, A. Madry, and A. Kurakin, "On evaluating adversarial robustness," *arXiv preprint arXiv:1902.06705*, 2019.

[16] C. Grumer, J. Peck, F. Olumofin, A. Nascimento, and M. De Cock, "Hardening DGA classifiers utilizing IVAP," in *2019 IEEE BigData*, IEEE, 2019.

[17] J. Peck, C. Nie, R. Sivaguru, C. Grumer, F. G. Olumofin, B. Yu, A. C. A. Nascimento, and M. D. Cock, "CharBot: A simple and effective method for evading DGA classifiers," vol. 7, 2019, pp. 91 759–91 771.

[18] L. Sidi, A. Nadler, and A. Shabtai, *MaskDGA: A black-box evasion technique against DGA classifiers and adversarial defenses*, 2019. arXiv: 1902.08909 [cs.CR].

[19] J. Spooren, D. Preuveneers, L. Desmet, P. Janssen, and W. Joosen, "Detection of algorithmically generated domain names used by botnets: A dual arms race.," Association for Computing Machinery, 2019, pp. 1902–1910, ISBN: 978-1-4503-5933-7.

[20] V. Vovk, I. Petej, and V. Fedorova, "Large-scale probabilistic prediction with and without validity guarantees." [Online]. Available: http://alrw.net/articles/13.pdf.