DGA Bot Detection with Time Series Decision Trees

Anaël Bonneton
INRIA & ANSSI
anael.bonneton@ssi.gouv.fr

Daniel Migault
Ericsson Security Research
daniel.migault@ericsson.com

Stephane Senecal and Nizar Kheir
Orange Labs
firstname.lastname@orange.com

***Abstract*—This paper introduces a behavioral model for botnet detection that leverages the Domain Name System (DNS) traffic in large Internet Service Provider (ISP) networks. More particularly, we are interested in botnets that locate and connect to their command and control servers thanks to Domain Generation Algorithms (DGAs). We demonstrate that the DNS traffic generated by hosts belonging to a DGA botnet exhibits discriminative temporal patterns. We show how to build decision tree classifiers to recognize these patterns in very little computation time. The main contribution of this paper is to consider whole time series to represent the temporal behavior of hosts instead of aggregated values computed from the time series. Our experiments are carried out on real world DNS traffic collected from a large ISP.**

I. Introduction

The Domain Name System (DNS) is used by millions of users to locate and access resources on the Internet. It is a key component of the Internet which enables access to resources such as web servers, hosting, content sharing and mailing services. Unfortunately, the use of DNS is not limited to benign browsing activities. Modern cybercrime increasingly relies on networks of infected hosts, commonly known as *botnets*. They are managed by remote attackers who perform attacks such as denial of service, phishing, email spamming, and data theft [1]. Infected hosts - *bots* - must be able to locate and communicate with the command and control (C&C) servers set by the attackers. The link between the bots and the C&C servers is considered as the Achilles' heel of botnets as without it bots are useless for the attackers. This weakness has been used in recent years for spectacular take-down operations in which the C&C servers were identified and neutralized: Conficker in 2009 [2], ZeroAccess in 2013 [3], and GameOver Zeus in 2014 [4].

Attackers may hide and protect the communication channel to the C&C servers using the DNS. A first technique, known as fast-flux [5], is to constantly assign new IP addresses to the same domain name associated with a C&C server. This technique protects against IP blacklisting but not against domain blacklisting. Thus, attackers introduced *Domain Generation Algorithms* (DGAs) [6], [7], [8] in order to prevent domain blacklisting. DGAs generate a list of domain names on each bot and renew it periodically. At a given point in time, only a few of these DGA-generated domains are actually registered and point to the IP addresses of C&C servers. Bots attempt to contact the hosts in the domain list until one matches a C&C server. Since only the DGA author knows when the upcoming rendezvous domain has to be registered and activated, the DGA technique allows attackers to make their botnet much more resilient against domain blacklisting. When combined with fast-flux it makes the botnet even more resilient against takeovers.

Existing DGA bot detection systems rely mostly on the structure and the syntax of the randomly generated domain names [6], [7]. They leverage the similarities in the length, entropy, and character subsequences of malicious domain names to detect them. In this paper, we propose a fully behavioral DGA bot detection model based on the temporal behavior of hosts that can strengthen state of the art bot detection systems. We demonstrate that the DNS traffic generated by DGA bots exhibit discriminative temporal patterns. We recognize these patterns using *time series decision trees*. We propose an open source implementation of time series decision trees [1] and we carry out experiments on real world DNS traffic collected from a large ISP.

As far as we know, it is the first time that whole DNS time series, and not only aggregated values (for instance mean, variance, quantiles,...) computed from the time series, are used to detect infected hosts. Considering whole time series allows to recognize generic temporal patterns, and not only periodic or pseudo-periodic patterns. Besides, even if our detection model relies on whole time series, the detection is still performed in very little time.

Our behavioral detection model relies on a supervised model (decision trees) and thus requires labeled data, i.e. examples of DNS traffic of infected and non infected hosts. The labels can be provided by various sources: black and white lists supplied by the community or by third party security editors, manual processing, or unsupervised machine learning techniques which usually require manual interpretation. Here, we use clustering, an unsupervised machine learning technique, to build the labels required by the supervised detection model.

II. System Overview

The behavioral detection model we propose detects "infected IPs" and not infected hosts as from our DNS data two hosts sharing the same IP address cannot be differentiated. Hereinafter, an IP address is considered infected if at least one host using this IP address belongs to a botnet.

We proceed with a three step process in order to detect infected IPs from DNS traffic data:

1) The *labeling module* relies on a clustering technique. It generates labeled data for the training module: it provides a list of infected IPs for each detected botnet. See section III.

[1] https://github.com/abonneton/TimeSeriesDecisionTrees

2) The *training module* applies supervised learning, using the labels generated by the labeling module, in order to build a detection model, a time series decision tree, for each botnet family. It leverages the typical DNS temporal profile of IPs that belong to a same botnet. See section IV.
3) The *detection module* uses the detection model provided by the training module in order to detect new infected IPs in very little time. See section IV.

The labeling module leverages similarities between non-existent domains in an ISP network in order to build clusters of infected IPs that belong to a same botnet. It is only used to create the labeled data necessary to the training module, as it suffers from a very high time complexity and requires some human intervention to interpret the clusters.

The training module builds a time series decision tree model able to distinguish `Infected` from `NonInfected` IPs for each botnet family. Each IP is modeled by a 24 hour time series representing the number of DNS answers it received along the day.

It is important to emphasize that we do not introduce bias as the labeling and the training modules are based on different features. The labeling module relies on the queried domain names and their corresponding DNS error codes, whereas the training module relies on the full DNS time series for each IP.

The behavioral model for botnet detection was tested using a one day traffic capture at the resolving DNS servers of a large ISP from 2009. The capture concerns roughly 7 millions IPs requesting 60 millions different domain names over 40 billions of queries.

III. Labeling Module: Clustering

A. Methodology

The labeling module provides examples of infected IPs to the supervised training module. It processes only non-existent domain requests which resulted with an `NXDOMAIN` error response from the DNS service. It implements a hierarchical clustering that leverages similarities between `NXDOMAIN` queries across multiple IPs in the network: IPs belonging to the same DGA botnet are likely to share a common subset of queries towards `NXDOMAIN`s.

Thus, the labeling module processes DNS traffic in order to build clusters of `NXDOMAIN`s generated by the same DGA. Since the output clusters may not be all associated with a DGA botnet activity, the labeling module adopts a semi-manual approach where an administrator annotates the clusters and eliminates the ones that do not convey any malicious activity. Once the malicious clusters of domain names have been identified, the lists of infected IPs are derived: an IP belongs to a DGA botnet if it has queried at least one domain name among the DGA cluster.

The labeling module only provides labels for the training process. It does not aim at grouping together *all IPs* that belong to the same botnet. Hence, we adopt conservative thresholds for the clustering process in order to eliminate false positives (i.e. IPs belonging to the same cluster while not associated with the same DGA botnet), while tolerating a higher rate of false negatives.

The labeling module operates through the following steps:

1) **Data Preprocessing** A set of rules and heuristics are applied to reduce the volume of input data for the clustering process. It eliminates obvious benign DNS traffic, and keeps only suspect `NXDOMAIN`s for further clustering. For instance, Top Level Domains (TLDs) such as `.local` and `.home` are discarded from the input dataset as they are mostly associated with misconfigured Internet services. Besides, other TLDs unlikely to be associated with malicious activities such as `.arpa` and `.gov`, as well as secondary level domains such as `google`, `facebook`, and `yahoo` are also discarded. We have also implemented several heuristics in order to filter out specific Internet services that use the DNS as a transport protocol but that do not convey any name resolution requests. For instance, Cymru [9] provides a reverse "whois" service based on IP addresses and SPAMHAUS [10][11] provides an SBL service that indicates whether an IP address is associated with a SPAM activity.
2) **Hierarchical Clustering** Let $\mathcal{D}$ be the set of domain names. Each domain $d \in \mathcal{D}$ is represented by IP^d, the set of IP addresses that issued at least one unsuccessful DNS request towards it. The set of clusters $\mathcal{C}$ represents a partition of $\mathcal{D}$. For each cluster $c \in \mathcal{C}$, let $IP^c = \cup_{d \in c} IP^d$ be the set of IP addresses which queried at least one domain name in the cluster c. The domain names are clustered with a bottom-up (agglomerative) approach. Initially, there is one cluster per domain. Then, the closest clusters according to the similarity measure sim (1), c_1^* and c_2^*, are merged if they fulfill the constraint (2). The constraint (2) prevents the clustering algorithm from merging two clusters which do not share enough IPs. The parameter α determines how close two clusters must be in order to be merged.

$$\forall (c_i, c_j) \in \mathcal{C}^2, \ \mathrm{sim}(c_i, c_j) = |IP^{c_i} \cap IP^{c_j}| \quad (1)$$

$$\mathrm{sim}(c_1^*, c_2^*) > \alpha \cdot \min(|IP^{c_1^*}|, |IP^{c_2^*}|) \quad (2)$$

The clustering algorithm is detailed in Algorithm 1.

Algorithm 1 Clustering algorithm to label the data

```
1: $\mathcal{C} \leftarrow \{\{d\}_{d \in \mathcal{D}}\}$
2: while $True$ do
3:     $(c_1^*, c_2^*) \leftarrow \underset{(c_1,c_2) \in \mathcal{C}^2}{\mathrm{argmin}} \ \mathrm{sim}(c_1, c_2)$
4:     if $\mathrm{sim}(c_1^*, c_2^*) > \alpha \min(|IP^{c_1^*}|, |IP^{c_2^*}|)$ then
5:         Merge $c_1^*$ and $c_2^*$ in $\mathcal{C}$
6:     else
7:         break
8:     end if
9: end while
10: Output: $\mathcal{C}$
```

3) **Deriving Lists of Infected IPs** Expert knowledge and third party data are used to discard clusters which are not related to malicious activities and to name each

Botnet	# Domain names	# Infected IPs
Conficker A	250	134
Conficker B	250	7333
Kraken	$>$ 54 billions	1504

TABLE I: Identified botnets

identified DGA botnet. The list IP^c of IP addresses related to each malicious cluster c is then provided as input for the training module.

The labeling module is very close to the detection system proposed in [12]. It also relies on a hierarchical clustering of non-existent domain names but it considers DNS traffic on authoritative servers whereas the labeling module uses DNS traffic on the resolvers of an ISP. Both systems suffer from a high time complexity and require some human intervention to interpret the clusters. That is why in our system the hierarchical clustering does not perform the final detection.

B. *Interpretation*

Thanks to the clustering algorithm we are able to label three well-known botnets as illustrated in table I: Conficker A, Conficker B and Kraken. As we work on data dating from 2009, the detected botnets have already been well analyzed: Conficker A and Conficker B in [13], [2], [14] and Kraken in [15], [16].

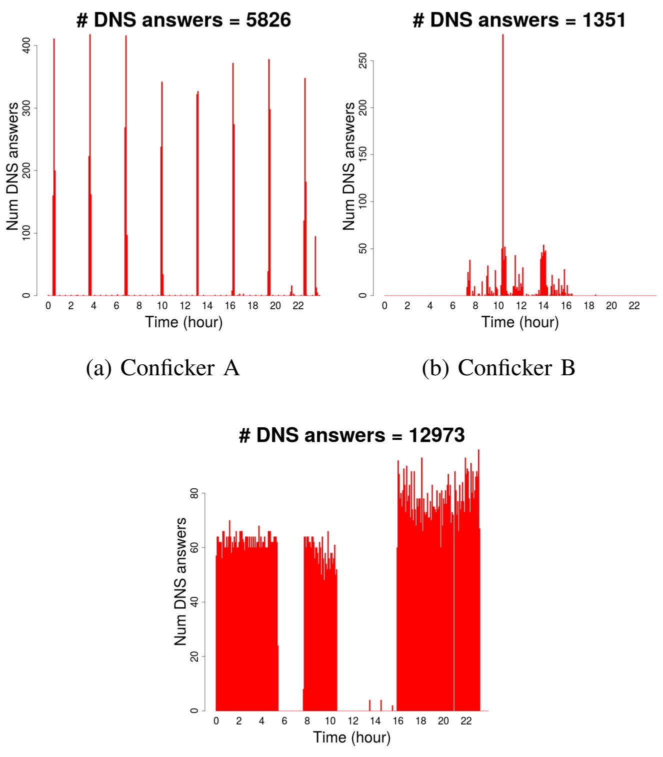
Unfortunately, with the traffic capture from 2009 we cannot prove that the labeling module is able to detect zero-day botnets. However, this section illustrates that the clustering approach we propose can by itself identify infected IPs without any prior knowledge.

Figure 1 shows that infected IPs have discriminative DNS time series. Indeed, this figure represents the DNS time series of one of the infected IP which depicts the typical temporal behavior of each identified botnet. Conficker A presents traffic bursts every 3 hours, Conficker B presents one burst every day. In contrast, Kraken presents a continuous heavy DNS traffic, with blanks of several hours. Such temporal patterns present significant characteristics that may be used to identify botnets. They will be taken into account in the training module described in the following section.

IV. Training and Detection Modules: Time Series Decision Trees

This section describes the core of our work: a classifier based on time series decision trees to detect IPs belonging to DGA botnets. The input of our classifier is the DNS time series $T \in \mathcal{T}$ associated with each IP address representing the number of DNS answers it received along the day. Besides, each IP address has a label denoted as $\mathrm{label}(T) \in \mathcal{L} = \{\mathtt{Infected}, \mathtt{NonInfected}\}$.

The time series decision trees we consider are based on an adaptation [17] of CART decision trees [18] for time series features. A CART decision tree is a classifier expressed as a recursive partition of the training data. It consists of a binary tree where the root node contains all the training instances.



(a) Conficker A

(b) Conficker B

(c) Kraken

Fig. 1: Typical DNS temporal behavior of infected IPs

Each internal node contains a set of training instances $\mathcal{T}$ and a *split condition* s based on the input features. It splits its training instances $\mathcal{T}$ into two subsets, $\mathcal{T}_{left}$ and $\mathcal{T}_{right}$ the instances of the left and right children, according to its split condition s. The split conditions for time series are defined in section IV-A.

A node is a leaf if its training instances $\mathcal{T}$ fulfill one of the base case conditions:

1) All the instances have the same label:
$$\forall (T_0, T_1) \in \mathcal{T}^2, \ \ \mathrm{label}(T_0) = \mathrm{label}(T_1) \qquad (3)$$

2) All the instances are equivalent according to a distance, dist, defined between input instances:
$$\forall (T_0, T_1) \in \mathcal{T}^2, \ \ \mathrm{dist}(T_0, T_1) = 0 \qquad (4)$$

The leaves are designated with the most represented label among its training instances.

a) Training Module: The process of building decision trees pertains to a recursive divide-and-conquer algorithm: one goes on splitting the nodes until all terminal nodes fulfill one of the base case constraints (3) or (4). The algorithm is detailed in Algorithm 2. A decision tree is built from the training instances $\mathcal{T}_{train}$ with BuildDecisionTree(root, $\mathcal{T}_{train}$).

For a given node associated with the set of training instances $\mathcal{T}$ we first compute a list of split candidates (see section IV-A). Then, the split $s^\star$ minimizing the child nodes

Algorithm 2 Recursive function to build a decision tree

```
1:  function BuildDecisionTree(node, $\mathcal{T}$)
2:      if BaseCase($\mathcal{T}$) then
3:          node $\leftarrow$ Leaf($\mathcal{T}$)
4:      else
5:          splits $\leftarrow$ SplitCandidates($\mathcal{T}$)
6:          $s^\star \leftarrow \underset{s \in splits}{\text{argmin}} \frac{|\mathcal{T}_{left}|}{|\mathcal{T}|} \text{H}(\mathcal{T}_{left}) + \frac{|\mathcal{T}_{right}|}{|\mathcal{T}|} \text{H}(\mathcal{T}_{right})$
7:          $(\mathcal{T}_{left}, \mathcal{T}_{right}) \leftarrow$ SplitInstances($\mathcal{T}$, $s^\star$)
8:          BuildDecisionTree(node.left_child, $\mathcal{T}_{left}$)
9:          BuildDecisionTree(node.right_child, $\mathcal{T}_{right}$)
10:     end if
11: end function
```

impurity is selected among the candidate splits. A node is said completely "pure" if all its instances have the same label. The impurity of a node is the largest when its labels are equally mixed together in it (in our case, 50% of `Infected` and 50% of `NonInfected` IPs). Several node impurity measures can be defined and used, in the following we consider the *entropy* (5).

$$\text{H}(\mathcal{T}) = -\sum_{l \in \mathcal{L}} p^l_{\mathcal{T}} \log_2(p^l_{\mathcal{T}}) \tag{5}$$

where $p^l_{\mathcal{T}}$ is the proportion of instances in $\mathcal{T}$ having the label l:

$$\forall l \in \mathcal{L},\ p^l_{\mathcal{T}} = \frac{|\{T \in \mathcal{T} / \text{label}(T) = l\}|}{|\mathcal{T}|}$$

b) Detection Module: Once such a decision tree is built, the class label for a new instance can be predicted by going through the tree from the root to a leaf according to the conditions at each split and the instance input variables. The label designating the final leaf is the predicted label for this new instance.

A. *Time Series Split Candidates*

This section presents two kinds of split candidates for time series proposed in [17]: the *standard split* and the *cluster split*. The way these splits partition a two-dimensional space is depicted in figure 2.

Let $\mathcal{T}$ be the set of training instances of a node we want to split and d a distance for time series.

- A *standard split*, denoted as $S(T_0, r)$ where $T_0 \in \mathcal{T}$ and $r \in \mathbb{R}^\star_+$, partitions $\mathcal{T}$ with a sphere of radius r centered in T_0 (6) :

$$\begin{aligned} \mathcal{T}_{left} &= \{T \in \mathcal{T};\ \text{d}(T, T_0) \le r\} \\ \mathcal{T}_{right} &= \{T \in \mathcal{T};\ \text{d}(T, T_0) > r\} \end{aligned} \tag{6}$$

- A *cluster split*, denoted as $C(T_0, T_1)$ where $(T_0, T_1) \in \mathcal{T}^2$, selects two time series T_0 and T_1 and partitions the time series in $\mathcal{T}$ according to their distance from T_0 and T_1 (7) :

$$\begin{aligned} \mathcal{T}_{left} &= \{T \in \mathcal{T};\ \text{d}(T, T_0) \le \text{d}(T, T_1)\} \\ \mathcal{T}_{right} &= \{T \in \mathcal{T};\ \text{d}(T, T_0) > \text{d}(T, T_1)\} \end{aligned} \tag{7}$$

In [19], the authors have worked on heuristics and on various distances for time series, in order to improve the time series decision trees developed in [17]. They propose to constrain cluster splits with T_0 and T_1 having different labels in order to reduce the search space of the best cluster split and so to reduce the decision tree building time complexity. Hereinafter, the cluster split without any additional constraint will be called *plain cluster split* (denoted as `PC`), and the one with the additional constraint *constrained cluster split* (denoted as `CC`).

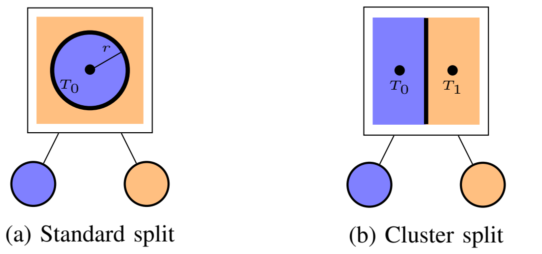


Fig. 2: A two-dimensional representation of the splits

Several SplitCandidates functions (line 5 of Algorithm 2) are defined from the split definitions. First, (8), (9), and (10) present SplitCandidates functions which lead to homogenous decision trees, i.e. decision trees composed exclusively of either standard splits or cluster splits.

$$Splits_S(\mathcal{T}) = \left\{S(T_0, r)\ /\ T_0 \in \mathcal{T},\ r \in R^{\mathcal{T}}_{T_0}\right\} \tag{8}$$

where

$$R^{\mathcal{T}}_{T_0} = \{\text{d}(T_0, T)\ /\ T \in \mathcal{T} - \{T_0\}\}$$

$$Splits_{PC}(\mathcal{T}) = \left\{C(T_0, T_1)\ /\ (T_0, T_1) \in \mathcal{T}^2\right\} \tag{9}$$

$$\begin{aligned} &Splits_{CC}(\mathcal{T}) = \{C(T_0, T_1)\ / \\ &(T_0, T_1) \in \mathcal{T}^2,\ \ \text{label}(T_0) \ne \text{label}(T_1)\} \end{aligned} \tag{10}$$

Then, SplitCandidates functions which lead to heterogeneous decision trees, (11) and (12), are derived from (8), (9), and (10).

$$Splits_{S\_PC}(\mathcal{T}) = Splits_S(\mathcal{T}) \cup Splits_{PC}(\mathcal{T}) \tag{11}$$

$$Splits_{S\_CC}(\mathcal{T}) = Splits_S(\mathcal{T}) \cup Splits_{CC}(\mathcal{T}) \tag{12}$$

These five SplitCandidates functions will be considered for the experiments in section V.

B. *Time Series Distances*

Building decision trees for time series requires to define a distance for such input data. The Euclidean distance is an alternative as the DNS time series in the dataset have the same length: they are sampled at the same rate and they all have the same duration. However, this distance is very brittle as it does not allow similar shapes to match if they are out of phase.

In order to illustrate the problem caused by the Euclidean distance, figure 3 introduces three times series that can be viewed as highly similar for the botnet detection problem as they could represent three bots belonging to the same botnet. Time series T_1 and T_2 have exactly the same period (6 hours) but are not synchronized. Time series T_3 has a pseudo-period of value $6h \pm \varepsilon \cdot 1h$ where $\varepsilon \in [0, 1]$ is a random variable. Such

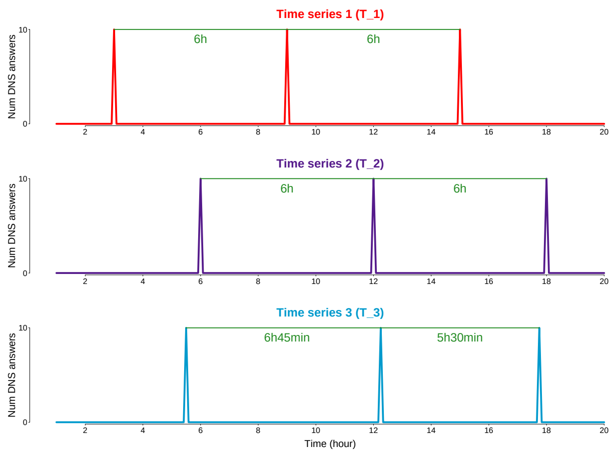


Fig. 3: Three similar time series

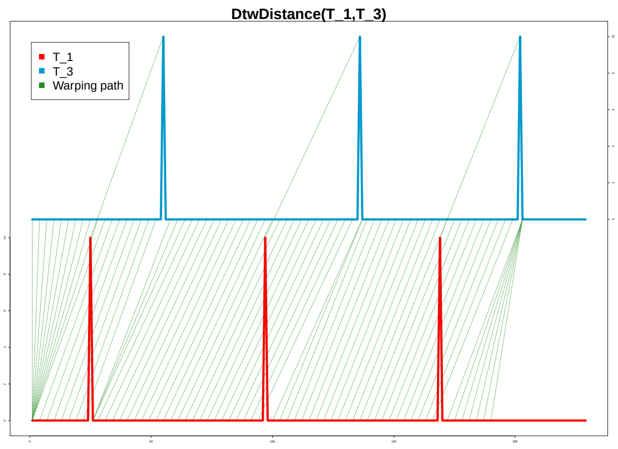


Fig. 4: DTW distance resynchronizes out of phase time series

pseudo-period techniques are used in practice by attackers in order to elude classical detection methods.

The Euclidean distance is not able to grasp the similarities of these three time series. We thus use an elastic distance: the *Dynamic Time Warping* (DTW) distance (introduced in [20]). The DTW distance between any two of these time series is null.

DTW stretches or compresses the time series locally in order to make one resemble the other. It picks the deformation of the time axis of two time series, called the *warping path*, which brings them as close as possible. Figure 4[2] depicts the best warping path for the time series T_1 and T_3.

The distance between the two time series is computed, after stretching and compressing, by summing up the local distances of individual aligned elements. We use distances from the $\mathcal{L}_p$ family for this purpose : $\mathcal{L}_p(x, y) = (\sum(x_i - y_i)^p)^{\frac{1}{p}}$. $\mathcal{L}_1$ and $\mathcal{L}_2$ correspond to the Manhattan and to the Euclidean distances respectively.

Besides, the time distortions are constrained thanks to a distortion window, denoted as w. This distortion window can be understood as the maximum allowable absolute time deviation between two matched elements. The DTW distance in figure 4 is computed without any windowing constraint ($w = 20h$). Actually, this distortion window greatly influences the DTW distance. Indeed, when the distortion window is only 1 hour, DTW cannot resynchronize the two times series as not enough distortion is allowed. However, when the distortion window is 4 hours, then DTW can resynchronize the time series, and their distance is null. Remarkably, despite of the large search space, DTW with the windowing constraint can be computed with $O(w \cdot M)$ computational complexity, using *dynamic programming* techniques, where M is the dimension of the time series.

The parameters p for the DTW local distance $\mathcal{L}_p$ and w for the DTW distortion window will be taken into account for the experiments in section V.

[2]Figure generated thanks to the `R` library `dtw` [21]

C. *Pruning*

Full decision trees, whose leaves are all pure, usually suffer from *overfitting*, i.e. they separate the labels with excessive precision and do not generalize enough. In other words, they enjoy a very low error rate on training data, but they suffer from a high error rate on testing data which are not used to train the model. In order to address this issue, decision trees can be pruned.

Yamada and Yokoi used the pessimistic pruning method for their time series decision trees [17] as it is the quickest and it does not require any test data. However, Mingers concluded in his survey [22] that this pruning method gives very bad results on certain datasets and that it should be treated with caution. Douzal-Chouakria and Amblard did not apply any pruning method to the time series decision tree models they propose in [19]. Relying on Mingers' survey, we apply the error-complexity pruning method developed by Breiman et al. [18]. The least useful subtrees according to an error-complexity measure are removed from the decision tree. They are replaced by leaves containing all the training instances of the subtrees and their label is the most represented one.

Pruning generally increases the performance of decision trees. Besides, it reduces drastically the size of the trees, and thus makes them much easier to interpret. For instance, figure 5 depicts the performance indicators comparison (the F-score and the accuracy will be defined in the following section V-A) and the comparison of the tree size both for the full tree learned and for the best related pruned tree. These results are obtained from the experiments presented in the following section.

V. Experiments

A. *Protocol*

We test the time series decision trees with several combinations of parameters (see table II) in order to find a good trade-off between detection performance, interpretation of the decision trees and prediction execution time. Besides, this experiment allows to understand the impact of each parameter.

All the experiments are run on Linux 3.2 on a dual-socket computer with 15Go RAM. Processors are Intel Xeon E5-2650

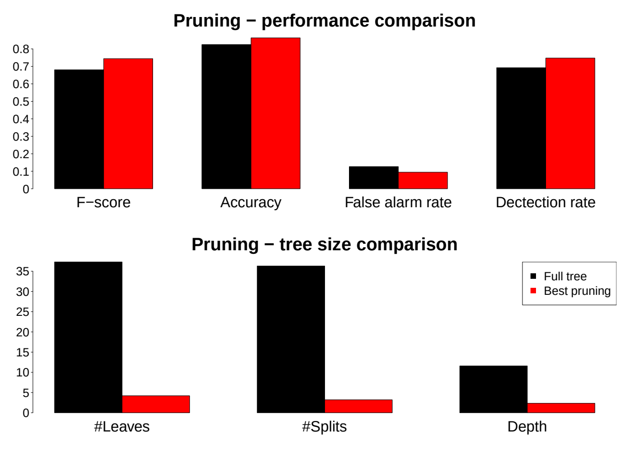


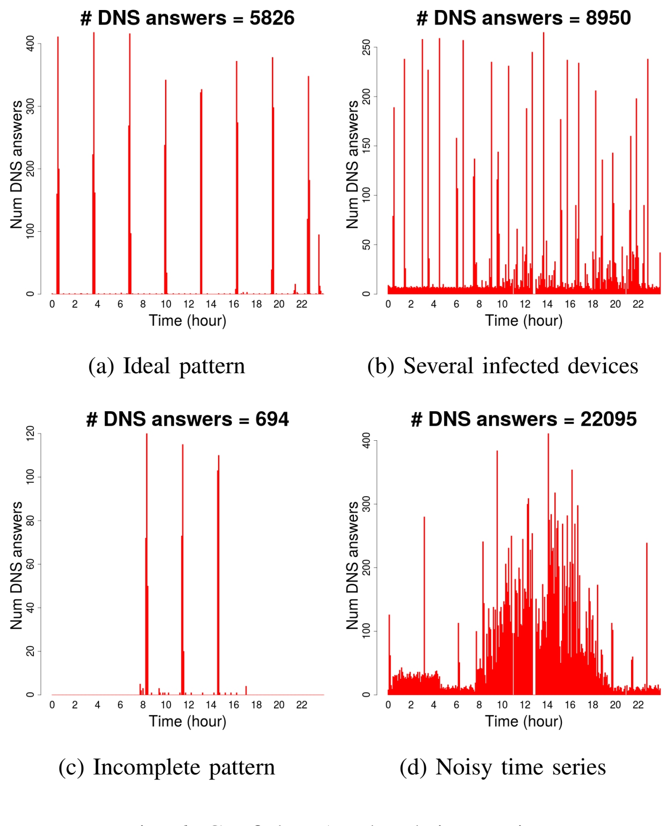
Fig. 5: Pruning improves decision trees

Parameter	Values
`SamplingInterval`	30min, 10min, 5min, 3min, 2min
`SplitCandidates`	`PC, CC, S, PC_S, CC_S`
`LocalDistance`	$\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$, $\mathcal{L}_4$, $\mathcal{L}_5$, $\mathcal{L}_6$, $\mathcal{L}_7$, $\mathcal{L}_8$, $\mathcal{L}_9$, $\mathcal{L}_{10}$
`DistortionWindow`	0h, 1h, 2h, 3h, 4h, 5h, 6h, 7h, 8h, 9h, 10h, 11h, 12h, 14h, 16h, 18h, 20h, 22h, 24h

TABLE II: Decision tree model parameters

Segment	# DNS answers
0	1 to 10
1	11 to 100
2	101 to 1000
3	1001 to 10000
4	10001 to 100000

TABLE III: Dataset segmentation



(a) Ideal pattern (b) Several infected devices

(c) Incomplete pattern (d) Noisy time series

Fig. 6: Conficker A related time series

CPUs clocked at 2.00 GHz with 8 cores each and 2 threads per core. The time series decision trees are built thanks to our `C++` implementation [23].

The experiments detailed in this section are carried out on Conficker A. We have run similar experiments on Conficker B and Kraken and the final results are directly presented at the end of the section (see table IV).

The experiments are carried out on a dataset which is composed of the 134 IPs `Infected` by the Conficker A botnet identified by the labeling module (see figure 1a) and of 366 `NonInfected` IPs selected randomly among the ones not labeled as `Infected` by the labeling module. Most of the IPs belonging to Conficker A identified by the labeling module do not exhibit so clearly the typical temporal pattern (see figure 6a). For instance, figure 6b depicts a very peculiar time series where several devices using the same IP are infected by Conficker A. More typically, figure 6c shows a time series exhibiting the DGA pattern incompletely. Indeed, when the infected device is switched off, the malware cannot contact its C&C server any more. Finally, figure 6d emphasizes that we work with real-world data and hence time series are often noisy.

Looking at the typical DNS temporal pattern of Conficker A (see figure 6a), one could think that the detection problem can be easily solved via a frequency analysis. We conducted investigations in this direction with tools like periodograms and autocorrelation. However, the frequency analysis did not seem more promising than the temporal one.

In order to better understand the dataset, we partitioned the IPs into several segments based on the total number of DNS answers received along the day with a logarithmic scale (see table III). Figure 7 shows that segments 0 and 1 contain almost only `NonInfected` IPs whereas segment 4 contain only `Infected` IPs. Thus, it will be very easy for a classifier, even if it relies on a very simple model, to be accurate on these segments. However, prediction will be much more difficult for IPs in segments 2 and 3.

In order to compare several decision tree models, we perform a *k-fold stratified cross-validation* (hereinafter referred to as "k-fold CV"), for which:

1) The training dataset is divided into k folds. Each fold has approximately the same number of instances and the same proportion of `Infected`/`NonInfected` instances as the whole training dataset.
2) The model is trained k times: each time, one fold is the testing set and the other folds form the training set.

With this method, each instance in the training dataset is used once for testing. For each combination of parameters, we run

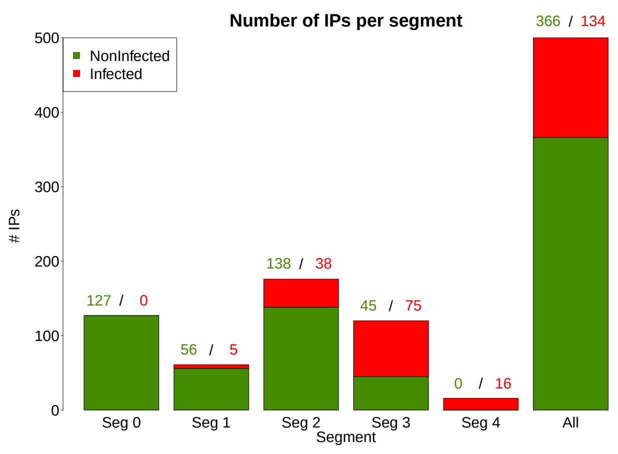


Fig. 7: Descriptive statistical analysis of the dataset

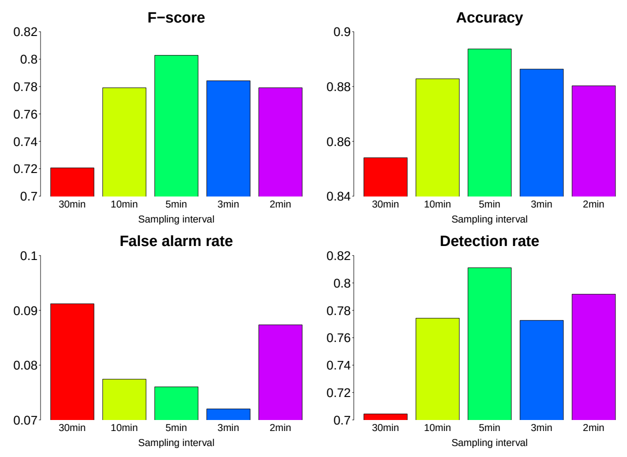


Fig. 8: Comparison of sampling steps

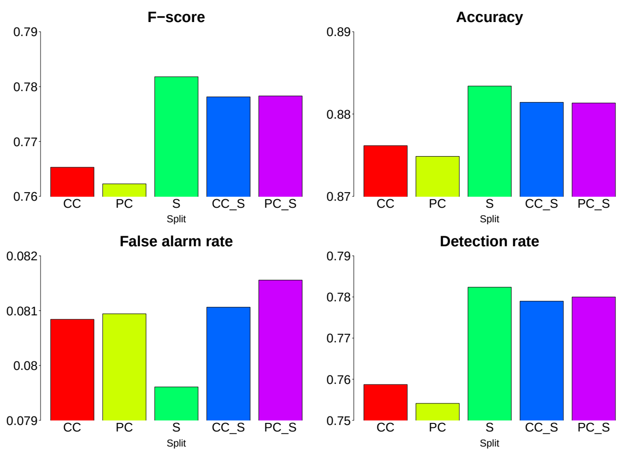


Fig. 9: Comparison of splits

a 5-fold CV. We use the following criteria for selecting the best parameter values:

1) *Detection performance*: false alarm and detection rates, accuracy[3], F-score[4] [24]
2) *Tree size*: number of splits, number of leaves, tree depth

The tree size is a very important criterion, not only for interpretation but also for the prediction execution time: the bigger a decision tree is, the more time-consuming the prediction is. Overall, we run 4,750 5-fold CVs (one for each combination of parameters, see table II) and store for each cross validation the results for the full decision tree and for the best pruned decision tree. We then average all these performance results in order to show the efficiency of the pruning method (see figure 5).

B. Results

In order to select the best values of the parameters, we consider each parameter independently, averaging the results of the best pruned decision trees over the other parameter values (we do not consider the full decision trees). We consider each parameter independently in order to understand their impact on the decision trees.

1) Best Values of the Parameters:

- **Sampling Interval** Figure 8 illustrates that when the sampling interval decreases from 30 minutes to 5 minutes, the F-score increases (the detection rate increases and the false alarm rate decreases). The decision tree is able to extract more information from the time series when it is sampled with a smaller sampling rate until it reaches the value of 5 minutes. At 3 minutes, the false alarm rate goes on decreasing, but the detection rate decreases as well, resulting in a decrease of the F-score. Thus, a 5 minute sampling interval seems the best compromise. With such an intermediate value for the sampling step, the execution time for computing the distances matrix is not too high as a 24 hour time series is represented by a 288-dimensional vector.
- **Split Candidates** First, the constrained cluster split does not seem to outperform drastically the plain cluster split (see figure 9). Indeed, the constrained cluster split has slightly a better F-score than the plain version, but the generated trees are bigger and so harder to interpret. Now, if we compare only the uncombined splits (cluster and standard splits), the standard split performs better according to all the performance indicators. Moreover, the decision trees built with the standard split are smaller. Besides, the combined splits (`PC_S` or `CC_S`) do not seem to perform better than the standard split (`S`). Finally, for the three botnets considered we come to the same conclusion as the authors in [17]: the standard split is the best choice for building accurate and interpretable decision trees (see table IV).
- **DTW Local Distance** Figure 10 depicts the detection performance indicators for various DTW local distances $\mathcal{L}_p$. It shows that when p increases, the

[3]Proportion of instances correctly classified
[4]Considers both false alarm rate and detection rate

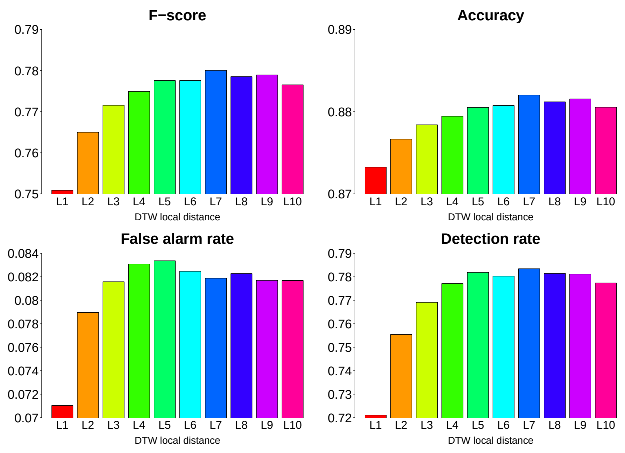


Fig. 10: Comparison of DTW local distances

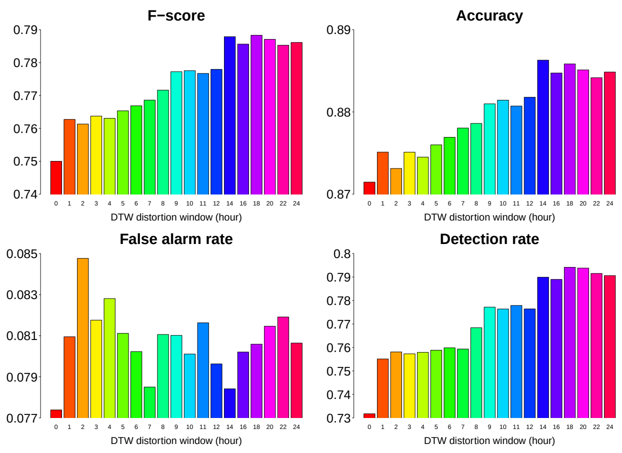


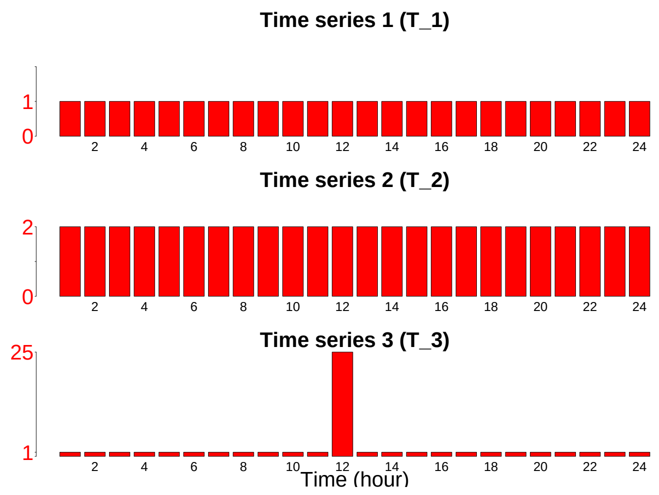
Fig. 12: Comparison of DTW distortion windows

detection rate increases but unfortunately the false alarm rate increases as well.
An intuition of this phenomenon is given in figure 11. Figure 11a introduces three toy-example time series T_1, T_2 and T_3. For designing an efficient botnet detection model, the distance between T_1 and T_2 (denoted as $\mathcal{L}_p(T_1, T_2)$) should be as small as possible, and the distance between T_1 and T_3 (denoted as $\mathcal{L}_p(T_1, T_3)$) should be large. Indeed, T_1 and T_2, which are very steady, could represent the number of DNS answers received by two `NonInfected` IPs. However, T_3 presents a peak that could be linked to DGA requests, and so we would like the time series distance to emphasize the dissimilarity between T_1 and T_3. Figure 11b represents $\mathcal{L}_p(T_1, T_2)$ and $\mathcal{L}_p(T_1, T_3)$ for several values of p. For the Manhattan distance ($p = 1$), T_1 and T_2 are as similar as T_1 and T_3: this distance is not adapted to the detection problem. When p increases $\mathcal{L}_p(T_1, T_2)$ decreases while $\mathcal{L}_p(T_1, T_3)$ remains the same. When p has a high value, the $\mathcal{L}_p$ distance emphasizes the peaks and so increases the detection rate, but unfortunately increases the false alarm rate as well.
Globally, the F-score and the accuracy increase until $p = 7$. Thus, $\mathcal{L}_7$ is considered as the best DTW local distance.

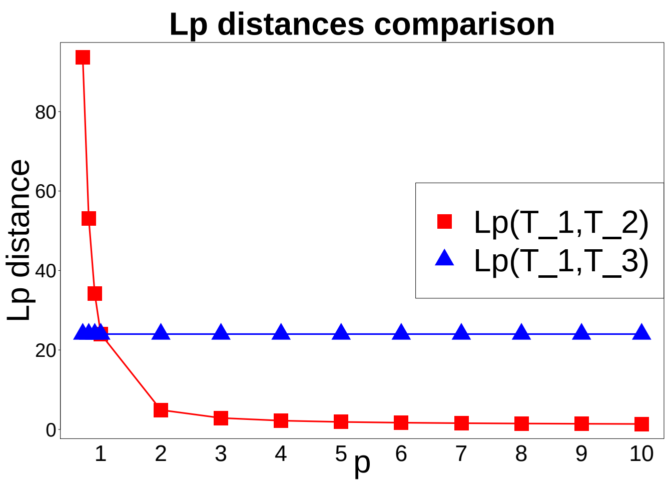
- **DTW Distortion Window** The decision trees perform better when the DTW distortion window is not null according to the F-score, the accuracy and the detection rate, as seen in figure 12. When the distortion window is null, DTW does not allow any time stretching or compressing, and thus it corresponds to a plain $\mathcal{L}_p$ distance, so that out of phase time series cannot be resynchronized. DTW is more time consuming than usual $\mathcal{L}_p$ distances but it allows to improve the performance of the model. The F-score increases until the DTW distortion window reaches the value of 14 hours (see figure 12). Besides, the generated decision trees are the smallest with this distortion window value. Thus, 14 hours is considered as the best value for the DTW distortion window. At first glance, this result is somewhat surprising considering the typical DNS temporal pattern of Conficker A (see figure 1a). As the `Infected` IPs considered try to contact their C&C server every three hours, one could expect that the best performance is reached with a distortion window around 1h30min. Indeed, a distortion window value around half of the time series period should resynchronize all the `Infected` time series. However, all the `Infected` time series do not present a perfect temporal pattern, as depicted in figure 6a. If two end users switch on their `Infected` devices at different times of the day (for instance one from 8am to 10am and the other one from 6pm to 8pm), then a bigger distortion window is required in order to resynchronize their time series.

- **Best Model Performance** The results of the independent selection of the best parameter values for the three botnets considered are presented in table IV. Thanks to the selection of the best parameter values, the decision tree performs better than Support Vector Machines (SVM) trained with aggregated values computed from the time series (mean, min, max, median, first and fourth quartiles) according to the accuracy and the F-score on all the data segments (see figure 13). Overall, the time series decision trees enjoys a higher detection rate but unfortunately it has also a higher false alarm rate. However, as the detection rate increases more than the false alarm rate, globally the F-score is higher.

2) Interpretability of Decision Trees: Figure 14a displays the decision tree built with the best values for the parameters. It is composed of a unique standard split based on an `Infected` IP time series. The selected IP received many DNS answers along the day but does not present the typical temporal pattern of Conficker A depicted in figure 6a. The execution time for building this decision tree (computing the distances matrix, building the full decision tree, generating the list of increasingly pruned trees and selecting the best pruned decision tree among the list) is 45 seconds. Once the decision tree is built, predicting the label for a new instance requires less than a tenth of a second (0.083 seconds). Even if the distortion window is somewhat high (14 hours), the prediction task is performed



(a) Examples of time series



(b) $\mathcal{L}_p$ distances comparison

Fig. 11: Interpretation of the $\mathcal{L}_p$ distances

`SamplingInterval`	5min		F-score	82.44%
`Split`	`S`		Accuracy	90.40%
`LocalDistance`	$\mathcal{L}_7$		False alarm rate	7.38%
`DistortionWindow`	14h		Detection rate	84.33%

(a) Conficker A

`SamplingInterval`	5min		F-score	73.22%
`Split`	`S`		Accuracy	86.4%
`LocalDistance`	$\mathcal{L}_4$		False alarm rate	7.37%
`DistortionWindow`	14h		Detection rate	69.40%

(b) Conficker B

`SamplingInterval`	5min		F-score	85.50%
`Split`	`S`		Accuracy	92.20%
`LocalDistance`	$\mathcal{L}_1$		False alarm rate	5.46%
`DistortionWindow`	11h		Detection rate	85.82%

(c) Kraken

TABLE IV: Best parameters and related model performance

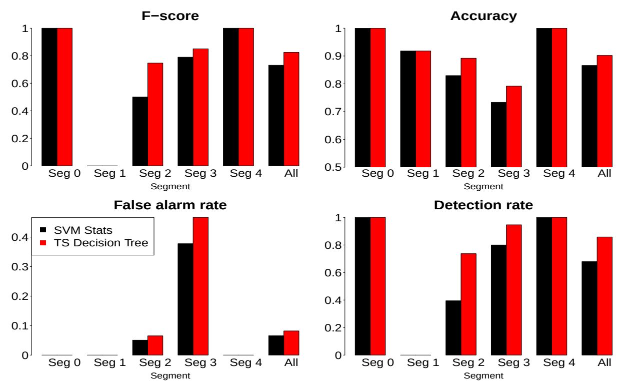


Fig. 13: Performance comparison between time series decision trees and SVM for Conficker A

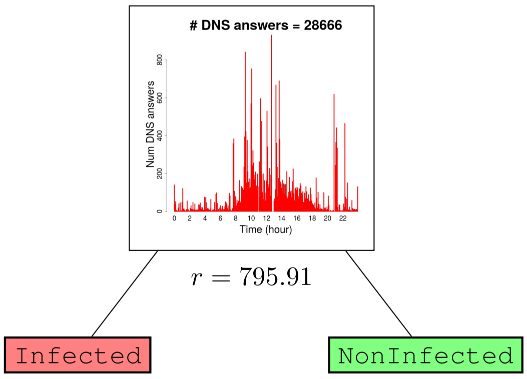
quickly as the decision tree contains only one standard split: making a prediction for a new IP requires to compute only one distance and to perform one comparison.

The decision tree pictured in figure 14b is built with a 5 minute sampling interval, `Plain Cluster` split, $\mathcal{L}_5$ as DTW local distance and a 1h30min distortion window. Even if no constraint is imposed on the labels of the time series selected, in the two splits the time series belong to different labels. The first split is composed of a `NonInfected` IP time series with very few DNS answers and of an `Infected` IP time series with many DNS answers and two peaks inside. The first cluster split relies mostly on the number of DNS answers received along the day. The second split relies partly on an `Infected` IP time series which displays the typical temporal pattern of Conficker A. This decision tree is interesting as it shows very well how the prediction is performed: first it compares mainly the number of DNS answers received along the day, and then bases the prediction on the typical temporal pattern of Conficker A.

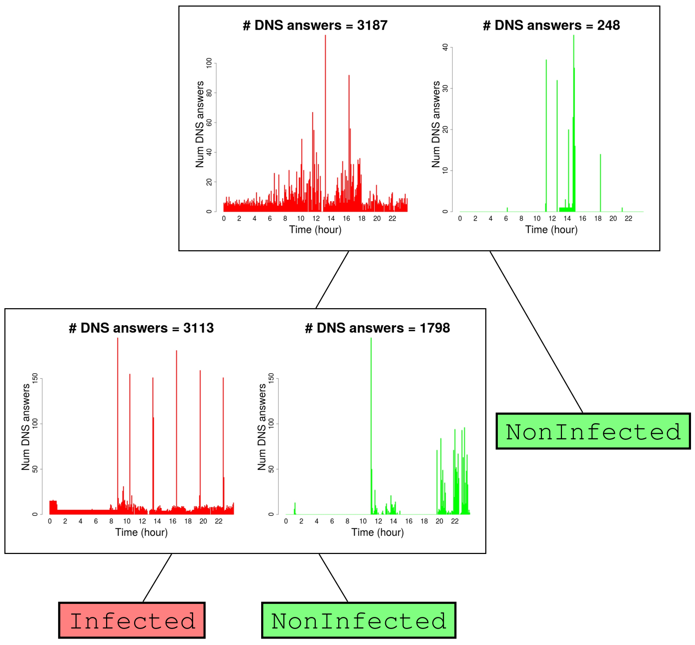
3) False Alarm Rates: For the three botnets considered during the experiments the false alarm rates lie between 5 and 7%. Such alarm rates are prohibitive for a real world bot detection system. However, our detection model relies solely on the timestamps and source IP of DNS queries while state of the art botnet detection systems use numerous features. Our goal is to show that whole times series can be used as discriminative features for botnet detection while keeping a low computational cost for detection. This way, our behavioral model can be used to enhance state of the art bot detection systems.

VI. Related Works

In the last couple of years much work has been done in order to create better botnet detection systems. Researchers agree that considering temporal correlations is important for botnet detection. Indeed, bots belonging to a given botnet share



(a) Decision tree for Conficker A (`Standard` split)



(b) Deeper decision tree for Conficker A (`Plain Cluster` split)

Fig. 14: Time series decision trees for detecting Conficker A bots

similar temporal patterns while communicating and performing malicious activities. Besides, malicious domain names used by a given botnet are queried or accessed with a similar fashion. In this section, we analyze and discuss the different ways temporal correlations have been taken into account in network-based botnet detection systems.

BotFinder [25] is a bot detection system based on NetFlow data. It builds a clustering from the average number of source and destination bytes and several temporal features: the time interval between two subsequent flows, the duration of the flows and the Fast Fourier Transformation (FFT) over a binary sampling of the C&C communication. Their study shows that the FFT is the feature with the highest normalized contribution to detection as most C&C communication patterns are periodic. However, their detection model performs very poorly on non-periodic communication patterns.

BotMiner [26] processes raw traffic in order to detect infected hosts. It clusters hosts according to their communication patterns and the malicious activities they perform. The clustering based on communication patterns relies on quantiles of variables extracted from network flow time series but quantiles lose the information on temporal correlations.

BotSniffer [27] performs a light packet inspection and analyzes HTTP and IRC flows to detect infected hosts and C&C servers. Temporal correlations are considered with an algorithm detecting whether several hosts behave similarly at a similar time window. However, although infected hosts periodically connect to their C&C server, the exact time is different as they are infected at different times. Thus, if the time window is not large enough, the algorithm will not be able to detect the temporal correlation of hosts connecting to the same C&C server. Besides, this algorithm is not able to detect generic temporal patterns, it is restricted to periodic and pseudo-periodic ones.

Disclosure [28] detects C&C servers building a Random Forest classifier from NetFlow data. This system includes atomic temporal features relying on auto-correlation and "regular access patterns" which check whether an host connects periodically to a given server. Besides, the paper mentions temporal features based on time series representing the flow volume of clients. The authors aim at discriminating between benign traffic that follows well known diurnal patterns and C&C servers traffic which is uniform. However, they provide no information about how these features are computed.

Exposure [29], [30] performs a passive DNS analysis in order to identify malicious domain names. The detection system is based on decision trees and includes four discriminative atomic time based features ("short life", "daily similarities", "repeating patters" and "access ratio") computed from DNS time series associated to each domain name. The "daily similarities" feature checks whether a domain name is queried with the same temporal pattern two consecutive days. It is computed thanks to the Euclidean distance between the time series associated to the domain name the two consecutive days. However, as mentioned in section IV-B, the Euclidean distance will fail to detect "daily similarities" if the temporal pattern is pseudo-periodic, or if it is periodic with a period which is not a divisor of 24. This feature could be made more robust thanks to the DTW distance we use to build our detection models (see sec.IV-B).

Thus, in the literature many atomic features have been extracted from network time series and some of them are very discriminative for the botnet detection task. However, they are often restricted to the detection of periodic or pseudo-periodic patterns. The behavioral model we propose takes into account whole time series and so allows to recognize generic temporal patterns.

VII. Conclusion

In this paper, we propose a behavioral model for DGA bot detection based on whole DNS time series. First, we show that DGA bots present discriminative temporal patterns. Then, we build machine learning models, time series decision trees, that are able to detect bots belonging to a DGA botnet recognizing its typical DNS temporal pattern. The time series decision trees are able to recognize any temporal pattern, they are not restricted to periodic or pseudo-periodic patterns like most of the models seen in the botnet detection literature.

Thanks to the white-box characteristics of decision trees, the detection procedure can be interpreted easily. Adapting the time series decision trees to detect other DGA based botnets is quite easy as we understand the impact of each parameter of the model. The decision trees learned enjoy a very compact representation with very few decision rules and hence allow to detect bots in very little time.

The performance of our model (see tables IV) is quite good as it relies only on temporal features. While the false positive rates are too high for a production environment, our model can be used to enhance state of the art bot detection systems. For instance, it could add a behavioral dimension to the DGA bot detection system Pleiades designed by Antonakakis et al. [6] which relies only on the syntax and structure of the domain names.

Currently, one decision tree performing a binary classification (`Infected` vs `NonInfected`) is built for each botnet. Our detection model could be easily adapted to build a unique decision tree performing a multi class classification with the labels `NonInfected`, $\texttt{Botnet}_1$, $\texttt{Botnet}_2$, $\cdots$.

Finally, several decision trees built with different parameters constitute an ensemble classifier called *Random Forest* [31]. Usually, such an ensemble classifier leads to better prediction results than the individual ones. Recently, Jason Lines and Anthony Bagnall [32] assessed the performance of several elastic distances for time series classification on many datasets. They conclude that the overall performance is not significantly different, but that the classifiers built with different distances have significant diversity in predictions on many of the datasets. For this reason, they build ensemble classifiers and show that the performance improvement of ensemble classifiers over single ones is dramatic. Thus, it would be interesting to consider time series ensemble classifiers on our dataset in order to improve the detection performance. Besides, ensemble classifiers would allow to add a detection threshold to choose the trade-off between false alarm and detection rates. In practice we are often willing to reduce the detection rate in order to get a lower false alarm rate.

ACKNOWLEDGEMENT

The authors thank Clément Jamgotchian for his work on the labeling module, Emmanuel Herbert, Stanislas Francfort and Jean-Baptiste Durand for their valuable remarks, and Marie-Françoise Colinas, Olfa Ben Haddada, Michael Liljenstam, Vincent Lemaire, Jean-Philippe Wary, Eric Gourdin and Adam Ouorou for their continuous support regarding this research work conducted at Orange Labs.

References

[1] M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis, “A multifaceted approach to understanding the botnet phenomenon,” in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 2006, pp. 41–52.

[2] D. Piscitello, “Conficker summary and review,” *ICANN, May*, vol. 7, 2010.

[3] K. J. Higgins, “Zeroaccess botnet surges,” in *InformationWeek Dark-Reading*, 2012.

[4] B. Krebs, “Operation tovar targets gameover zeus botnet, cryptolocker scourge,” in *KrebsonSecurity - In depth security news and investigation*, 2012.

[5] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling, “Measuring and detecting fast-flux service networks.” in *NDSS*, 2008.

[6] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou II, S. Abu-Nimeh, W. Lee, and D. Dagon, “From throw-away traffic to bots: Detecting the rise of dga-based malware.” in *USENIX security symposium*, 2012, pp. 491–506.

[7] 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*. Springer, 2014, pp. 192–211.

[8] 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*. ACM, 2009, pp. 635–647.

[9] “Team cymru community service,” URL: http://www.team-cymru.org/.

[10] “The spamhaus project,” URL: http://www.spamhaus.org/.

[11] “spamcop.net,” URL: http://www.spamcop.net/.

[12] M. Thomas and A. Mohaisen, “Kindred domains: detecting and clustering botnet domains using dns traffic,” in *Proceedings of the companion publication of the 23rd international conference on World wide web companion*. International World Wide Web Conferences Steering Committee, 2014, pp. 707–712.

[13] F. Leder and T. Werner, “Know your enemy: Containing conficker,” *The Honeynet Project, University of Bonn, Germany, Tech. Rep*, 2009.

[14] P. Porras, H. Saidi, and V. Yegneswaran, “An analysis of confickers logic and rendezvous points,” *Computer Science Laboratory, SRI International, Tech. Rep*, 2009.

[15] P. Amini and C. Pierce, “Kraken botnet infiltration,” 2008.

[16] C. Pierce, “Owning kraken zombies, a detailed dissection,” Technical report, http://dvlabs. tippingpoint. com/blog/2008/04/28/owningkraken-zombies, Tech. Rep., 2008.

[17] Y. Yamada, E. Suzuki, H. Yokoi, and K. Takabayashi, “Decision-tree induction from time-series data based on a standard-example split test,” in *ICML*, 2003, pp. 840–847.

[18] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[19] A. Douzal-Chouakria and C. Amblard, “Classification trees for time series,” *Pattern Recognition*, vol. 45, no. 3, pp. 1076–1091, 2012.

[20] H. Sakoe and S. Chiba, “Dynamic programming algorithm optimization for spoken word recognition,” *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 26, no. 1, pp. 43–49, 1978.

[21] T. Giorgino, “Computing and visualizing dynamic time warping alignments in R: The dtw package,” *Journal of Statistical Software*, vol. 31, no. 7, pp. 1–24, 2009. [Online]. Available: http://www.jstatsoft.org/v31/i07/

[22] J. Mingers, “An empirical comparison of pruning methods for decision tree induction,” *Machine learning*, vol. 4, no. 2, pp. 227–243, 1989.

[23] TBD, “`C++` implementation of time series decision trees.”

[24] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.

[25] F. Tegeler, X. Fu, G. Vigna, and C. Kruegel, “Botfinder: finding bots in network traffic without deep packet inspection,” in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 349–360.

[26] G. Gu, R. Perdisci, J. Zhang, W. Lee *et al.*, “Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection.” in *USENIX Security Symposium*, 2008, pp. 139–154.

[27] G. Gu, J. Zhang, and W. Lee, “Botsniffer: Detecting botnet command and control channels in network traffic,” 2008.

[28] L. Bilge, D. Balzarotti, W. Robertson, E. Kirda, and C. Kruegel, “Disclosure: Detecting botnet command and control servers through large-scale netflow analysis,” in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 129–138.

[29] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi, “Exposure: Finding malicious domains using passive dns analysis.” in *NDSS*, 2011.

[30] 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.

[31] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[32] J. Lines and A. Bagnall, “Time series classification with ensembles of elastic distance measures,” *Data Mining and Knowledge Discovery*, pp. 1–28, 2014.