Malicious traffic detection using traffic fingerprint

Introduction

In networks in general and the internet in particular, there exists a problem of malicious traffic. This malicious traffic might allow malware to spread, contaminate and communicate – as is the problem with Botnets. Such malicious traffic causes a change in the network behavior, endangers the data, the organization in which it exists and could potentially cause tangible damage to lives in case of SCADA networks.

In our article, we will present a solution for quickly identifying malware traffic by using traffic fingerprinting.

In 1978, Avraham Lempel and Jacob Ziv presented their algorithm for variable-rate compression [1]. Their dictionary-based algorithm has been used extensively for compressing many different file types, from images to text and audio.

In 1992, Feder, Merhav and Gutman [2] proposed a method for prediction of the next outcome of a sequence using the LZ78 dictionary compression algorithm. Their proposal is to create a tree based on the dictionary entries, and assign conditional probabilities to each event.

The idea to apply the method thought of by Feder, Merhav and Gutman to malware detection was first suggested by Cohen, Dolev, Gilboa and Leshem [3]. Their idea included using scalar quantization on packet interarrival times in order to predict whether unknown traffic is malicious or not.

Assumptions and model

In our model we assume the malware tries to connect a remote location. The malware attempts to impersonate legitimate network traffic and might even encrypt its packages. In our work we will look at the interarrival time between packets denoted as t_i.

Our work

Our goal is to examine if the ideas proposed in [3] are viable for malware detection. We decided to build a proof-of-concept application using Python due to the large amount of add-on packages for parsing input files and numerous mathematic packages.

Our program is split into two operating phases: a training phase and a testing phase.

Training phase

The pre-cleaned malware captures are processed to extract packet interarrival times. These interarrival times are placed in a vector which is then quantized into a user-specified amount of values (centroids) using the K-Means++ algorithm [4].

The quantized string is then used to build a LZ78 tree based on the algorithm presented in [2]. This is now our fingerprint.

This action is performed for each malware capture separately.

The fingerprints are stored in a local malware fingerprint database for easy access.

Testing Phase

A similar process is conducted for the unknown capture after it is cleaned as well (interarrival time extraction, quantization and building a LZ78 tree).

The unknown capture is then compared against the known malware database fingerprint by fingerprint, using three different algorithms – KL-Distance [5], Log Loss [6] and Hamming Loss [7] [8].

The decision score takes the KL-Distance value and compares it with an empirically decided threshold of 3.0.

Data collection

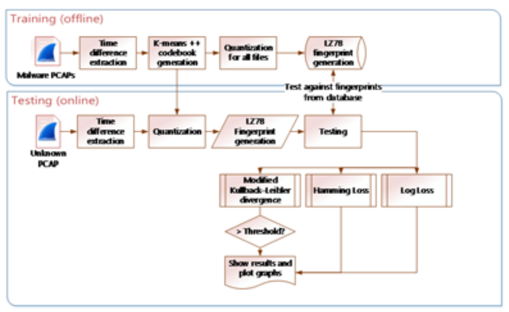

Training (offline)
Malware PCAPs
Time difference extraction
K-means ++ codebook generation
Quantization for all files
LZ78 fingerprint generation
Test against fingerprints from database
Testing (online)
Unknown PCAP
Time difference extraction
Quantization
LZ78 Fingerprint generation
Testing
Modified Kullback-Leibler divergence
Hamming Loss
Log Loss
> Threshold?
Show results and plot graphs

Figure 1- Program structure

Our goal was to test our system with real-world malware. In order to obtain captures, we cooperated with CYREN, which gave us access to their APT lab and database.

Cleaning the captures

The captures obtained were very long and contained a lot of irrelevant traffic.

In order to clean the captures, we applied the same Wireshark display rule and exported a new, clean PCAP file.

The rule was intended to keep only TCP and DNS IPv4 traffic – essentially limiting the capture to WAN traffic only.

Packet interarrival time extraction

Defined for every packet *i* such that $\{t_i - t_{i-1} : \forall i > 1\}$

K-means clustering and K-means++

In order to quantize the packet interarrival times into a string parsable by the LZ78 tree-building algorithm, we used the K-Means++ algorithm.

K-Means clustering, the basis for the K-Means++ algorithm is commonly used to partition a data-set into k groups by selecting k clusters and then refining them iteratively. Since the algorithm is at its base NP-Hard, several algorithms, such as Lloyd-Max are commonly used to converge to an optimal result more quickly.

K-Means++ [4] is an algorithm for choosing the initial seed values of the K-Means clustering algorithm first proposed in 2007 by D. Arthur and S. Vassilvitskii. It is an approximation algorithm for the NP-Hard K-Means problem, which avoids the sometimes poor clustering found by the Lloyd-Max algorithm.

In our program, the input for K-Means++ is an observation vector which is all interarrival times seen during the extraction process described above. The output for K-Means++ is a list of centroid values.

In order to perform quantization for each malware capture, we first need to set decision boundaries. The decision boundaries are set halfway between every two centroids.

Building the fingerprint

In order to build a LZ78 tree, first we must generate a dictionary containing all unique entries from the quantized string.

For example, the string '*abbacbaccbcabb*' is parsed into the following dictionary entries *a; b; ba; c; bac; cb; ca; bb*.

A tree is then built based on the dictionary entries, in such a way that each entry that is a suffix of an existing tree node is inserted as its descendant. Notable exceptions are single letters which are inserted as descendants of the root.

After the tree is built, our algorithm assigns uniform probabilities from the bottom up, summing up the probabilities from the immediate descendants. This is based on the work shown by [3] and [2].

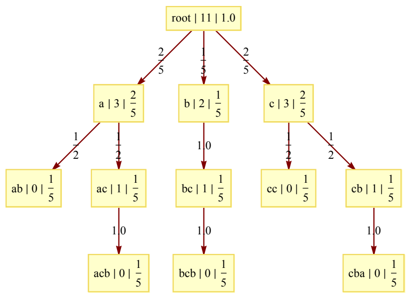


Figure 2 - Fingerprint tree

Figure 2 above is an example of a LZ78 tree generated for some malware traffic. The conditional probability is easy to calculate from this tree. For example, probability of event 'ac' given 'a' is $\frac{2}{5}\cdot\frac{1}{2}=\frac{1}{5}$

Comparison and prediction methods

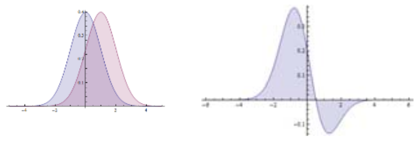
In our program we implemented three methods for comparison and prediction.

KL distance [5]

Kullback–Leibler divergence is a metric for measuring the distance between two probability measures, defined as $D(\mathrm{P}\parallel\mathrm{Q})\triangleq\sum_i \ln(\mathrm{P}_i/\mathrm{Q}_i)\cdot P_i$

In our program, each test is composed of two probability models, consisting of discrete probabilities for the fingerprint and the raw capture data. When comparing the two trees on a node-by-node basis - there could be a case where Q_i=0. To counter this case, we replaced zeroes with a very small ε.

In the following example, two Gaussian distributions $P(x)$ and $Q(x)$ with means 0 and 1 respectively.



$$D(\mathrm{P}\parallel\mathrm{Q})=\tfrac{1}{2}$$

Figure 3 - KL Distance example

Log loss and Hamming loss

The two techniques Log loss [6] and Hamming loss [7] [8] are functions that represent the cost or value of an event, or an error metric.

The error metric is used because we wish to predict the interarrival time for the next packet, based on the context of the interarrival times we have seen so far.

Since we have already assigned probabilities for each event, these will be the probabilities that will feature in the loss functions.

$$LogLoss=-\frac{1}{n}\sum_{i=1}^{n}[y_i\log(\hat{y}_i)+(1-y_i)\log(1-\hat{y}_i)]$$

$$\text{HammingLoss}(x_i,y_i)=\frac{1}{|D|}\sum_{i=1}^{|D|}\frac{xor(x_i,y_i)}{|L|}$$

Where L is the number of labels, D is the number of samples, x_i is the prediction and y_i is the ground truth

In Log loss, the use of the log function on the probability causes extreme punishment for being confident about a wrong prediction.

In Hamming loss, any mismatch between the prediction and the real value will cause a punishment of 1.

Test results

For testing our application, we first cleaned 19 different captures of five main viruses: Renovator, Hesperbor, Darkcomet, Cryptlocker and Bladabindi.

These malware captures were recorded at different parts of the day on different computers. Some even contain slightly different behaviour, due to different set-ups and time of day.

The parameters we used in our program set the number of centroids to be produced to 25 and the threshold value to 3.0.

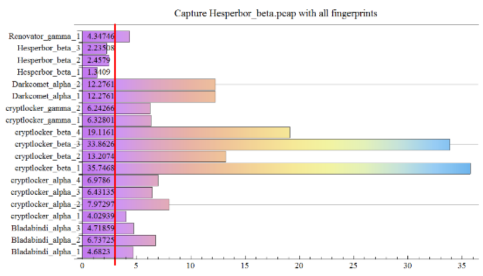


Figure 4 - Capture contaning Hesperbor behaviour

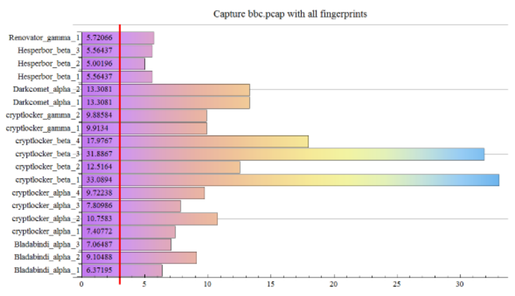


Figure 5 - Capture containing no known malware

In Figure 4 - Capture contaning Hesperbor behaviourFigure 4, a virus called Hesperbor is hiding inside a 'dirty' capture containing a lot of varied traffic (surrounding and interspersed). The capture fingerprint was compared against the entire database. The values assigned represent the KL distance between the capture fingerprint and the malware fingerprint. The greater the number, the more substantial the distance. The vertical line represents the empirically derived threshold.

If the value assigned is below the threshold then we can decide with good confidence what malware the capture contains.

In Figure 5 we took a clean capture of access to bbc.co.uk. We can see that none of our virus fingerprints were identified in this capture (all values above 3.0).

ROC graph

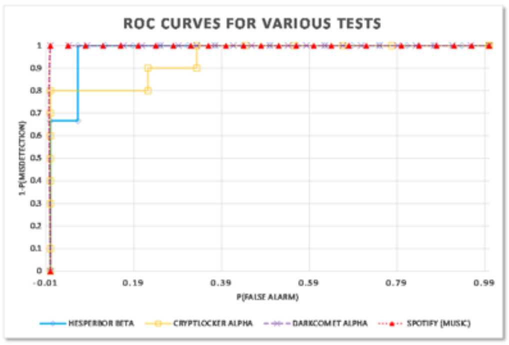


Figure 6 - ROC graph showing testing behaviour for different malware/software fingerprints

This ROC graph shows the characteristic results with a few malware fingerprints and one non-malware protocol.

Works Cited

[1] A. Lempel and J. Ziv, "Compression of individual sequences via variable-rate coding," *IEEE Transmissions on Information Theory,* pp. 530-536, September 1978.

[2] M. Feder, N. Merhav and M. Gutman, "Universal prediction of individual sequences," *IEEE Transactions on Information Theory,* vol. 38, no. 4, pp. 1258-1270, July 1992.

[3] A. Cohen, S. Dolev, N. Gilboa and G. Leshem, "Anomaly Detection, Dependence Analysis and," Beersheba, 2012.

[4] D. Arthur and S. Vassilvitskii, "k-means+: The advantages of careful seeding," *Proceedings of the eighteens annual ACM-SIAM symposium on Discrete algorithms,* pp. 1027-1035, 2007.

[5] S. Kullback and R. Leibler, "On Information and Sufficiency," *The Annals of Mathematical Statistics,* vol. 22, no. 1, pp. 79-86, 1951.

[6] C. M. Bishop, Pattern Recognition and Machine Learning, M. Jordan, Ed., Cambridge: Springer, 2006, p. 209.

[7] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining,* vol. 3, no. 3, pp. 1-13, 2007.

[8] R. W. Hamming, "Error detecting and error correcting codes," *Bell Systems technical journal,* vol. 29, no. 2, pp. 147-160, 1950.