Malicious traffic detection using traffic fingerprint

Arnon Shimoni and Shachar Barhom

Abstract—We consider the problem of detecting malicious traffic in high bandwidth links. With the ever increasing bandwidth and traffic, deep packet inspection interferes with throughput and becomes computationally demanding. We developed a learning algorithm based on the well-known universal compression algorithm Lempel-Ziv 78 [1]. We built a proof-of-concept application that emulates a real-world situation and attempts to identify malware. Our algorithm builds a traffic fingerprint for well-known malware using only the time difference between packets. The algorithm then compares the fingerprint against unknown traffic. We study the effectiveness of our method with real-world malicious traffic.

Index Terms—Anoamly Detection, Botnets, Command and control channels, Lempel-Ziv, Universal Compression, Probability assignment

1 Introduction

Cyberattacks are an attempt to damage, disrupt, or gain unauthorized access to a computer, computing systems or a network. Cyber-attacks can affect a wide range of domains and system and potentially cause tangible damage to lives in case of SCADA networks.

Malware is a piece of computer software that uses vulnerabilities in computer hardware and software in order to alter the state or function of computers and computer networks without permission (explicit or implicit). Modern malwares depends on communication networks in order to receive commands, coordinate attacks (DDoS), relay information to the attacker and infect new targets.

Detecting malicious traffic in high bandwidth links is a challenging and complicated task. One of more prevalent solutions is deep packet inspection (DPI). DPI scans the entire packet stream, and can be used to identify malware communication in the data section of the packet. The main drawbacks of this approach are the computational power needed to classify the traffic, as well as the difficulty of inspecting encrypted packets.

A different solution is feature extraction, which attempts to overcome the disadvantage of deep packet inspection by extracting a limited number of features from the packet. When performing an analysis over large amount of traffic data and malwares, deciding which features to extract requires a large amount of memory and computation power.

Our research focuses on a different approach for traffic classification, known as traffic fingerprint. This method overcomes some of the disadvantages of the methods described. Our solution examines only the time difference between packets.

The learning algorithm we represent is based on the well-known universal compression algorithm Lempel-Ziv 78 [1]. We create a traffic fingerprint using the time differences from malware communication. We represent malware communication events as discrete sequences over small finite alphabet. This sequence is then used for building a Lempel-Ziv 78 tree with a probabilistic prediction model [2]. This modified tree is used as fingerprint for each specific malware and represents the malware behaviour. A similar approach was used when attempting to identify a unique user typing on a computer [3]. This approach enables us to make fast and accurate decisions without the need for packet analysis.

A. Shimoni and S. Barhom are with the Department of Communication System Engineering, Ben-Gurion University, Beer-Sheva, 84105, Israel. E-mails: {arnons,barhoms}@post.bgu.ac.il

2 Preliminaries

2.1 Lempel Ziv 78

In 1978, Avraham Lempel and Jacob Ziv presented their algorithm for variable-rate compression [1] (LZ78). Their dictionary-based algorithm has been used extensively for compressing many different file types, from images to text and audio. The LZ78 algorithm is a universal prediction, one pass algorithm. It builds a weighted tree from sequences of a finite alphabet.

The LZ78 tree holds a dictionary of phrases parsed from the input text (training) and is constructed incrementally as follows: At the beginning, the dictionary is empty. During each step of the algorithm, the smallest prefix of consecutive symbols not yet seen is added to the dictionary. As such, each phrase is unique in the dictionary and it extends a previously seen phrase by one symbol. For example, the string *'abbacbaccbcabb'* is parsed into the following dictionary entries *a; b; ba; c; bac; cb; ca; bb* (See example in Figure 1)

Since each phrase extends a previously seen phrase, we can order them in a tree, as described in [4]. In [2] the authors proposed a method for prediction of the next outcome of a sequence using the aforementioned LZ78 tree, by assigning conditional probabilities to each event. In [3], an expanded LZ78 tree is used in order to identify the user typing on a computer keyboard. The authors suggested an expansion of the LZ78 tree seen in [4] by using input shifting and back-shift parsing, in order to rectify noisy statistics caused by small training sets.

The idea to apply the LZ78 tree prediction method to malware detection was first suggested in [5]. Their idea included using vector quantization on packet time differences in order to predict whether unknown traffic is malicious or not.

2.2 Malware Traffic

Malware (contraction of malicious software) is any piece of software intended to gather information, cause damage or infiltrate a target computer without permission. In the past, most malware was in the form of viruses and worms, and was usually distributed physically. In recent years, with the spread of computer networks in general and the internet in particular,

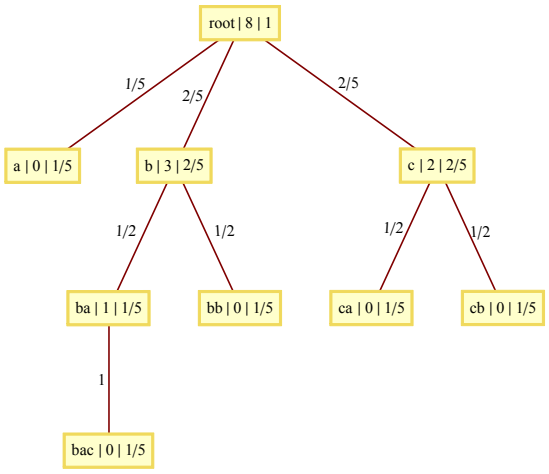

Fig. 1. Sample LZ78 tree generated from the string *'abbacbaccbcabb'*

most malware now utilizes the internet or a local network for spreading and coordinating actions.

When a virus coordinates actions, it is known as a botnet. Such actions might include distributed attacks (DDoS), personal data gathering, e-mail spam, etc. A botnet usually has a command and control (C&C) channel that it utilizes for coordinating such attacks. The C&C channel is usually obscured by impersonating a well-known protocol to some extent, such as HTTP port 80, HTTPS port 443, IRC ports, etc. The traffic generated by the malware when communicating through the C&C channel tends to remain consistent [6].

Lets examine the Cryptolocker ransomware as a test case. Cryptolocker is a ransomware trojan which infects Windows based PCs. Usually, the attack begins as an e-mail attachment, after which the Trojan is installed on the PC. When activated, the Trojan encrypts a selection of files on the computer using RSA public-key cryptography with the private key stored on a remote server. The user is then given an ultimatum to pay bitcoins in order to receive the decryption key. If no payment is made by the deadline, the Trojan threatens to erase the key from the server and keep all of the data encrypted.

The virus therefore has an interesting network traffic profile. It first arrives via e-mail. Then, once installed it begins looking for a server. First, it attempts to access a hard-coded IP 184.164.136.134. If that fails, it generates a pseudo-random URL based on the time of day. This rule is known and allows the operator of the Trojan to pre-register the pseudo-random domain names [7].

Once a suitable command and control server has been found, the malware will start to communicate through regular HTTP POST requests (See figure 2), albeit only as a wrapper for RSA encrypted data (See figure 3). This behaviour is for the most part constant and was identified in several captures on different days and different machines.

3 Identifying Traffic Based on the LZ78 Fingerprint

The core of our application is the LZ78 based fingerprint. Each fingerprint represents the behaviour of one malware capture. We must first describe a method for transformation of the network behaviour into a parsable input sequence which is

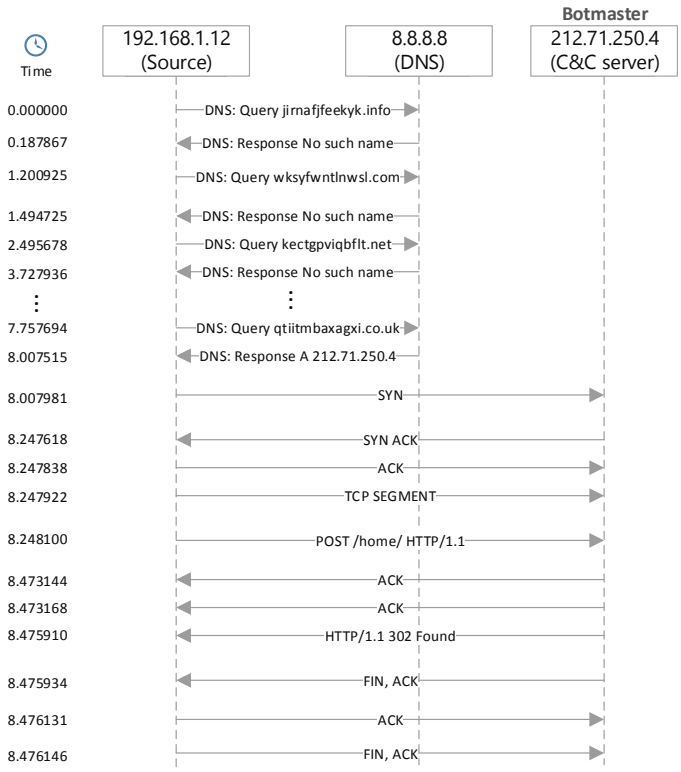

Fig. 2. Cryptolocker sample conversation list

```
POST /home/ HTTP/1.1
Cache-Control: no-cache
Connection: Close
Pragma: no-cache
Accept: */*
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT
    6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR
    3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0)
Content-Length: 192
Host: qtiitmbaxagxi.co.uk

..c/y@.
 ..'$xD2..._..dZX5..@\.....e..b.B.....+.=A<"#m...q..4Ip
      .........C.z<{.T)-..e..A..q.......n..s..........4t.r
      ..x.i.....y....E.]l+..c..........6..$..c...,.0..W...0f
      ..B.<2...w}[.....f..S2
```

Fig. 3. Cryptolocker outgoing TCP packet

quantized using a vector-quantization clustering algorithm to be fed into the LZ78 tree creation algorithm.

3.1 Representation via quantized packet time-difference

The packet time difference is the time elapsed between to packet arrival/departure events in the same flow. We will define the time difference between packets as in (1) below.

$$\Delta_i \triangleq t_{i+1} - t_i \tag{1}$$

Each stream of length k is transformed into a sequence of time differences $\Delta_1, \Delta_2, ..., \Delta_k$ Because $\Delta_i \in \mathbb{R}^+$ is in an infinite range, we wish to reduce the number of time differentials and smooth them over. Thus, we perform a vector quantization using K-Means clustering. This enables us to use fewer symbols in our learning phase which reduces variance.

3.2 K-means clustering and K-means++ quantization scheme

In order to quantize the packet time differences 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++ [8] 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 time differences 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.

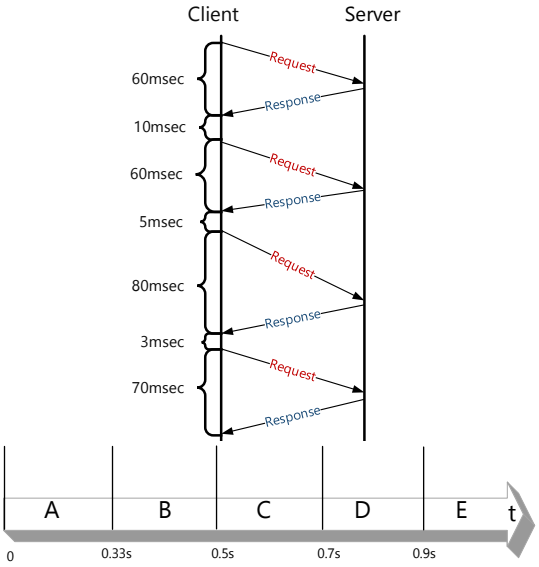

Fig. 4. Packet time difference quantization and letter assignment example

3.3 Classification Methods

3.3.1 Smoothed KL Distance

KullbackLeibler divergence is a metric for measuring the distance between two probability measures, defined as

$$D(P||Q) \triangleq \sum_i \ln(P_i/Q_i) \cdot P_i \tag{2}$$

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 $Q_i = 0$, causing an undefined value. To counter this case, we implemented a modified version we call *Smoothed KL Distance*. The algorithm runs on two trees, where T_{FP} is the fingerprint, while T_U is the unknown capture. (The trees are built as described in section 3.4.1)

Algorithm 1 Smoothed KL Distance

```
procedure SMOOTHEDKL(T_FP, T_U)
    sum ← 0
    for all Node n_i in T_FP do
        p_i ← probability of node n_i
        if Exact matching node m ∈ T_U then
            q_i ← probability of node m
        else
            q_i ← probability of closest node in T_U (lexicographically)
            q_i ← q_i · ε                    # ε is very small
        end if
        sum ← log_k(p_i/q_i) · p_i           # k — amount of centroids
    end for
    return sum
end procedure
```

3.3.2 Hamming Loss and Log Loss

The two techniques Log loss [10] and Hamming loss [11], [12] 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 time difference for the next packet, based on the context of the time differences 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.

$$LogLoss = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)] \tag{3}$$

$$HammingLoss(x_i, y_i) = \frac{1}{|D|}\sum_{i=1}^{|D|}\frac{xor(x_i, y_i)}{|L|} \tag{4}$$

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.

3.4 Program structure and algorithm

We built 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.

3.4.1 Training phase

The pre-cleaned malware captures are processed to extract packet time differences. These time differences are placed in a vector which is then quantized into a user-specified amount of values (centroids) using the K-Means++ algorithm. The quantized string is then used to build a LZ78 tree based on the algorithm presented previously. This represents our malware fingerprint[1]. The fingerprints are stored in a local malware fingerprint database for easy access.

1. This action is performed for each malware capture separately

Algorithm 2 Training Algorithm

1: **procedure** TRAINING
2: $GV_O \leftarrow []; GV_C \leftarrow []$
3: $Database \leftarrow []$
4: **for all** Capture$_i$ ←Malware captures **do**
5: APPENDTO(GV_O, $[\Delta_1, \Delta_2, ..., \Delta_k]$)
6: **end for**
7: $GV_C \leftarrow$K-MEANS++(GV_O)
8: **for all** Capture$_i$ ←Malware captures **do**
9: $V_i \leftarrow \Delta_1, \Delta_2, ..., \Delta_k$
10: Apply quantization transformation $\Delta \Rightarrow c^*$ where $c^* = \arg\min_{c_i} |\Delta - c_i|$
11: Map each quantized value into a corresponding letter from the Latin alphabet (a,b,c,)
12: $Tree_i \leftarrow$LZ78 TREE GENERATION([c_1,c_2,c_3,...] as a string)
13: Append to $Database \leftarrow Tree_i$
14: **end for**
15: **return** $Database$
16: **end procedure**

3.4.2 Testing phase

In the testing phase, an unknown capture is to be assigned a score in comparison with our fingerprint database. The lower the score, the better the match. A process identical to the training phase is performed on the unknown capture to be tested, with the exception of Algorithm 2 stage 13 — the capture is not placed in the database. The unknown capture is then compared against the known malware database fingerprint by fingerprint, using three different algorithms Smoothed KL-Distance, Log Loss and Hamming Loss. Each of these algorithms returns a numerical value for each pair of fingerprints. These values will be used to classify the capture as malware or malware free.

4 DATASETS AND EXPERIMENTAL SETUP

4.1 Gathering data

4.1.1 Malware dataset

We used Wireshark captures containing known malware captured by a third-party using sandboxed computers. The captures were identified by the third-party, and contained many varied traffic interspersed: UDP, TCP, HTTP, POP3 and ICMP to name a few.

We used captures from malware that communicates to remote machines via WAN. Thus, we were left with captures of Renovator (1 capture), Cryptolocker (10 captures), Bladabindi (3 captures), DarkComet (2 captures) and Hesperbor (3 captures).

Some of the malware we tested attempt to appear as standard HTTP traffic (as described with Cryptolocker in section 2.2), while some access other TCP ports.

4.1.2 Control dataset

The control dataset includes some background data and website access traffic. The traffic was captured using Wireshark on standard Windows 7 personal computers. Supplemental captures were obtained from readily available Wireshark capture repositories. Captures include standard HTTP traffic, SMTP, IRC, Spotify, IMAP and NNTP (7 captures).

4.2 Filtering the captures

Filtering irrelevant entries in the traffic captures required onerous non-automated work. First, all traffic inside the LAN was removed. WAN IPs were cross-referecned with malware research websites in order to discover if any IPs have already been 'incriminated'. Similar actions were performed on DNS requests to reveal malicious URLs. All traffic to non-incriminated URL and IPs was then removed, leaving behind the core behaviour of the malware.

4.3 Experimental results

Our test setup included training on the malware dataset and control dataset. We set the number of centroids to 25.

We gathered uncleaned captures of malware and other background traffic, captured on different occasions and different machines. These captures will be tested and verified against the fingerprint database created during training.

The results shown only includes the *Smoothed KL distance* method. The other methods examined did not yield sufficiently certain results.

Table 1 specifies results for comparing an unknown capture versus the fingerprint database using the *Smoothed KL Distance method*.

A lower result indicates that the distance between the unknown capture and the fingerprint is smaller, meaning a better match.

Examining the *Cryptolocker* α test, we can see that the best matches are against other captures of Cryptolocker (See Table 1 below), with the exception of Cryptolocker β which exhibited slightly different behaviour.

The *Hesperbor* malware capture shows much better results when compared to other versions of the malware, with a distance metric smaller than 1. For the other fingerprints, the distance metric is much larger (> 3.0).

We have found empirically that results below 3.0 represent a good match. In future works, this parameter can be used as a classifier rule for deciding if a capture contains malware traffic.

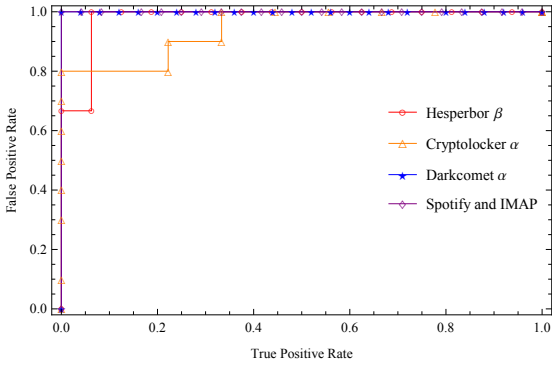

Fig. 5. Characteristic ROC for various captures

5 CONCLUSION

The conclusion goes here.

APPENDIX A

Appendix 1

ACKNOWLEDGMENTS

The authors would like to thank...

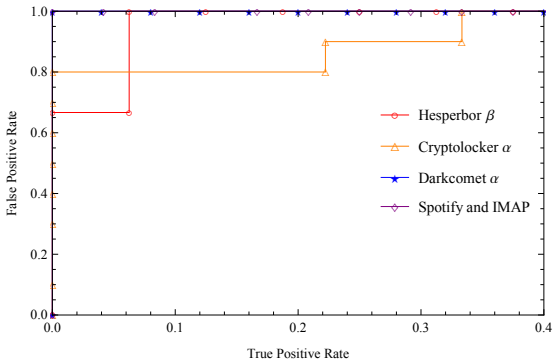

Fig. 6. Zoom in on Figure 5

REFERENCES

[1] Avraham Lempel and Jacob Ziv. Compression of individual sequences via variable-rate coding. *IEEE Transmissions on Information Theory*, page 530536, September 1978.

[2] Meir Feder and Neri Merhav and Michael Gutman. Universal prediction of individual sequences. *IEEE Transactions on Information Theory*, 38(4):12581270, July 1992.

[3] Mordechai Nisenson and Ido Yariv and Ran El-Yaniv and Ron Meir. Towards Behaviometric Security Systems: Learning to Identify a Typist. *Knowledge Discovery in Databases: PKDD 2003, Lecture Notes in Computer Science*, 2838:363374, 2003.

[4] Jr Langdon G.G. A note on the Ziv - Lempel model for compressing individual sequences (Corresp.). *IEEE Transactions on Information Theory*, 29(2):284287, 1983.

[5] Asaf Cohen and Shlomi Dolev and Niv Gilboa and Guy Leshem. Anomaly Detection, Dependence Analysis and. Technical Report, Ben Gurion University of the Negev, Beersheba,, 2012.

[6] Nart Villeneuve and James Bennett. Detecting APT Activity with Network Traffic Analysis. 2012.

[7] EMSISoft Blog. CryptoLocker - a new ransomware variant Available: *http://blog.emsisoft.com/2013/09/10/cryptolocker-a-new-ransomware-variant/*, [Accessed 16 August 2014]

[8] David Arthur and Sergei Vassilvitskii. k-means+: The advantages of careful seeding. *Proceedings of the eighteens annual ACM-SIAM symposium on Discrete algorithms*, page 10271035, 2007.

[9] Solomon Kullback and Richard Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):7986, 1951.

[10] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, Cambridge, UK, 2006.

[11] Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview *International Journal of Data Warehousing and Mining*, 3(3):1-13, 2007.

[12] Richard W Hamming. Error detecting and error correcting codes. *Bell Systems technical journal*, 29(2):147160, 1950.

PLACE PHOTO HERE

Name Biography text here.

Name Biography text here.

TABLE 1
Test results for various raw captures

Fingerprint	**Cryptlocker α (raw)**	**Hesperbor (raw)**	**DozmotD**	**Spotify and IMAP**
Telnet	9.43	8.17	11.05	9.58
Spotify	11.78	15.25	17.73	**4.26**
SMTP	4.90	10.09	8.78	10.06
NNTP	12.08	9.37	13.83	7.75
IRC	27.07	21.01	23.90	16.58
HTTP, JPEG	18.98	8.12	18.25	11.64
Bitorrent	5.9	9.11	9.96	5.52
Renovator γ 1	5.02	4.31	5.80	4.65
Hesperbor β 3	3.39	1.40	4.67	6.80
Hesperbor β 2	3.47	1.69	4.98	5.45
Hesperbor β 1	3.32	**0.44**	4.62	5.90
Darkcomet α 2	11.11	12.18	13.44	13.96
Darkcomet α 1	11.11	12.19	13.41	13.94
Cryptolocker γ 2	0.45	8.1	5.09	9.07
Cryptolocker γ 1	0.39	6.22	5.95	9.93
Cryptolocker β 4	2.23	23.73	21.50	23.16
Cryptolocker β 3	5.07	34.79	31.52	33.80
Cryptolocker β 2	1.54	13.26	12.60	13.97
Cryptolocker β 1	1.87	32.58	29.60	30.95
Cryptolocker α 4	1.15	6.07	5.74	9.79
Cryptolocker α 3	0.48	5.51	6.58	8.05
Cryptolocker α 2	**0.31**	7.11	8.37	9.85
Cryptolocker α 1	1.54	3.25	**2.91**	7.25
Bladabindi α 3	3.18	5.41	6.60	6.56
Bladabindi α 2	4.29	6.67	9.12	9.60
Bladabindi α 1	3.66	3.61	6.38	6.74

Notes:

- **Bold** text marks best match.
- All raw captures differ from training captures.
- The DozmotD malware underwent no training at all.