Expert Systems With Applications 118 (2019) 246–260

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

The arms race: Adversarial search defeats entropy used to detect malware

Héctor D. Menéndez*, Sukriti Bhattacharya, David Clark, Earl T. Barr

Gower St, London, WC1E 6BT, United Kingdom

ARTICLE INFO

Article history:
Received 16 May 2018
Revised 23 August 2018
Accepted 6 October 2018
Available online 6 October 2018

Keywords:
Malware
Information theory
Entropy
Time series
Packing
Adversarial learning

ABSTRACT

Malware creators have been getting their way for too long now. String-based similarity measures can leverage ground truth in a scalable way and can operate at a level of abstraction that is difficult to combat from the code level. At the string level, information theory and, specifically, entropy play an important role related to detecting patterns altered by concealment strategies, such as polymorphism or encryption. Controlling the entropy levels in different parts of a disk resident executable allows an analyst to detect malware or a black hat to evade the detection. This paper shows these two perspectives into two scalable entropy-based tools: EnTS and EEE. EnTS, the detection tool, shows the effectiveness of detecting entropy patterns, achieving 100% precision with 82% accuracy. It outperforms VirusTotal for accuracy on combined Kaggle and VirusShare malware. EEE, the evasion tool, shows the effectiveness of entropy as a concealment strategy, attacking binary-based state of the art detectors. It learns their detection patterns in up to 8 generations of its search process, and increments their false negative rate from range 0–9%, up to the range 90–98.7%.

1. Introduction

Arms races alternate between incremental and disruptive moves like the stockpiling of armaments and the invention of airplanes. The malware detection/evasion arms race is no exception. Its history exhibits periods of minor moves and counter-moves like tweaking malware to avoid known signature of disruptive moves like the transition to polymorphic concealment. Our core contribution is to show how to use search to restrict the adversary to only making disruptive moves. Given an evasion or detection technique, we use machine learning to search for transformations that produce variants that force the adversary to make expensive, disruptive moves. The specific detection and evasion techniques we consider use information theoretic entropy.

To conceal their malware, black hats often rewrite it. Polymorphism hides malware by encoding it and decoding it at runtime. Because it is trivially semantic preserving, it is the dominant way black hats conceal their malware — in 2016, Webroot reported that 97% of malware is polymorphic (Lonas, 2016). There are two main classes of polymorphism: those that encrypt the malware and those that compress it.

Both encryption and compression increase the entropy of their input, when forced to produce output whose size is not much larger than the input. Neither Trojans that are large relative to their hosts nor disk-resident malware that hide themselves polymorphically can violate this constraint without risking becoming too large and therefore less viable. In this case, polymorphism introduces an entropy signature (Lyda & Hamrock, 2007) that distinguishes it from many classes of benign-ware. The promise of entropy as a malware detector is that it works on executables as binary strings, without needing pre-processing, disassembly, dynamic analysis, reverse engineering, or manual analysis.

Structural Entropy (SEnt) (Sorokin, 2011) took the first step toward effectively exploiting entropy to detect malware when treating executables solely as binary strings. When a file is separated into chunks, the *entropy signature* of that file is its per-chunk entropy. SEnt computes the entropy signature of a file over small, fixed size chunks. After computing a file's entropy signature, SEnt segments this signature into sequences of chunks, treats these segments as symbols, then uses Levenshtein to compare the resulting string against the segmentation strings it extracts from other files. It has two severe limitations. First, it does not scale: it relies on the pairwise comparison ($O(n^2)$) of a suspicious program with a zoo of known malware and benign-ware. Second, its decomposition of an entropy signature into segments, whose entropy it compares, relies on six parameters, three of which are implicit and baked into SEnt

* Corresponding author.
E-mail addresses: h.menendez@ucl.ac.uk (H.D. Menéndez), s.bhattacharya@ucl.ac.uk (S. Bhattacharya), david.clark@ucl.ac.uk (D. Clark), e.barr@ucl.ac.uk (E.T. Barr).

https://doi.org/10.1016/j.eswa.2018.10.011

technique itself (Section 4.1). The authors do not elucidate the settings of these implicit parameters, nor is it obvious from first principles how they should be set and whether they can be learned. There are other three explicit parameters: the chunk size that divides the file into blocks to measure their entropy, the number of chunks or blocks used and a noise threshold.

To overcome these limitations, we present a disruptive detection move called EnTS (Entropy Time Series), a new entropy-based malware detector. EnTS scales and requires only three explicit parameters that can be learned from a corpus. EnTS constructs a metric space for entropy signatures. EnTS considers an entropy signature as a time series, then applies wavelet analysis to clean the signature and extract a simplified signature from the amplitude and longitudinal variation in a file's entropy. EnTS then treats this signature as a point. In this way, EnTS constructs a metric space, while SEnt resorts to segmentation and then to the pairwise computation of edit distance to construct its metric space (Section 2). Eliminating SEnt's segmentation step removes the implicit parameters that SEnt requires. The zoo defines EnTS' three explicit parameters: the chunk size, number of chunks, and noise threshold (Section 4.1).

We designed EnTS to defeat polymorphism. On a corpus that includes the Kaggle and the VirusShare training sets and an equal number of benign-ware from download.com , EnTS achieves 82% accuracy when maximizing 100% precision and 93.9% accuracy when maximizing accuracy. EnTS surpasses the quality of SEnt in terms of scalability: it is more than 15 times faster and linear in its time and memory consumption in contrast to SEnt's $O(n^2)$. EnTS' accuracy is between 2 to 5 points higher than SEnt's in all cases. EnTS is also good at detecting metamorphic malware, not just polymorphic, because metamorphic malware often has compressed or encrypted regions. It detect all the metamorphic variants from our test corpus. EnTS outperforms all 56 VirusTotal AV engines[1] applied to the same data, the best of which achieved only 40.6% accuracy. EnTS' time complexity is linear in the number of files being classified; it is 3000 times faster than its main competitor in accuracy, another information theoretic technique named normalized compression distance (Li & Vitányi, 2013) that we built as a baseline (Section 4.1).

Clearly, the next disruptive evasion to defeat EnTS must control the entropy of packed regions. For this task, we developed EEE (the evolutionary packer or 'El Empaquetador Evolutivo'). EEE is a polymorphic engine that controls the entropy of packed regions under a tight space budget, increasing the size of its input at most 1% (Section 5). EEE creates a packed variant with a specific entropy signal by creating chunks with a specific entropy and injecting into the packed binary. To decide position, entropy, and size of these chunks, EEE leverages evolutionary computation. EEE is an instance of adversarial machine learning; it uses search to exploit the vulnerabilities of a entropy-based detector in order to fool it (Section 5).

EEE defeats SEnt, EnTS, and other state of the art binary-based detection techniques (Section 6.2). EEE explodes the false negative (FN) rates of these techniques: Prior to EEE's application, these technique's FN rates range over [0%–9.4%]; after EEE's application, their FN rates range over [90.8–98.7%]. EEE rapidly learns to defeat these tools; It takes two generations to defeat the weakest ones and eight to defeat the strongest. To its credit, EnTS resists EEE better than the other techniques (Section 6.3).

EnTS embeds binaries into entropy metric space and uses machine learning to detect malware and to advance the state of the art. Defeating it requires EEE, a disruptive entropy-based evasion technique that uses search to control the entropy of the binary it is concealing.

The main contributions of this paper are:

- We introduce EnTS (Section 2), the new state of the art in entropy-based malware detector that operates in linear time. EnTS detects malware with high accuracy and 100% precision, achieving better detection rates than any single VirusTotal AV engine (Section 4.7), outperforming the previous state of the art and two baselines constructed from normalized compression distance and compressibility rate (Section 4.4).
- We introduce EEE (Section 5), a new polymorphic engine that leverages information theory and adversarial machine learning to evade detection. EEE defeats all known mechanisms for detecting malware in binaries based on entropy or n-gram, including EnTS: it pushes all of their false negative rates over a 90% (Section 6).
- This work presents a blueprint for how to use search and machine learning to automate incremental moves (EEE's entropy signature adaptation), thereby forcing the disadvantaged player to resort to disruptive moves (EEE forces expensive dynamic detection).

EnTS, EEE, and the corpus on which we evaluated them will be available online[2]

2. EnTS: Entropy Time Series Analysis

The dominant forms of polymorphism hide their payloads using either compression or encryption. These are string transformations that change the entropy of their input string. The promise of string-based malware detection is that it can distinguish benign-ware from polymorphic malware based on the differences in the entropy of their binaries. This is a potentially disruptive move that could obsolete the current state of the art polymorphic engines. In pursuit of this game changing malware detection, we present EnTS[3], which we designed to advance the state of the art in the scalability and accuracy of string-based malware detection.

Our goal is to define entropy-based signatures for code. Shannon entropy (Shannon, 1948) is defined over an event sequence, hence, we must convert strings to event sequences. First, we define the event space as the byte sequence within a string, and measure their entropy counting the byte frequency. This defines the entropy signature. This method can be applied to any string-based information source, like source code. EnTS instantiates this idea for binaries.

We consider each file as a stream of chunks (fixed length segments), each with an associated entropy value. The entropy of each chunk is calculated from the byte frequencies of that chunk from which a probability distribution on the bytes is calculated. As we want to compare strings of different lengths, we normalize the signature to a fixed length. This signature is noisy so we use wavelet analysis to clean it. Finally, EnTS leverages machine learning to classify the signatures.

EnTS exploits time series analysis. Time series have been widely studied in the literature (Brockwell & Davis, 2013), applied in many fields, and have often been used for prediction. Typically, time series are either analysed to estimate the next value (Chatfield, 2000) or grouped by similarity (Liao, 2005). Here, we focus on similarity in our design of Entropy Time Series or EnTS.

Like other machine learning malware detection techniques, EnTS requires labelled data (Dua & Du, 2016). It compares a sus-

[1] VirusTotal comprised 56 AV engines at the time of our experiments, 2016.

[2] EnTS is available at https://github.com/hdg7/EnTS and EEE is available at https://github.com/hdg7/EEE.

[3] EnTS is also part of our technical report on entropy and n-gram based detection (Bhattacharya, Menéndez, Barr, & Clark, 2016).

picious binary P against a zoo containing labelled malware and benign-ware. It considers P to be a malicious if it is more similar to malware than benign-ware. EnTS creates the classification space from these binaries as follows:

1. File division: Use the smaller median file length for the zoo, to set a fixed number of chunks, N.
2. Entropy signature: Use a (Procrustean) deterministic algorithm to choose evenly spread chunks from each file to produce a vector of N chunks in order. Calculate the entropy for each chunk in the vector to obtain an N-vector of entropies.
3. Wavelet de-noising: Apply a wavelet transform to obtain an N-vector of smoothed entropies with less trivial variation.

This N-vector of smoothed entropies forms the time series for each file. We can then interpret each time series as a coordinate in an N-dimensional space and train a machine learning classifier to distinguish malware and benign-ware.

2.1. File division

We compute the entropy signature of a file, F, as a Discrete Haar wavelet Transformation (Section 2.3). This requires that the entropy signature length and, as a consequence, the number of chunks, N, be a power of 2, *i.e.* $N = 2^\alpha$ for some $\alpha \in \mathbb{N}$:

$$\alpha = \left\lceil \log \frac{\min\{median(Z|_M), median(Z|_B)\}}{c} \right\rceil,$$

where $Z|_B$ is the zoo's benign-ware, $Z|_M$ is the zoo's malware and c is the *chunk size*. The ceiling operator produces an integer between the two median lengths for the two sets of binaries. Then, for each program P, we divide its binary representation into chunks of size c.

The chunk size is a critical parameter for EnTS. Chunks are file segments but we also considered sliding windows as an alternative. This was quickly rejected because it adds redundant information into the entropy signature.

Given that the atomic constituents of chunks are bytes, it is easy to see that a chunk size of 256 bytes is optimal with respect to the amplitude of entropy variation. There are $256 = 2^8$ possible different bytes. According to Cover and Thomas (2012) the entropy is maximum if and only if the distribution is uniform, *i.e.* the entropy of a chunk will be maximal when every possible byte has equal probability. This corresponds to a uniform distribution, where every element has the same appearance probability. To measure the probability of appearance of these 256 elements, we need, at least, 256 samples composing a chunk. Hence, the minimum chunk size that allows the maximal possible variation in entropy (from 0 to 8 bits) is 256. On the other hand we want as many chunks as possible in each file so we also want the length of chunks to be as small as possible. These chunks provide more information about the entropy signature, showing granular variations within it.

Example. Consider a zoo of just two binary files, P and Q, and a chunk size of c. These programs, considered as binary strings, are divided in chunks. Suppose that $length(P) = 20c$ and $length(Q) = 6c$. Each chunk is related to a wavelet coefficient, therefore, the number of coefficients would be 20 for P and 6 for Q. However, the Haar wavelet requires 2^α coefficients. Next section shows how to adapt the width.

2.2. The entropy signature

Once we have the chunk division for a file, we need to reduce or increment the number of chunks to N, in order to fit the mother wavelet, which is explained next in Section 2.3. The selection process of the chunks is equidistant. The first and last chunks have special status because file head and tail are usually relevant parts in malware analysis. To choose the rest, we calculate an increment value $inc = (|C| - 1)/(N - 1)$ to get the next chunk index using the floor of the accumulation of this factor as the next chosen index. For each chosen chunk, we calculate its Shannon entropy on the basis of the byte frequencies of the chunk:

$$H(C_j) = -\sum_{b \in C_j} p(b) \log_2 p(b), \tag{1}$$

where $p(b)$ is the probability of byte b within the jth chunk, C_j, of program P, calculated from its frequency count within the chunk.

The concept of chunk generates a local entropy computation. Therefore repeated chunks will not reduce the entropy. For example, imagine a string (00011011) over the alphabet $\Sigma = \{0, 1\}^2$. This string has maximum local entropy, *i.e.*, chunk entropy. However, the global entropy $H((00011011)^k)$ tends to 0, as $k \to \infty$ with Σ fixed. In this case, local or chunk entropy is maximised, while global entropy is minimized. This problem did not occur in our dataset and can be easily checked.

Example. Following the example of Section 2.1, we need to adapt the width of P and Q to the Haar wavelet coefficients, which are 2^α. Suppose that we choose $\alpha = 3$, then we need $N = 2^\alpha = 8$ coefficients. For P we need to contract the number of chunks from 20 to 8 and for Q we need to increase the number of chunks from 6 to 8. In order to choose these chunks, we generate a subset of the current chunks using a jump factor for each file. The chunk index is initially set to 0, and it is incremented in every step by $inc_1 = 19/7 = 2.71$ for P and $inc_2 = 5/7 = 0.71$ for Q. The indices are selected using the floor of the accumulated jump value, so the chosen indices will be:

$$I_P = (0, 2, 5, 8, 10, 13, 16, 19) \quad I_Q = (0, 0, 1, 2, 2, 3, 4, 5)$$

After, we only need to calculate the entropy of each chunk, defining an N-vector of entropy values for each file which is considered as an entropy time series.

2.3. Wavelet denoising

This last step smooths the entropy signal using wavelets. In our case, the mother wavelet is defined by:

$$W(N, b) = \frac{1}{|N|^{1/2}} \sum_{j=1}^{|C|} H(C_j) \cdot \Psi_{HAAR}\left(\frac{t_j - b}{N}\right). \tag{2}$$

where N corresponds with the dimensions of the final N-vector space, b is a shifting parameter, $H(C_j)$ are the entropy values, $|C|$ is the total number of chunks, t_j is the current chunk j in the sequence and $\Psi_{HAAR}(t)$ is the Haar wavelet defined by:

$$\Psi_{HAAR}(t) = \begin{cases} 1, & 0 \le t < 1/2 \\ -1, & 1/2 \le t < 1 \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

The Haar wavelet is chosen because it approximates a step function from the original function (Addison, 2002). Other wavelets, such as Daubechies or Biorthogonal wavelets were considered, but their performance was worse and their results were similar to the Haar wavelet. As EnTS focuses on the variation patterns, a noise free step function provides all the information it needs about the most relevant entropy variations.

Then, we calculate the discrete Haar wavelet transformation. Each iteration in the process is divided into two parts: calculating the scale coefficients and calculating the detail coefficients. The scale coefficients contain the most relevant information about the

signal while the detail coefficients contain information about the small variations. In each iteration, the coefficients used are the scale coefficients for the previous iteration, *e.g.* in iteration number 2 only the scale coefficients of iteration 1 are used to calculate the scale and detail coefficients of iteration 2, and the other wavelet coefficients are not modified. According to the Haar wavelet equations, a scale coefficient is calculated by:

$$s_i^1 = \frac{1}{\sqrt{2}}(x_i + x_{i+1}), \quad s_i^\alpha = \frac{1}{\sqrt{2}}(s_i^{\alpha-1} + s_{i+1}^{\alpha-1}), \ \alpha > 1,$$

and a detail coefficient is calculated by equations:

$$d_i^1 = \frac{1}{\sqrt{2}}(x_i - x_{i+1}), \quad d_i^\alpha = \frac{1}{\sqrt{2}}(s_i^{\alpha-1} - s_{i+1}^{\alpha-1}), \ \alpha > 1,$$

The scale coefficients are positioned at the beginning of the wavelet and the detail coefficients after the scale coefficients. For example, with $\alpha = 3$, the iterations generate the coefficients as follows:

$$\begin{array}{c}(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7)\\ \downarrow\uparrow\\ (s_0^1, s_1^1, s_2^1, s_3^1, d_0^1, d_1^1, d_2^1, d_3^1)\\ \downarrow\uparrow\\ (s_0^2, s_1^2, d_0^2, d_1^2, d_0^1, d_1^1, d_2^1, d_3^1)\\ \downarrow\uparrow\\ (s_0^3, d_0^3, d_0^2, d_1^2, d_0^1, d_1^1, d_2^1, d_3^1)\end{array}$$

In the final iteration the wavelet, W, has been constructed. We can use it to reduce the noise from the entropy time series, using a *threshold,* τ, on the wavelet coefficients in this final iteration. Those values that are below the threshold are set to 0. This process improves the performance of the classification task by eliminating minor variations in the original signature.

Lastly, we apply the inverse wavelet transformation to reconstruct the entropy signature without the noise.

$$x_i = \frac{1}{\sqrt{2}}(s_k^1 + d_k^1), \quad s_k^\alpha = \frac{1}{\sqrt{2}}(s_k^{\alpha+1} + d_k^{\alpha+1}), \ \alpha > 0$$

$$x_{i+1} = \frac{1}{\sqrt{2}}(s_k^1 - d_k^1), \quad s_{k+1}^\alpha = \frac{1}{\sqrt{2}}(s_k^{\alpha+1} - d_k^{\alpha+1}), \ \alpha > 0$$

The resulting coefficients vary between 0 and 8 because of the choice of chunk size and will be used as coordinates of the entropy time series in the classification space. This space allows the creation of scalable models based on machine learning classifiers, and significantly improves the speed of the classification process. The model scales linearly (as we discuss in Section 4.1 and show in Section 4.6) because it does not require a pairwise comparison between every element on the training data, as other state of the art algorithms such as Structural Entropy do (Sorokin, 2011). The classifier will infer a way of discriminating the malware and benign-ware files within the zoo, focused on targeting 100% precision, that is one of our main goals.

Example. Following the examples of Sections 2.1 and 2.2, We need to remove noise and simplify each time series by obtaining the reconstruction coefficients. Now, for purposes of illustration, we focus on P. We apply the discrete Haar wavelet transformation. Assume that the entropy values for P are (4,5,4,1,1,2,1,2) then the wavelet transformation process will give us:

$$W_P|_{\alpha=1} = (6.4, 3.5, 2.1, 2.1 \mid -0.7, 2.1, -0.7, -0.7)$$
$$W_P|_{\alpha=2} = (7, 3 \mid 2, 0, -0.7, 2.1, -0.7, -0.7)$$
$$W_P = (7 \mid 2.8, 2, 0, -0.7, 2.1, -0.7, -0.7)$$

We apply the threshold, in this example it is 0.75, to W_P and we get

$$W_P = (7, 2.8, 2, 0, 0, 2.1, 0, 0).$$

Then, we apply the reconstruction process to W_P and we get the reconstructed signal as (4.5,4.5,4,1,1.5,1.5,1.5,1.5). These values are the coordinates of P's signature in the space.

3. Experimental data

We want to study EnTS' performance on encryption-only, compression-only and both encrypting and compressing polymorphic engines, so we use three malware datasets: Kaggle (`Kag`) malware competition dataset[4], packed (`Pck`) malware from VirusShare[5], and `Mix`, a dataset that we construct from `Kag` and `Pck`. EnTS requires labelled benign-ware (Benign) to operate (Section 2) so, for each of these cases, we collect 3 corresponding benign-ware datasets `KagB`, `PackB`, and `MixB`.

`Kag` contains two subsets: train and test. Kaggle's test subset is not labelled, so we train and test on the train subset. It is composed of 10,869 Malware files. The dataset contains 9 malware families whose features are summarised in Table 1. The families are useful for understanding how EnTS works. There are two files per malware: a byte representation (hexdump) and an asm file with IDA Pro-information from the disassembly process. We used `xxd`[6] to convert the hexdumps to binary executables. According to Kaggle description, these binaries are not packed. This dataset was published February 2015, and it has become a benchmark on malware analysis, used in more than 50 research papers (Ronen, Radu, Feuerstein, Yom-Tov, & Ahmadi, 2018).

`Pck` contains Win32 malware whose packing system was known, so we focused on malware uploaded to VirusShare in January 2016. This database is composed of approximately 131,000 Malware files, covering different types of malware. We filter them using Linux file command. Only those files identify as PE software passed this filter. By combining Yara[7] with packer rules extracted from the YaraRules project[8]. We extracted 10,000 malware with known packers. Around 70 specific packing systems were detected by Yara, however, several of them came from the same family, so we focused on the most frequently occurring families (Table 2).

`Mix` synthesises `Kag` and `Pck` by sampling: one joining different types of malware and benign-ware and the second for distinguishing packed and non-packed software. The former is formed by $\frac{1}{3}$ from Polymorphic data (`Kag`), $\frac{1}{3}$ from Metamorphic (`Kag`) and $\frac{1}{3}$ from `Pck`. In industry, white hats often must analyse different kinds of malware at the same time. This dataset aims to emulate this scenario. The second mixed dataset, `Mix 2`, is composed by $\frac{1}{2}$ packed and $\frac{1}{2}$ non-packed files. This dataset aims to evaluate our abilities discriminating packed and non-packed.

For Benign, we collected 2000 packed benign (`PckB`) and 2000 non-packed benign files (`KagB`). For `Mix` the benign-ware mixed as $\frac{2}{3}$ non-packed and $\frac{1}{3}$ packed (to keep packed and non-packed proportions), in the first case and $\frac{1}{2}$ packed, $\frac{1}{2}$ packed for the second case. The resulting datasets have 2000 malware and 2000 benign instances. The benign files were collected from download.com. All the benign-ware was submitted to VirusTotal[9] to ensure that no anti virus detect it as malware. All the files are also PE executables and the packed files are discriminated using Yara.

EnTS is a classifier (Section 2). The class imbalance problem is the bane of classifiers (Domingos, 2012). In our corpus, malware outnumbers benign-ware so we uniformly sampled 10,000 files

[4] https://www.kaggle.com/c/malware-classification.
[5] http://virusshare.com.
[6] http://linux.about.com/library/cmd/blcmdl1_xxd.htm.
[7] http://yara.readthedocs.org.
[8] http://yararules.com/.
[9] https://www.virustotal.com/.

Table 1
Information about Kaggle classes:number of instances, instances used in EnTS' experiments, concealment strategies and types of malware.

Class	Instances	K. 1	K. 2	K. 3	K. 4	K. 5	Conc.	Type
Ramnit	1541	300	255	295	281	287	Poly	Worm
Lollipop	2478	476	427	454	479	443	Poly	Adware
Kelihos_3	2942	550	594	522	540	510	Poly	Botnet
Vundo	475	76	99	83	80	102	Meta	Trojan
Simda	42	7	7	9	8	6	Poly	Botnet
Tracur	751	126	142	150	126	143	Poly	Trojan
Kelihos_1	398	75	65	68	75	77	Encr	Botnet
Obf.ACY	1228	218	232	228	219	230	Meta	Trojan
Gatak	1013	172	179	191	192	202	Poly	Trojan
Total	10,868							

Table 2
Information about VirusShare packers: number of instances in EnTS samples, and instances in SLaMM experiments.

Packer	Total	P. 1	P. 2	P. 3	P. 4	P. 5
Armadillo	542	101	98	111	105	127
ASPack	186	32	37	42	41	34
ASProtect	54	10	4	11	17	12
Borland	2123	417	435	425	438	413
NET	2351	476	486	483	454	452
PECompact	445	88	80	89	89	99
UPX	3175	641	665	610	635	624
Rest	1124	235	195	229	226	239
Total	10000					

from the Kag and Pck dataset without replacement and randomly divided it into five partitions with equal size (Tables 1 and 2).

4. Evaluating EnTS

We built EnTS to be accurate and scalable, here we demonstrate that it achieves both ends, its accuracy is 93.9% improving over SEnt by 1.5 points, and the other state of the art up to 8.9 points, and it scales linearly.

The evaluation consists of four steps. The first is data selection. In every experiment, we have chosen the same number of malware and benign-ware instances. The training data consists of two thirds of the instances and the remaining third comprises the test data. The test data is always fresh data for either approach and is randomly selected by uniform sampling at the beginning of the process. The second step consists of the feature space generation for classification in EnTS. Once this is prepared, we train a classifier as appropriate. After, we evaluate malware detection on the test set, recording the accuracy and the false positive rate.

4.1. Algorithms and Parameters for EnTS study

To compare against EnTS, we implemented three other information theoretic features of binary strings from the literature as baselines. The first is the compression rate (CR) which calculates the ratio of the compressed length to the uncompressed length for a given file compressor and is related to the Kolmogorov complexity of the specific file (Li & Vitányi, 2013). We chose LZMA2 as the compressor and its maximum compression parameters and the maximum windows size, *i.e.*, 4GB, using the package 7zip .

The second is the Normalised Compression Distance (NCD), which approximates the Normalised Information Distance (Li & Vitányi, 2013), a universal, generic, information theoretic metric. Formally, NCD is

$$NCD(P,Q) = \frac{C(PQ) - min\{C(P), C(Q)\}}{max\{C(P), C(Q)\}},$$

where P, Q are strings, PQ is their concatenation, and $C(\cdot)$ is the compressed size function for a specified compressor. NCD also uses LZMA2 as the compressor.

Finally, we compare against Structural Entropy (SEnt). Sorokin introduced this technique in 2011 (Sorokin, 2011) and Baysa et al. applied it to metamorphic malware in 2013 (Baysa, Low, & Stamp, 2013). It divides a file into chunks, calculates the entropy of each chunk, then groups the chunks into arbitrarily sized segments (the information for each segment is its average entropy and its size). It generates a similarity matrix, performing a pairwise comparison on the files based on Levenshtein distance. This approach is $O(n^2)$, where n is the number of files. Further, the variable number and variable size of segments in a file means this approach may determine a file with more segments to be totally different from another file with fewer segments even though the overall entropy pattern in the two files is similar. EnTS escapes this problem: it extracts a fixed length entropy time series from a file as a token stream and operates directly on this time series and therefore all of the file's information at once. The implicit parameters chosen for this comparison are the same as those used in both Baysa and Sorokin's work: $\tau = 0.3$, $c_\epsilon = 0.6$ and $c_\alpha = 1.4$.

EnTs has three parameters: the chunk size, signature size, and the wavelet threshold. In this experiment we set the chunk size to 256 (Section 2.1). The signature size (*i.e.* the number of dimensions) is 512 (2^9) because the smaller zoo (packed files) has an average size of 116 KB ($\alpha = \lceil \log(116 \cdot 2^{10}/2^8) \rceil = 9$, Section 2.2). The wavelet threshold is 0.5 (Section 2.3).

EnTS uses two classic classification algorithms: Random Forest[10] and Inference Trees[11]. To optimize finding the boundary between malware and benign-ware for each measure, we used multiple-learning to combine these two algorithms. Multi-learning divides the learning process, specialising it to different regions of the space. Multi-learning penalises false positives during construction (Hothorn, Lausen, Benner, & Radespiel-Tröger, 2004).

4.2. EnTS' accuracy

EnTS was designed to detect polymorphic malware. *"How accurate it is at detecting all types of malware?"* Following related work (Section 7), we consider a detector to be accurate when its accuracy is at least 90%. EnTS uses a classifier (Section 2). To determine how much of its performance is due to its machine learning classifier and how much to its similarity metric, we compared EnTS with other information theory similiarity measures, using the same parameters and classifier and ask: *"How does EnTS' accuracy compare to that of other information theory similarity measures?"*

NCD and SEnt generate a similarity matrix while EnTS and CR describe point coordinates for the signatures and the compression

[10] https://cran.r-project.org/web/packages/party/index.html.
[11] https://cran.r-project.org/web/packages/party/index.html.

Table 3
Accuracy Results for all datasets and techniques. The best results are remarked in bold. The second best results are remarked in italic. The ▲and ▼symbols indicate whether a technique is statistically better or worse to EnTS respectively, according to the Wilcoxon test.

Data	NCD	CR	SEnt	EnTS
Kag 1	▼93.9 ± 0.3	▼91.7 ± 0.1	▼*94.5 ± 0.2*	**98.1 ± 0.1**
Kag 2	▼*94.1 ± 0.2*	▼90.5 ± 0.3	▼*94.1 ± 0.2*	**98.2 ±0.2**
Kag 3	▼*94.0 ± 0.3*	▼90.5 ± 0.0	▼93.7 ± 0.3	**97.8 ±0.2**
Kag 4	▼94.0 ± 0.3	▼91.6 ± 0.0	▼*94.4 ± 0.2*	**97.5 ±0.1**
Kag 5	▼*95.4 ± 0.3*	▼90.6 ± 0.1	▼93.9 ± 0.2	**98.0 ±0.1**
Pck 1	▲**95.4 ±0.2**	▼82.2 ± 0.2	▼92.0 ± 0.2	*94.1 ± 0.2*
Pck 2	▲**95.1 ±0.2**	▼83.0 ± 0.1	▼93.1 ± 0.2	*94.3 ± 0.2*
Pck 3	▲**95.1 ±0.2**	▼81.2 ± 0.2	▼91.1 ± 0.2	*94.0 ± 0.2*
Pck 4	**95.1 ±0.2**	▼83.3 ± 0.2	▼93.0 ± 0.2	**95.1 ±0.2**
Pck 5	▲**95.7 ±0.2**	▼81.1 ± 0.2	▼92.3 ± 0.2	*94.1 ± 0.2*
Mix 1	▼91.7 ± 0.3	▼85.0 ± 0.1	▼*92.4 ± 0.2*	**93.9 ±0.2**

rate, respectively. Applying machine learning to EnTS and CR is straightforward because we have the points and we only need to discriminate them. For NCD and SEnt we consider each row of the training similarity matrix as coordinates, due to the number of files is fixed. This provides the points that the machine learning algorithm uses. For testing, we will consider the coordinates as the similarities among the test files and the training files. The machine learning algorithms chosen are non-deterministic approaches (they choose a random seed during the initialization process), then, we need to generate different models to measure their median performance (Hothorn et al., 2004). Hence, each experiment has been carried out 100 times, and the median and standard deviation have been provided to compare the results. Furthermore, in order to compare different algorithms, we have applied the Wilcoxon test to evaluate whether there is statistical significance among the results or not. We consider that there is statistical significance when the p value is less than 0.05 using EnTS as benchmark. In order to reduce the redundancy of correlated variables in the space, we have eliminated those dimensions whose Pearson correlation was higher than 0.8 with respect to other dimension. This reduces the space to the 5% of the original dimension.

Table 3 shows the direct comparison between the four techniques discriminating malware and benign-ware, according to the accuracy. It divides the results by technique and provides the accuracy of applying each algorithm to the specific datasets described in Section 3. For Kag, EnTS and NCD generally obtain the best results (EnTS is over 97% of accuracy in all cases and NCD over 93%). SEnt is always worse than EnTS and CR is the worst approach. For Pck, all techniques reduce their accuracy but NCD, which increments its discrimination abilities. EnTS and NCD keep competitive results compare with the rest of the techniques (over 94% and 95% in all cases). Mix 1 shows that EnTS and SEnt are better discriminating malware and benign-ware than the other techniques (93.9% and 91.7% of accuracy, respectively) when the dataset mixes packed and non-packed binaries. NCD obtains worse results than in the previous cases and CR is the worst technique. This analysis shows that EnTS and SEnt are the best techniques classifying malware, when no previous information about the malware has been obtained.

Findings. We originally asked whether EnTS is accurate. Targeting accuracy, it obtains 98.0% in Kag, 94.1% in Pck and 93.9% in Mix. We also compared EnTS to the other techniques. EnTS is more accurate than CR and SEnt, and similar to NCD. These results show that NCD and EnTS are competitive classifiers in all cases, although EnTS scales 3000 times better (Section 4.6). CR does not detect any malware that the other techniques do not also detect; EnTS easily defeats SEnt.

Table 4
False positives and true positives rates for all techniques and datasets. The ROC curve that has been chosen is the median of all the ROC curves generated during the experimental process. Italic characters highlight the best results for different cut-off values. Bold characters highlight the best results with 0 false positives and the highest false positive tolerance.

	0	0.002	0.01	0.05	0.1	0.15
Kag NCD	**0.67**	*0.76*	0.86	0.96	0.98	0.99
Kag CR	0.00	0.00	0.41	0.70	0.91	0.96
Kag SEnt	0.44	0.60	0.72	0.92	0.98	0.99
Kag EnTS	0.38	0.70	*0.93*	*1.0*	*1.0*	**1.0**
Pck NCD	0.19	*0.26*	*0.55*	*0.96*	*0.98*	**0.99**
Pck CR	0.00	0.00	0.27	0.55	0.75	0.79
Pck SEnt	**0.20**	0.25	0.39	0.88	0.95	0.96
Pck EnTS	0.18	*0.26*	0.53	0.94	*0.98*	**0.99**
Mix 1 NCD	0.61	*0.67*	*0.77*	0.89	0.94	0.96
Mix 1 CR	0.00	0.21	0.26	0.58	0.81	0.83
Mix 1 SEnt	0.29	0.34	0.45	0.82	0.95	0.97
Mix 1 EnTS	**0.64**	*0.67*	0.76	*0.92*	*0.97*	**0.98**

4.3. EnTS' precision

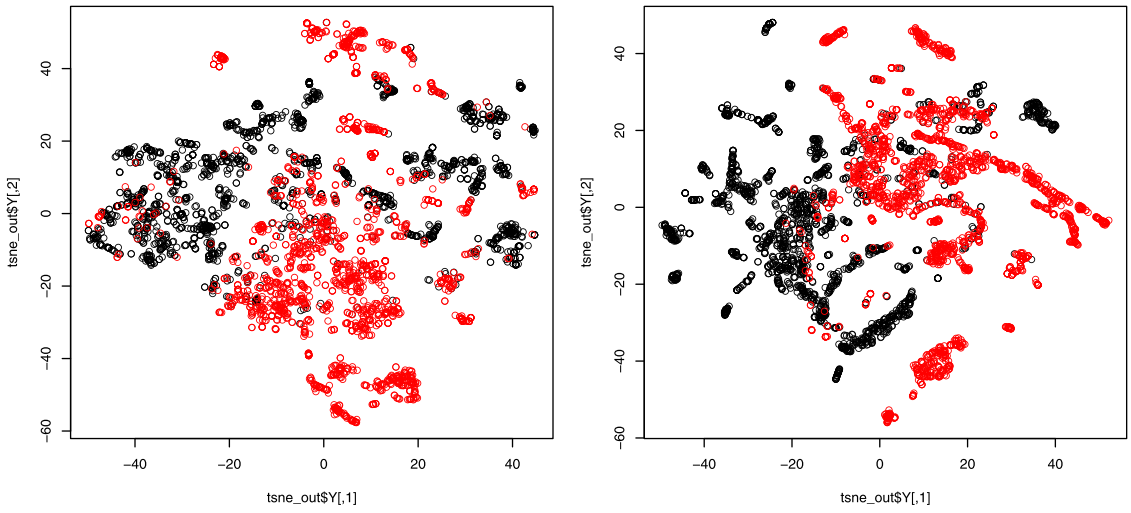
One of the main aims of a malware detector is to reduce false positives, and, as a consequence, improve precision (Section 7). We ask *"Does EnTS accurately and precisely detect malware?"*. We aim to achieve a precision of 100% (*i.e.* there are no false positives). Due to the classification nature of EnTS we use the ROC curve to decide a cut-off during its validation process. We compared EnTS with the other information theory similarity measures, using the same parameters and classifier and ask: *"How does EnTS' accuracy and precision compare to that of other information theory similarity measures, like NCD?"*

Improving precision is equivalent to reducing false positives. The classifier penalizes false positives during the learning process, as mentioned above, to ensure that the model effectively detects malicious programs. The cut-off or threshold used in the ROC curve also provides a confidence value to the random forest voting system that helps to reduce false positives. Using 10 cross-fold validation in the training set, we set the cut-off to the most conservative value, *i.e.* the one that ensures 0 false positives in all validation sets. It is this last model that we apply to the test data.

Table 4 shows the median results for the ROC curves for all the experiments. In this table, we can see how the threshold variation modifies the true/false positives rates for each dataset. For Kag, EnTS detects 38% of malware with 100% precision (*i.e.* 0 false positive), NCD detects 67% and SEnt detects 44%. For Pck, EnTS detection rate is reduced to 18%, NCD to 19% and SEnt to 20%. For Mix, NCD improves its results significantly (61%), as well as EnTS (64%). This table shows that EnTS only outperforms all techniques with 100% precision, when the data is mixed, in the other scenarios there are not significant improvements. These results also discard CR as a classifier targeting the 100% precision. After setting the threshold to the 100% precision, the median accuracy achieved by EnTS for Kag is 69.0%, for Pck is 59.0% and for Mix is 82.0%.

We aim to understand why EnTS obtains better results in some specific cases and why it is not performing as well as the other techniques with respect to the precision.

First, the results suggest that EnTS can easily separate non-packed malware and benign-ware. In order to provide an intuition about how this separation is performed we have generated a t-SNE (Maaten & Hinton, 2008) projection of Mix. This projection aims to show, in a low-level dimensional space (normally 2 or 3 dimensions) the separation among the data instances in their original (usually high-level dimensional) space, according to a specific metric, in our case, the Euclidean distance. During the mapping process, which generates the projection, t-SNE aims to keep coher-



(a) Mix EnTS (b) Mix NCD

Fig. 1. t-SNE projections of NCD and EnTS spaces using Mix data. While the borders are fuzzy, both techniques achieve good global separation; NCD's clustering is more compact, but comes at great computational cost.

ence among those instances that are close in the high dimensional space, setting them close in the low dimensional space. Compared with other projection techniques like Isomap, Sammon mapping or a Locally Linear-Embedding, t-SNE performs better showing high dimensional data discrimination based on their manifolds structure (Maaten & Hinton, 2008). It is effective with space of similar dimensions to us, for instance, Maaten and Hinton showed its performance projecting spaces of 784, 10,304 and 1024 dimensions to 2 and 3. Our space contains 512 dimensions (Section 4.1), hence our dimensionality is inside the projection bounds.

Fig. 1shows the results of this projection for EnTS and NCD. During the discrimination process, the application of Random Forest helps to discriminate those sections which are fuzzier, due to the multi-learner approach, that sets different trees in these sections to ensure a clearer discrimination. To analyse the false positive rate, we focus on those black instances (benign) invading red clusters (malware). If a black instance is in the middle of a red cluster, it will be considered as malware with high probability. Therefore, even when the cut-off is more conservative, it will still be misclassified. In Fig. 1 we can see this phenomena in both: EnTS and NCD. In this case of NCD this normally happens closer to the boundary. The cut-off sets this whole boundary section as benign, to reduce false positives. EnTS has wider red regions free of black instances. However, the boundary of these regions are problematic, due to they are covered by several black instances. This forces the cut-off to consider the boundary as benign-ware.

Findings. We asked whether EnTS is accurate and precise. It *is* precise: it obtains 100% precision. However, it falls short of the 90% accuracy bar. It obtains 64% accuracy on `Mix`, 38% on `Kag` and 18% on `Pck`. Again, EnTS is more accurate than CR and SEnt, and similar to NCD.

4.4. EnTS' accuracy by family

We want to go deeper in the specific concealment strategies used by the families and packing systems and how they affect the performance of each technique. This leads us to ask: *"How do the different families and packers affect EnTS and the other baselines?"*

Table 5breaks down the results from Table 3 in families (`Kag`), packing systems (`Pck`) and strategies (`Mix`). This only increments the granularity of the binary classification in order to detect how different families or packers affect it. For `Pck`, NCD achieves the best performance in almost all cases, followed by EnTS, but NCD's time and memory performance is significantly lower (Section 4.6). All techniques are good discriminating Armadillo system, as well as NET. For `Kag` families, we can see that NCD and EnTS outperforms the rest of techniques in all cases. This also shows the effectiveness of EnTS when it is applied to metamorphic malware. Due to metamorphic malware has not intuitive entropy variations we focus on the two specific families: Vundo and Obfuscator.ACY. Vundo was previously studied by Li et al., who provided a description about the metamorphic engine (Li, Loh, & Tan, 2011). This description mentions that the data section is encrypted or compressed, therefore this produces entropy variations that can be detected by EnTS. This fact is also detected in the entropy signature, where there are long sections with higher entropy than others. For Obfuscator.ACY the previous pattern is also frequent in the entropy signature, but in smaller sections, probably related to encrypted or compressed strings. These variation patterns make the metamorphic data totally unique for EnTS, and it is the reason it can easily detect them. For `Mix`, the best results are for polymorphic and metamorphic data, applying NCD and EnTS. For `Pck`, NCD is the best, followed, in this case, by SEnt which is close to EnTS.

Findings. We wondered how different concealment strategies affect EnTS and the other baselines. EnTS and NCD are strong against polymorphism and, surprisingly, metamorphism. They can handle specific families and packers, forcing malware writers to create new ones.

4.5. Packed and non-packed

The current state of the art is focused on distinguishing between packed and non-packed software, this leads us to ask: *"How does EnTS' detect packing compared with the other information theory similarity measures?"*

EnTS and NCD are accurate detecting malware in different packed and non-packed zoos, this section aims to analyse their ability to discriminate between packed and non-packed software. `Mix 2` dataset was designed to fulfil this goal: the dataset contains 50% packed and 50% non-packed binaries, mixing malware and benign-ware in the same proportions.

The accuracy values for distinguishing packed and non-packed are: 88.1 $\pm$ 0.7 for NCD, 69.4 $\pm$ 0.3 for CR, 82.1 $\pm$ 0.4 for SEnt and 88.9 $\pm$ 0.3 EnTS. According to Wilcoxon test, EnTS and NCD

Table 5
Breakdown of Table 3 results by malware families in Kag, packing systems in Pck and concealment strategy in Mix. Bold characters highlight the best results and italic character the second. Underlined characters highlight results higher to a 99%.

Dataset	Class	NCD	CR	SEnt	EnTS
Kag	Ramnit	▼*97.9 ± 0.7*	▼95.6 ± 0.3	▼96.1 ± 1.1	**98.8 ± 0.4**
	Lollipop	▼*97.9 ± 0.5*	▼94.6 ± 0.2	▼96.0 ± 0.8	**99.2 ± 0.3**
	Kelihos3	*98.2 ± 0.7*	▼93.8 ± 0.3	▼95.5 ± 0.8	**98.9** ± 0.2
	Vundo	*97.4 ± 1.2*	▼94.9 ± 0.1	▼92.8 ± 1.8	**98.0** ± 0.4
	Simda	▼*95.0 ± 2.5*	▼97.1 ± 1.6	▼97.1 ± 0.0	**100.0 ± 0.0**
	Tracur	▲**99.5 ± 1.4**	▼94.2 ± 0.2	▼96.6 ± 1.4	*98.6* ± 0.4
	Kelihos1	▼*98.4 ± 1.2*	▼97.0 ± 0.5	▼94.6 ± 2.7	**100.0 ± 0.9**
	Obf.ACY	▲**98.7 ± 0.8**	▼93.8 ± 0.2	▼96.1 ± 1.0	*97.6* ± 0.6
	Gatak	▼*98.2 ± 0.8*	▼93.2 ± 0.3	▼96.2 ± 1.2	**99.3 ± 0.5**
Pck	Armadillo	▲**99.5 ± 0.3**	▼94.7 ± 0.6	▼95.7 ± 0.5	*97.3 ± 0.3*
	ASPack	▲**91.6 ± 3.7**	▼59.3 ± 2.5	▼80.2 ± 3.2	*88.8 ± 3.6*
	ASProtect	▼*80.0 ± 7.4*	▼36.0 ± 0.0	▼58.0 ± 4.9	**88.0 ± 7.5**
	Borland	▲**97.1 ± 0.5**	▼78.3 ± 0.6	▼84.3 ± 1.2	*88.8 ± 1.0*
	NET	**99.6 ± 0.1**	▼95.2 ± 0.2	▼98.7 ± 0.4	*99.2 ± 0.3*
	PEComp	*91.4 ± 4.4*	▼70.1 ± 2.5	▲**94.3 ± 1.1**	*91.4 ± 1.3*
	UPX	▲**97.0 ± 0.3**	▼76.8 ± 0.6	▼92.5 ± 0.5	*94.6 ± 0.7*
	Rest	▲**91.7 ± 0.9**	▼77.9 ± 1.1	▼77.8 ± 1.7	*82.4 ± 1.6*
Mix	Meta	**100.0 ± 0.0**	▼95.9 ± 0.3	▼96.8 ± 0.5	**100.0 ± 0.0**
	Poly	**99.6 ± 0.2**	▼95.5 ± 0.3	▼98.2 ± 0.4	**99.6 ± 0.1**
	Packed	▲**86.9** ± 0.8	▼65.8 ± 0.2	▲*85.1* ± 0.7	81.5 ± 1.0

Table 6
False positives and true positives rates for all techniques and Mix 2 dataset. The ROC curve that has been chosen is the median of all the ROC curves generated during the experimental process. Bold characters highlight the best results.

	0	0.002	0.01	0.05	0.1	0.15
Mix 2 NCD	0.27	0.29	0.44	0.75	**0.88**	**0.95**
Mix 2 CR	0.11	0.11	0.16	0.31	0.42	0.51
Mix 2 SEnt	0.16	0.24	0.38	0.57	0.71	0.79
Mix 2 EnTS	**0.51**	**0.53**	**0.64**	**0.81**	**0.88**	0.92

Table 7
Average time results for the different methods and all the databases. Time is approximated in minutes (m) and days. Bold characters highlight the best results.

	NCD	CR	SEnt	EnTS
Kag Space Gen	> 5 days	30 m	40 m	2 m
Kag Classification	25 m	0.4 m	3 m	0.5 m
Kag Total	> 5 days	30.4 m	43 m	**2.5m**
Pck Space Gen	> 5days	30 m	40 m	2 m
Pck Classifications	25 m	0.4 m	3 m	0.5 m
Pck Total	> 5 days	30.4 m	43 m	**2.5m**
Mix Space Gen	> 5 days	30 m	40 m	2 m
Mix Classification	25 m	0.4 m	3 m	0.5 m
Mix Total	> 5 days	30.4 m	43 m	**2.5m**

results are not significantly different. Table 6 shows the detection percentage considering packed as the detection target. It illustrates that EnTS overcomes the rest of the techniques specially for 0 false positives.

Findings. After comparing EnTS packing detection abilities with the other techniques, we discovered that EnTS is more accurate than CR and SEnt and similar to NCD. EnTS also performs better than the other techniques when the target precision is 100%.

4.6. Scalability

We explore the scalability by asking: *"Does EnTS scale better than NCD, CR and SEnt?"* Table 7 shows the average time consumption of the techniques for training and testing. The table is divided in three datasets (Kag, Pck and Mix), and three specific values: the space generation or training (where the algorithms generate the similarity matrices, entropy signatures or the compressibility values), the classification process and the total average time. EnTS outperforms every single technique. We can also see that NCD is the most impractical technique, taking 2 days in the best case and 5 in the worse. This shows that NCD is not optimal for malware detection. It is a consequence of the file compression and the pairwise comparison to generate the similarity matrix. The compression process also affects to CR which needs more time to calculate the ratios. The pairwise comparison affects to NCD and SEnt. EnTS uses no pairwise comparison, and this improves the time consumption. Besides, the entropy signature generation and the wavelet decomposition are linear processes, they do not generate a bottleneck during the analysis.

The memory consumption of each metric grows depending on the space size. For NCD and SEnt, this space is related to the similarity matrix, which grows as $O(P^2)$ while EnTS grows linearly $O(P)$ according to the number of programs, P, due to the number of coefficients (or coordinates) used in the space is fixed. CR also grows linearly according to the number of files.

The time consumption ranking for the techniques and for datasets containing 2000 malware and 2000 benign-ware starts with NCD consuming more than five days. It follows with SEnt consuming 43 min, CR consuming 40.3 min and finally EnTS consuming only 2.5 min. The equivalent memory consumption ranking starts with NCD and SEnt consuming a big square similarity matrix ($O(P^2)$). It follows with EnTS and CR as $O(P)$ techniques.

Findings. EnTS *does* scale better than NCD, SEnt and CR. It is linear scalable, 10 times faster than the second fastest technique.

4.7. EnTS vs AV Engines

This last part of the study was focused on comparing EnTS with commecial tools. We ask: *"Can EnTS improve the detection results of the AV engines?"*

We have compared EnTS with 56 Anti-Virus Engines. For this comparison, we have sent all the test set from the Kag, Pck and Mix to Virus Total. In the case of Pck, all the data was already classified as Malware using this system, but Kag is fresher and there are a few anti-virus that can detect it. Table 8 shows the comparison between the best engines related to accuracy. We can see that EnTS and NCD obtain the best accuracy results.

Table 8
Comparison between the top ten Anti-Virus Engines, EnTS, CR, SEnt and NCD according to detection.

Technique	Kag	Pck	Mix
EnTS	**98.9%**	*93.0%*	*93.7%*
CR	94.4%	81.7%	85.7%
SEnt	95.6%	90.5%	93.4%
NCD	*98.2%*	**96.7%**	**95.5%**
Avast	29.1%	83.5%	40.6%
AVG	0.3%	86.1%	27.5%
Avira	6.4%	23.5%	11.8%
ESET	0.0%	87.1%	28.2%
GData	0.0%	87.5%	27.8%
Ikarus	0.6%	86.6%	27.2%
McAffe	0.0%	89.4%	28.8%
Qihoo	4.3%	61.2%	24.0%
VBA32	0.1%	73.4%	23.2%
VIPRE	0.0%	88.1%	27.5%

Findings. We find that EnTS *outperforms* all the 56 AV Engines in term of accuracy up to 69 points for the best anti-virus using the Kaggle data.

4.8. Discussion

The results suggest that EnTS quality depends on the sparsity of the data in the space. When the data is more sparse, *i.e.*, when the entropy signatures are different among them, it is more difficult for the classifier to find a good discrimination, however, in the opposite situation, it is clear that the variants generate small clusters in the space, where the families or the packers are set together. EnTS space is based on the signatures, it does not depend on the data, therefore the classifier can be easily transported to detect other malware or retrain with new malware, keeping no information of the original training data. Zero-day malware, which is totally different to all the previous data and more likely to benign-ware, might be a countermeasure for EnTS, but if black hats aims to repack or re-conceal variants from current malware they will find limitations set by EnTS (we will discuss this fact in the following section).

NCD's quality roots in the compressor: when NCD assigns a high similarity, the strings have patterns that can be identified by the compressor after the concatenation. However, when NCD sets two strings as different, it is not confident, because if one of the objects is already compressed, the distance will be maximum. NCD space is based on similarities, therefore, the object selection will affect the space construction. In this case, the fact that we work with specific families and packers improve the abilities during the detection process, because they are more likely to be similar among them. On the other hand, the scalability of NCD is extremely problematic if we want to use this technique as an online detector. However, EnTS is almost 3000 times faster than NCD for the zoos we had studied.

Next section will be focused on finding a potential countermeasure generating variants for EnTS.

5. EEE: the evolutionary packer

EnTS advances the state of the art in entropy-based malware detection, achieving an unprecedented combination of speed and accuracy. Is it a disruptive move? To answer this question, we immediately take the next step in the malware detection arms race and present EEE (the evolutionary packer or "El Empaquetador Evolutivo"), an EnTS countermeasure. EEE manipulates the entropy signature of the binaries to create malware variants. It injects controlled entropy regions (CERs) into the binary file and learns how many CERs to create and where to put them. In so doing, EEE defeats EnTS and all other frequency-based malware detectors.

Fig. 2shows EEE workflow. It uses a malware binary and a detector as starting points (Algorithm 1). The malware contains the malicious semantics, which is not modified. EEE changes the malware shape injecting CERs (Section 5.1). This produces variants whose new features aim to produce a misclassification in the detector. Due to the manipulation process might not be enough, we include a learning process, based on genetic algorithms (Section 5.2). EEE learns to create and place the CERs based on the variant's classification score, which feeds the fitness function of the learning process. Every variant generated is executable and it runs as the original malware after the unpacking process performed in runtime (Section 5.3).

Algorithm 1 EEE evolutionary process.

Input: P is the input program.
Det Malware detector.
Output: P^* is the best program variant.

```
1: P̂ = Compress(P)
2: Create an initial population of Chromosomes: Pop
3: for i = 0 to Total_Generations do
4:   for all C ∈ Pop do
5:     for all d_i ∈ C.D do
6:       for j = 0 to d_i.Num do
7:         // cers is a priority queue sorted by position
8:         cers.enqueue(d.N * |P̂|, d.Size, d.Density)
9:     cers.removeOutOfRange()
10:    cers.NormalizeSize(|P| − |P̂|)
11:    P' := inject(P̂, cers, C.Del)
12:    C.fitness = Det.DetectionProb(P')
13:    P* = arg max(Det.DetectionProb(P'),Det.DetectionProb(P*))
14:  Pop = reproduction(Pop) + selection(Pop)
15:  Pop.crossover()
16:  Pop.mutate()
17: return P*.
```

The adversarial machine learning process of EEE is embedded into the fitness function. Every time that EEE generates a new variant with the aim of reducing the classification abilities of the machine learning based malware detector, it is playing adversary to the machine learning algorithm. From an adversarial perspective, EEE has access to the classifier and can get the classification probability, but it does not know which specific features needs to be modified, that information is learnt during the search process depending on the response from the malware detector.

5.1. Controlled entropy regions.

EEE introduces controlled entropy regions (CERs) anywhere in the binary file. A CER is a set of random bytes constructed so that the entropy of the byte distribution is under our control. EEE sets a delimiter, Del, at the starting and at the ending points of each CER for bounding them. This delimiter identifies the CERs into the binary. Three parameters control the generation of controlled entropy regions: Size, Density and a Gaussian probability distribution sampled to select their placement position (described by its mean μ and standard deviation σ). The Size parameter is given as the percentage of the available size a region can use. The Density parameter is a number between 0 and 1 used to calculate the percentage of bytes used in the CER from the 256 possible bytes. The probability distribution is a Normal distribution over the interval [0,1]. EEE samples this distribution and multiplies the value obtained by the size of the compressed file to define an insertion point.

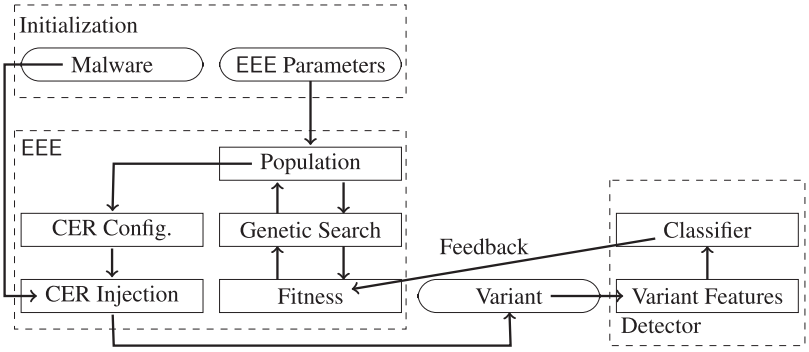


Fig. 2. The architecture of EEE, the Evolutionary Packer, showing the initialization of the packer and the GA at top, the interactions among the components of EEE and the interaction with the malware detector at bottom.

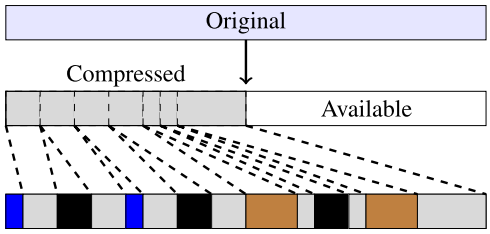


Fig. 3. Modifications on the original binary performed by EEE. First, the original file is compressed. After, the controlled entropy regions are set. Depending on their position they lie between different pieces of the compressed program.

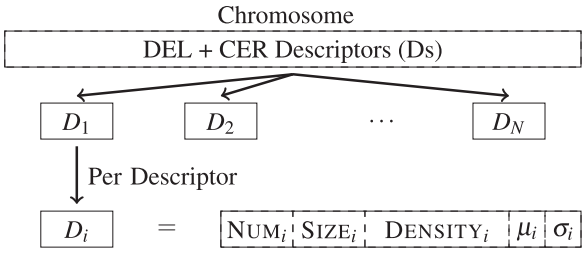


Fig. 4. EEE Chromosome scheme. A chomosome is a vector where the first delimiters; then follows the types of CER, where the information for each descriptor is the number of CERs for that descriptor, the SIZE, the DENSITY, and the parameters for the position distribution.

To construct a CER, we sample bytes from a uniform distribution until the number sampled over 255 reaches or exceeds the DENSITY. We concatenate these bytes in sampling order into an initial string, then construct a new string by repeating that initial string until the desired length is reached, based on Shannon's entropy definition (Eq. (1)) so this method achieves the maximum entropy possible for the probability distribution on byte frequencies in the CER. For example, imagine three strings, 111111, 121212 and 123123. The entropy of the first is 0, as the probability of 1 is maximum. The entropy of the second is 1, as the probability for each value is 0.5. The entropy for the third is 1.58 because the probability for each value is $\frac{1}{3}$, and so on. The entropy increases as we introduce values until our string contains all 256 bytes, when the entropy is maximum, *i.e.*, 8.

A fixed number of CER descriptors limits the CER search space. Descriptors define a set of CERs sharing similar properties. Descriptors are formed by a SIZE, a DENSITY, and a pair of Normal distribution parameters. Formally, a CER descriptor $d = (\text{NUM}, \text{SIZE}, \text{DENSITY}, \mu, \sigma)$, where NUM represents the number of CERs instantiated on this descriptor. The number of descriptors and their characteristics are parameters for the genetic search algorithm (Section 5.2).

For giving room to the CERs without significantly incrementing the file size, we start compressing the binary. After, we instantiate the CER descriptors creating NUM CERs per descriptor. Each CER has a memory position ranged from 0 to the size of the compressed area. There is a low probability to generate out-of-range CERs, due to the Gaussian distribution. Those CERs are removed. The size of the CER depends on the available area, *i.e.*, the difference between the original binary size and the compressed size. We aim to not create a variant bigger than the original malware, after the CER injection (within a 1% of tolerance). This allows EEE to work with compressed binaries. The CERs sizes are normalized according to the available size. After setting their final size, EEE injects the CERs (Fig. 3) between two delimiters (DEL).

5.2. Genetic CER creation and placement

The genetic algorithm looks for the best combination of CER descriptors and delimiters (Algorithm 1). They form the chromosome, where the CER parameters become the encoding or search space. The number of descriptors is fixed to limit the size of this space, improving the performance.

EEE encodes the CER descriptors parameters into a real valued vector, which serves as a chromosome during EEE's search. Fig. 4 shows the components of a chromosome: delimiter (DEL), and, for each region descriptor: DENSITY, SIZE, μ, σ and number of regions (NUM).

The adversarial search process runs as given in Algorithm 1. It starts compressing the original program $P \rightarrow \hat{P}$ and creating a population of chromosomes (*Pop*) that represents different parametrizations for the CERS (lines 1 and 2). These chromosomes are created by sampling from a uniform distribution. Then for each chromosome, it creates the CERs, as explained in Section 5.1 (lines 4 to 10), and injects them into the compressed program $\hat{P}$, creating a program variant P' (line 11). This variant remains executable, as explained in 5.3. The algorithm measures the variant's quality to calculate the chromosome's fitness (line 12). If this fitness is better than the best variant found during the whole search so far, this record (P^*) of the best variant is updated (line 13). The fitness function, which guides the search, is the malicious class probability provided by the malware detector we seek to defeat. The search aims to minimize this value. Once the algorithm ends with the fitness computation of each chromosome, it applies four genetic operations to the population (lines 14 to 16): reproduction (chooses chromosomes for crossover), elitism selection (chooses the best chromosomes for use in the next generation), crossover (swaps a random number of common parameters between two chromosomes) and mutation (sets new values for a chromosome). These operations improve the population's quality using the information learnt from the previous generation. When there are no improve-

Table 9
Feature space, machine learning strategy and test accuracy for the 7 models we attack using EEE.

Technique	Features	Strategy	Accuracy	FNs
Structural Entropy (Baysa et al., 2013)	Entropy segments	Random Forest	91.6%	7.4%
EnTS	Entropy signatures	Random Forest	93.5%	9.4%
Kolter (Kolter & Maloof, 2006)	4-grams vector	Boosted J48	95.2%	0.0%
Kolter (Kolter & Maloof, 2006)	4-grams vector	SVM	89.3%	7.2%
Kolter (Kolter & Maloof, 2006)	4-grams vector	Boosted SVM	91.7%	6.4%
McBoost(Perdisci et al., 2008)	2-grams vector	Bagging J48	95.5%	2.8%
McBoots(Perdisci et al., 2008)	3-grams vector	Bagging J48	93.2%	6.8%

ments in the population, or after a fixed number of generations, the search process stops, and the algorithm returns P^*, the fittest individual (line 17).

5.3. Modifying UPX to produce EEE

EEE implementation is based on UPX (Oberhumer, Molnár, & Reiser, 2004) packer. UPX reformats binaries, compressing their sections and creating a new binary with three sections: (1) UPX0 an empty section where the code is uncompressed; (2) UPX1 the compressed original binary and the uncompression code (named stub); and (3) UPX2 a section containing all imports to properly run the binary.

When an UPX-packed binary is invoked, the stub in UPX1 executes and reconstructs the original binary by rebuilding the imports table using the imports in UPX2 and uncompressing the code in UPX1 into UPX0 (Sikorski & Honig, 2012). UPX uses the UCL compressor (Oberhumer et al., 2004). This compressor produces outputs with higher entropy, and consequently a n-gram distribution closer to uniformity than its input.

The adaptation of UPX to create EEE requires the manipulation of both the packing and unpacking processes in a synchronized way. The manipulation of the packing process is performed after the compression step when new space is available (Section 5.1). At this point, EEE reads the parameters for the CERs and creates new regions with different entropy densities. The positions of these regions in the binary depend on a Gaussian probability distribution (Fig. 3). The delimiters are set at the beginning and end of the regions. The manipulation of the unpacking process employs the stub, *i.e.* the assembly code injected into the packed file that will undo the packing process at runtime. Inside the stub, we include a step that identifies and eliminates the CERs before decompression. The identification process uses the inserted delimiters to find the CERs. Following these steps, we create executable static variants that in execution run just as their original programs.

6. Evaluating EEE

We built EEE to learn the limits of EnTS. Here, we conduct experiments using EEE to find these limits. We establish baselines by evaluating EEE againts SEnt and frequency-based techniques extracted from the literature. Unfortunately, we find that EEE comprenhensively defeats all the frequency-based techniques, including EnTS. We discuss the prospects for EnTS in Section 6.4. For EEE study, we use executable malware. Due to Kaggle malware has no PE headers, the binaries can not run. Therefore for EEE experiments we used directly the VirusShare dataset for training the detection algorithms (Section 3). The variants generated by EEE require a base binary, hence we sample the malware from this dataset to choose it, in order to ensure that it is known malware for the detectors.

6.1. Algorithms and parameters for EEE study

We have focused this part of the study on machine learning tools. In Table 9 the different techniques are listed. For each technique we also list the feature space (n-gram vectors or entropy) and the machine learning technique that is used during the classification phase. Kolter techniques (Kolter & Maloof, 2006) use the same feature space (4-grams) and authors report 3 classifiers that generates top results: Boosting combined with J48 trees, Support Vector machines and boosting combined with Support Vector Machines. In the case of McBoost (Perdisci, Lanzi, & Lee, 2008) the authors use dynamic analysis to generate an unpack version of the malware, however, this work is not focused on dynamic analysis, therefore we take the two classifiers used for the authors to decide if a binary is malware or not (authors named these classifier as C1 and C2). Structural entropy and EnTS are combined with Random Forest, as described in Section 4.1.

For the algorithms we have set the specific parameters specified in each paper, for those parameters that are not specified, we left the default parameters of the implementation. For training, we have used $\frac{2}{3}$ of the whole data and we leave $\frac{1}{3}$ of fresh data for testing.

The parameters for EEE are the following: the genetic algorithm has a population containing 50 individuals and evolves during 20 generations. In each generation the chromosomes are chosen for reproduction using a tournament process, while 10 individuals are chosen to pass directly to the next generation by elitism. Those that are chosen for reproduction used a two-point crossover operator whit a probability of 0.8 and the elements mutate with a probability of 0.1. When no changes are produced in the fitness value after 5 generations, the algorithm considers it as a convergence point and stops. For those parameters of UPX that are not controlled by the GA, we have set the default parameters. The search is also bounded for those parameters that have no maximum limit: the delimiter length is fixed to 8 bytes, the number of CER types is 10 and for each type the number of CERs goes from 0 to 20.

EEE focuses on entropy-based detectors, but it is more sensitive to some detectors than others, therefore Section 6.2 studies its effectiveness against different statistical detection techniques. After, we aim to understand its learning abilities. This is studied in Section 6.3.

We have trained the 7 techniques described in Section 6.1, and we measured their test accuracy and FNs, shown in Table 9.

6.2. EEE effectiveness against frequency-based detection.

Initially, our interest is to understand how the detectors can achieve high level of accuracy. Table 9 shows the accuracy for the classifiers.

EEE aims to defeat detectors based on entropy features, therefore initially, we ask: *"How effective is EEEagainst entropy-based malware detectors?"* For this experiment, we train two detectors using our malware corpus: Structural Entropy (SEnt) (Baysa et al.,

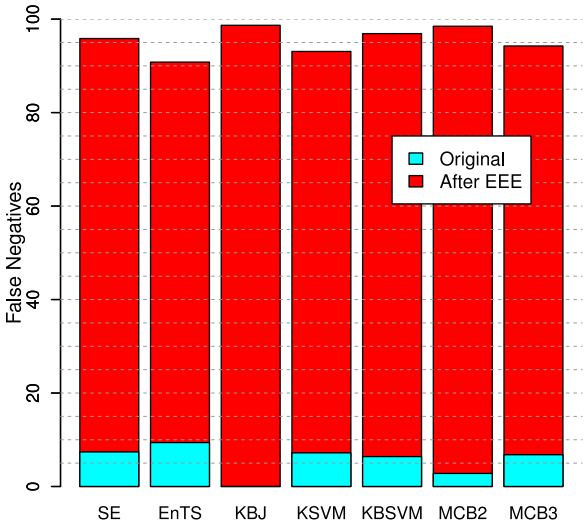


Fig. 5. False negative rate for the malware detectors. The blue bar represents the false negative rate on the corpus before applying EEE. The red bar represents the false negative rate on the same corpus after applying EEE. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

2013) and Entropy Time Series (EnTS). After, we sample 250 packed malware and 250 non-packed malware from the training corpus using an uniform distribution. EEE repacks these instances, originally classified as malware, until they are no longer detected, increasing the false negatives rate of the detector.

Fig. 5shows how the false negative rate is increased by EEE for Structural Entropy (SEnt) and the Entropy Time Series (EnTS) detection techniques. SEnt is more sensitive than EnTS to EEE growing from 7.4% false negatives to 95.9%, while EnTS grows from 9.4% to 90.8%. This supposes an increment of, at least, 80 points showing the effectiveness of EEE against these techniques.

EEE manipulates the original entropy signature used by both, EnTS and SEnt. It changes its shape to obtain the misclassification. SEnt operates with the segmentation generated by the signature, and compares two signatures with the Levenshtein distance, *i.e.*, the edition needed from the original signature to obtain the other, using the segments' entropy and size as the edition units. Once EEE manipulates the signature, the edition from the original to the variant is higher, this makes the variant more different, guiding to the misclassification of SEnt. EnTS considers the whole signature as a point in a multi-dimensional space. EEE manipulations translate the point in several dimensions, changing its possition with respect to the discrimination boundary. This is enough to determine that the two signatures are far from each other, and, as a consequence, they are considered different.

Once we have evaluated the effectiveness of EEE against entropy-based detection methods, we also want to ask about its effectiveness against other frequency-based detection techniques, therefore, we ask: *"How well does EEEwork against n-gram vector based detection techniques?"*

In this case, EEE is not targeting these methods, however, both, the compression and the injection of controlled entropy regions, manipulate the n-gram frequency, therefore they affect the detection.

For this experiment, we implemented 5 different n-gram vector based detection methods extracted from the literature. Their accuracy and false negative rates are in Table 9. In this case we have chosen three different classifiers sharing the same feature space and two different feature space sharing the same classifier. This decision helps to understand the importance of the feature space and the chosen classifier.

Fig. 5show the false negative rates for all the techniques. For the three Kolter's technique, we can see that, even when the boosting based classifiers are more accurate, according to Table 9, they are also more sensitive to EEE than SVM. According to the figure, boosting J48 and boosting SVM increase their false negative rates in 98.7 and 89.7, respectively, while SVM only increases its false negative rate 85.9 points. For McBoost's techniques, the increment is stronger in the 2-gram feature space than in the 3-gram feature space (95.7 and 87.5 points, respectively). These results show that both, the feature space and the classifier are sensitive places for attacking using techniques such as EEE, because, for Kolter's classifiers, SVM shows better results than the others using the same feature space. MCBoost's detectors show that the 3-gram feature space is stronger than the 2-gram feature space, using the same classification technique.

These results show that EEE performs modifications that affect the n-gram counting process. This changes the distribution, affecting specially the most frequent grams. These changes affect the classifiers, specially those sensitive to specific features, such as the most frequent one.

Findings. EEE is effective against entropy-based detection, and surprisingly, it is also effective against n-gram vectors based detection, incrementing the false negative rates, at least, 80 points with respect to the original rate.

6.3. *EEE learning process*

The previous section measures the abilities of EEE to evade n-gram vectors and entropy based detection techniques, however, we are also interested in measuring the effort of EEE to defeat these techniques. This effort can be measure in terms of the evolutionary process. Therefore we ask: *"How many generations does the evolutionary process need to defeat a detection technique?"*

For this experiment we use the same setup of the previous section, and we increment the granularity to the number of generations. By design, all the classifiers detect malware when its malicious probability is higher than 0.5. Then, considering a population of EEE parameters, this experiment aims to measure when these parameters are properly set to generate variants that always evade the classifier. In terms of search, this is consider as a convergence point, therefore we want to find the convergence point of the evolutionary algorithm.

Fig. 6shows the evolution of the median detection probability of the whole population over a number of generations. The gray line in 0.5, represents the boundary between being detected as malware (over 0.5 probability) and not being detected as malware (under 0.5 probability). The figure shows that during the first generation at least one technique goes under 0.5 probability (McBoots detector using 2-grams). In the fourth generation, there is a strong decaying tendency for all the techniques. From the seventh generation, no technique is over the threshold. This shows how the learning process is reducing the detection abilities of all techniques, but it also shows that the behaviour of EEE is different for the different techniques.

It is important to remark that some techniques, such as McBoost's classifiers, have a stepped tendency. Analysing the classifier feedback during the evolutionary process, we discovered that it provides discreet values, generating a fixed set of detection levels. Kolter's and entropy based detection techniques have a continuous tendency. In this case, the classifier generates continues values. Continuity is better for the search process, due to it is easier to find a gradient by learning.

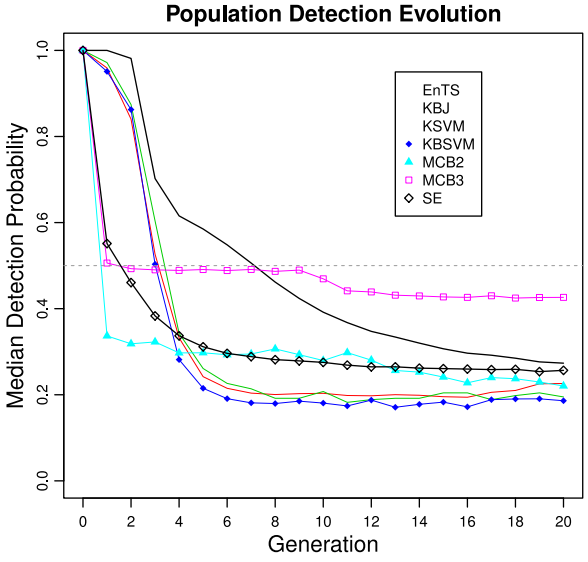


Fig. 6. Evolution of the median detection probability for the variants created using EEE by generation.

It is also interesting to remark that several techniques have an asymptotic behaviour close to 0.2. The search process focuses on defeating the classifier finding vulnerabilities on it, that is, areas where our modifications might generate a misclassification, but the variants used in this experiment come from the original training data of the classifier. Due to the classifier knows the original malware, the modifications are bounded by this knowledge. Trying to set this value from these variants is an open problem.

Findings. EEE learns to defeat detectors reaching its convergence point in less than 8 generations of the search process. The most resisting technique against EEE is EnTS.

6.4. Prospects for EnTS

EEE comprehensively defeats EnTS and the other frequency-based detectors from the state of the art. The next step in the arms race is how to improve EnTS to defeat EEE. There are potential improvements in different directions. First, the packing process of EEE is not protected against sophisticated unpacking techniques. Using one of these technique will remove the CERs exposing the original malware. Second, EEE constantly attacks the detection technique. Adding an extra protection to the classifier for detecting small variations on adversarial queries, might give it the ability of detecting an adversarial attack. However, a smart adversary could include some dummy variants to cheat this adversary detector. Finally, EnTS can specialize itself in detecting EEE, finding specific features of EEE variants that can detect an attack.

7. Related work

This work examines the prospects for frequency-based malware detection by taking two steps in the malware arms race. First, it introduces EnTS to advance the state of the art in frequency-based detection and, then, immediately creates EEE to advance the state of the art in evasion by defeating EnTS. This section contextualizes both tools with respect to the literature. First, we discuss malware detection in general, before turning prior specifically on frequency-based detection. We use frequency-based detection to explore the arms race, so we close with adversarial machine learning.

7.1. Dynamic and static analysis in the malware arms race

Somayaji described the cybersecurity arms race as a coevolution between the black hats and the white hats (Somayaji, 2004). He explained that this is a competition with both sides learning from each other, but he did not model this phenomena or study it. Guerra *et al.* did study this coevolution in the context of spam e-mails (Guerra et al., 2010). They studied this from the perspective of white hats, focusing on a tool for spam filtering and a dataset of spam across 12 years. This tool, and similar tools, forced spammers to evolve. At the same time, the filters also improved their features to detect new spam. This study provided evidence of the arms race coevolution, and we based our work in a similar idea, focused on modern machine learning detectors.

Our main detection scenery is disk resident malware, prevalent specially in Windows machines. Windows malware has become increasingly sophisticated at hiding itself and resisting analysis. The literature contains seveal works mainly focused on static, dynamic or hybrid techniques aiming to detect it.

Static analysis, whether based on abstractions of Control Flow Graphs and program semantics (Preda, Christodorescu, Jha, & Debray, 2007) or on opcode analysis (Santos et al., 2010), or focused on PE Headers and Static API Calls (Xu, Sung, Mukkamala, & Liu, 2007; Ye, Li, Jiang, & Wang, 2010) as features for machine learning, faces the increasing difficulty of initial reverse engineering. In addition, Moser et al. demonstrated hard limits to the ability of static anaysis to deal with obfuscation (Moser, Kruegel, & Kirda, 2007). Dynamic analysis via virtual machines and sandboxes can avoid anti-disassembly measures but suffer from resistance via dynamic defence predicates and red pill environment detection techniques (Paleari, Martignoni, Roglia, & Bruschi, 2009). Windows malware analysis aiming to integrate dynamic and static analysis, as Santos, Devesa, Brezo, Nieves, and Bringas (2012), Islam, Tian, Batten, and Versteeg (2013) and andA. Salim (2015), to produce features for data mining approaches suffer the same problems.

Recent approaches to Android malware exploit the relative lack of sophistication of that type of malware. These include Drebin (Arp et al., 2014), CopperDroid (Tam, Khan, Fattori, & Cavallaro, 2015), which combine machine learning with behavioural models. Other tools as DroidSIFT (Zhang, Duan, Yin, & Zhao, 2014) are focused on anomaly detection and malware family classification.

Malware detection tools focused on network neighbourhoods, for example, Nazca (Invernizzi et al., 2014) and AESOP (Tamersoy, Roundy, & Chau, 2014) show real promise in terms of scale and accuracy but require ground truth as a seed, just as our similarity techniques do. Also, the work of Zhongqiang Chen et al. focuses on how malware is propagate on these networks (Chen et al., 2012). Our work focuses on frequency-based detectors, that have the ability of detecting malware before any static or dynamic analysis.

7.2. Frequency-based detection

In 1994 Kephart presented an n-gram approach for extracting signatures but reported no results (Kephart, 1994). In 2001, Schultz et al. used several data mining techniques on binaries to distinguish between benign and malicious executables in Windows or MS-DOS format. Memory consumption was a scalability bottleneck. They experimented on a dataset of 3265 malware and 1001 benign files but lacked fresh data for testing. Validation achieved 97.11% Accuracy with 3.8% FP rate (Schultz, Eskin, Zadok, & Stolfo, 2001).

In 2006, Kolter and Maloof used Information Gain combined with byte level analysis of n-grams to classify and detect malware. Again they did not use fresh data for the test phase. They experimented on two small datasets, one of 476 malware, 561 benignware (95% accuracy with 5% FP in validation); the second of 1971

benign-ware, 1651 malware (94% accuracy and 1% FP in validation) (Kolter & Maloof, 2006).

In 2007, Lyda and Hamrock used the average entropy of a whole file and the entropy of specific code sections (discovered only by using static analysis). They showed that binary files with a higher entropy score tend to be correlated with the presence of encryption or compression. They compared more than 20,000 malware to check whether they are able to detect these concealment methods but did not consider malware detection (Lyda & Hamrock, 2007). In the same year, Stolfo et al. used 1-gram and 2-gram byte distributions for a file to compare it with different filetype models for filetype identification (Li, Wang, Santos, & Herzog, 2005) in malware detection within DOC and PDF files. They reported on experiments with over 140 pdfs and 361 benign and 616 malware and results with between 3% to 20% of false positives but no accuracy information. This work considered n-grams in a vector space, using their frequency and variation as features, but each dimension was a n-gram resulting in exponential increase in the number of dimensions (Stolfo, Wang, & Li, 2007).

Tabish et al. in 2009 divided files into blocks, and calculated frequency of n-gram histograms for each block, then extracted statistical and information-theoretic features from the histogram to define a feature vector per block. They used this to classify a feature vector as normal or potentially malicious. Pairwise comparison between blocks of different files reduces the scalability of this approach. They claimed an accuracy rate of 90% with a False Positive rate of around 10% (Tabish, Shafiq, & Farooq, 2009).

Santos et al. in 2011 introduced a semi-supervised methodology to reduce the labelling process. Their n-gram vector was the frequency of all possible n-grams, an important scalability limitation. After experiments on 1000 malware and benign-ware, they reported 89% of accuracy with 10% of false positives (Santos, Nieves, & Bringas, 2011).

Finally, Sorokin presented SEnt in 2011 (Sorokin, 2011). The first evaluation of SEnt was an use case comparison between two files. After, Baysa *et al.* extended it to metamorphic malware in 2013 (Baysa et al., 2013), showing that this technique scales quadratically. This was a consequence of the implicit pairwise comparison of the metric. Another relevant bottleneck, that the authors identified in the technique, was the definition of the segments that describe the files. This definition depends on three parameters whose setting depends on the analyst. The Levenshtein distance, applied during the files comparison, depends directly on the parameters. They directly affect the number of segments that will pass to this metric, affecting to the performance. EnTS is free of this parametrization, leveraging directly the properties of the wavelet to speed up the comparison and scale linearly.

EnTS has three advantages over previous work in detection via byte level content: (1) better accuracy combined with lowenr false positive rates, (2) better (linear) scalability in the detection phase, and (3) a more rigorous experimental approach. Nevertheless, EnTS is sensitive to adversarial machine learning, introduced in the next section.

7.3. Adversarial machine learning

Adversarial machine learning inspired our step forward into the arms race. This field aims to exploit the vulnerabilities of a learning system, attacking the test data distribution and making it different to the training data (Moreno-Torres, Raeder, Alaiz-RodríGuez, Chawla, & Herrera, 2012). The adversary introduces noise into the data or makes some other alteration to achieve a misclassification. This sensitivity was originally noticed on spam detectors (Chinavle, Kolari, Oates, & Finin, 2009), where the adversary studied different modifications to emails to enable the passing of machine learning based filters.

Xu, Qi, and Evans (2016) were the first authors applying these models to malware. They created evademl, a genetic programming tool that modifies pdf malware to cheat two machine learning based detectors, extracted from the literature: Hidost (Šrndic & Laskov, 2013) and PDFrate (Smutz & Stavrou, 2012). In this work, the authors knew the features used by the machine learning algorithm, the classifiers, and the training data. In particular, they had access to the classification probabilities, providing them with a search gradient per classifier. In addition, they were effectively working off-line with no evolution on the part of the detectors. In our experiments, EEE did not use any information about the training data or the detector features it attempted to attack. Moreover, EEE creates variants for Windows binary executable malware that is protected against disassembly or reassemble, while evademl manipulates PDF malware. Furthermore, since UPX is compatible with several different architectures, EEE can potentially be adapted to several different platforms (Oberhumer et al., 2004). More recent works, as the one introduced by Calleja, Martín, Menéndez, Tapiador, and Clark (2018) apply adversarial machine learning to cheat the triage process of malware analysis.

Adversarial machine learning has been also consolidated as an analytical process to measure the degree to which different machine learning algorithms can be exploited. A good example is the work of Biggio et al. who studied different vulnerabilities for Support Vector Machines. They also presented a methodology to improve the robustness of this classification technique (Biggio, Nelson, & Laskov, 2012). They extended this work to another classifier, where they also formalized the language for adversarial models (Biggio, Fumera, & Roli, 2014). While important, this work is tangential to this paper as we only used access to the classification output as a fitness function.

8. Conclusions

We have demonstrated that EnTS outperform previous information theoretic similarity measures. Its level of abstraction makes it difficult to counter and it offers scalability advantages. We have demonstrated excellent precision and accuracy on a representative mixture of malware types drawn from the Kaggle malware data and VirusShare. Indeed, EnTS outperforms existing AntiVirus engines (as represented in VirusTotal) for accuracy and precision. Its time complexity is bounded above by the number of files to be classified. As an automated, execution agnostic, string-based similarity metric it offers wider scalability advantages beyond its time complexity class alone – reducing human effort and reducing the need for dynamic or static analysis.

EEE also demonstrated its ability to increment false negatives on entropy and n-gram based detectors. It learns from them, creating variants whose properties are unknown to the classifier or similar to benign-ware. It is the first packer with the ability to learn about its concealment strategy.

Acknowledgments

This work has been supported by the next research projects: SeMaMatch EP/K032623/1, DAASE EP/J017515/1, LUCID EP/P005659/1 and InfoTestSS EP/P006116/1 from EPSRC.

References

Addison, P. S. (2002). *The illustrated wavelet transform handbook: introductory theory and applications in science, engineering, medicine and finance*. CRC press.

Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., & Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the annual symposium on network and distributed system security (ndss)*.

Baysa, D., Low, R. M., & Stamp, M. (2013). Structural entropy and metamorphic malware. *Journal of computer virology and hacking techniques, 9*(4), 179–192.

Bhattacharya, S., Menéndez, H. D., Barr, E., & Clark, D. (2016). Itect: Scalable information theoretic similarity for malware detection. arXiv:1609.02404.

Biggio, B., Fumera, G., & Roli, F. (2014). Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering, 26*(4), 984–996.

Biggio, B., Nelson, B., & Laskov, P. (2012). Poisoning attacks against support vector machines. In *Proceedings of the 29th international conference on machine learning*.

Brockwell, P. J., & Davis, R. A. (2013). *Time series: theory and methods*. Springer Science & Business Media.

Calleja, A., Martín, A., Menéndez, H. D., Tapiador, J., & Clark, D. (2018). Picking on the family: Disrupting android malware triage by forcing misclassification. *Expert Systems with Applications, 95*, 113–126.

Chatfield, C. (2000). *Time-series forecasting*. CRC Press.

Chen, Z., Roussopoulos, M., Liang, Z., Zhang, Y., Chen, Z., & Delis, A. (2012). Malware characteristics and threats on the internet ecosystem. *Journal of Systems and Software, 85*(7), 1650–1672.

Chinavle, D., Kolari, P., Oates, T., & Finin, T. (2009). Ensembles in adversarial classification for spam. In *Proceedings of the 18th acm conference on information and knowledge management* (pp. 2015–2018). ACM.

Cover, T. M., & Thomas, J. A. (2012). *Elements of information theory*. John Wiley & Sons.

Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM, 55*(10), 78–87.

Dua, S., & Du, X. (2016). *Data mining and machine learning in cybersecurity*. CRC press.

Guerra, P. H. C., Guedes, D., Meira, J. W., Hoepers, C., Chaves, M., & Steding–Jessen, K. (2010). Exploring the spam arms race to characterize spam evolution. In *Proceedings of the 7th collaboration, electronic messaging, anti-abuse and spam conference (ceas)*.

Hothorn, T., Lausen, B., Benner, A., & Radespiel-Tröger, M. (2004). Bagging survival trees. *Statistics in medicine, 23*(1), 77–91.

Invernizzi, L., Miskovic, S., Torres, R., Saha, S., Lee, S., Mellia, M., Kruegel, C., et al. (2014). Nazca: Detecting malware distribution in large-scale networks. In *Proceedings of the network and distributed system security symposium (ndss)*.

Islam, M. R., Tian, R., Batten, L. M., & Versteeg, S. (2013). Classification of malware based on integrated static and dynamic features. *J. Network and Computer Applications, 36*(2), 646–656. doi:10.1016/j.jnca.2012.10.004.

Kephart, J. O. (1994). A biologically inspired immune system for computers. In *In artificial life iv: Proceedings of the fourth international workshop on the synthesis and simulation of living systems* (pp. 130–139). MIT Press.

Kolter, J. Z., & Maloof, M. A. (2006). Learning to detect and classify malicious executables in the wild. *The Journal of Machine Learning Research, 7*, 2721–2744.

Li, M., & Vitányi, P. (2013). *An introduction to kolmogorov complexity and its applications*. Springer Science & Business Media.

Li, W., Wang, K., Santos, I., & Herzog, B. (2005). Fileprints: identifying filetypes by n-gram analysis. In *Information assurance workshop, usa, ieee press* (pp. 67–71). doi:10.1109/IAW.2005.1495935.

Li, X., Loh, P. K., & Tan, F. (2011). Mechanisms of polymorphic and metamorphic viruses. In *Intelligence and security informatics conference (eisic), 2011 european* (pp. 149–154). IEEE.

Liao, T. W. (2005). Clustering of time series data-a survey. *Pattern recognition, 38*(11), 1857–1874.

Lonas, H. (2016). The webroot 2016 threat brief: Next-generation threats exposed. http://webroot-cms-cdn.s3.amazonaws.com/7814/5617/2382/Webroot-2016-Threat-Brief.pdf.

Lyda, R., & Hamrock, J. (2007). Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy, 5*(2), 40–45. doi:10.1109/MSP.2007.48.

Maaten, L. v. d., & Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research, 9*(Nov), 2579–2605.

Moreno-Torres, J. G., Raeder, T., Alaiz-RodríGuez, R., Chawla, N. V., & Herrera, F. (2012). A unifying view on dataset shift in classification. *Pattern Recognition, 45*(1), 521–530.

Moser, A., Kruegel, C., & Kirda, E. (2007). Limits of static analysis for malware detection. In *Computer security applications conference, 2007. acsac 2007. twenty-third annual* (pp. 421–430). IEEE.

Oberhumer, M., Molnár, L., & Reiser, J. F. (2004). Upx: the ultimate packer for executables.

Paleari, R., Martignoni, L., Roglia, G. F., & Bruschi, D. (2009). A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the usenix workshop on offensive technologies (woot): 41* (p. 86).

Perdisci, R., Lanzi, A., & Lee, W. (2008). Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Annual computer security applications conference, 2008* (pp. 301–310). IEEE.

Preda, M. D., Christodorescu, M., Jha, S., & Debray, S. K. (2007). A semantics-based approach to malware detection. In *Proceedings of the 34th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2007, nice, france, january 17–19, 2007* (pp. 377–388). doi:10.1145/1190216.1190270.

Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., & Ahmadi, M. (2018). Microsoft malware classification challenge. arXiv:1802.10135.

andA. Salim, P. S. (2015). Integrated static and dynamic analysis for malware detection. *Procedia Computer Science, 46*, 804–811.

Santos, I., Brezo, F., Nieves, J., Penya, Y. K., Sanz, B., Laorden, C., et al. (2010). Idea: Opcode-sequence-based malware detection. In *Engineering secure software and systems, second international symposium, essos 2010, pisa, italy, february 3–4, 2010. proceedings* (pp. 35–43). doi:10.1007/978-3-642-11747-3-3.

Santos, I., Devesa, J., Brezo, F., Nieves, J., & Bringas, P. G. (2012). OPEM: A static-dynamic approach for machine-learning-based malware detection. In *International joint conference cisis'12-iceute'12-soco'12 special sessions, ostrava, czech republic, september 5th-7th, 2012* (pp. 271–280). doi:10.1007/978-3-642-33018-6_28.

Santos, I., Nieves, J., & Bringas, P. G. (2011). Semi-supervised learning for unknown malware detection. In *International symposium on distributed computing and artificial intelligence* (pp. 415–422). Springer.

Schultz, M. G., Eskin, E., Zadok, E., & Stolfo, S. J. (2001). Data mining methods for detection of new malicious executables. In *2001 IEEE symposium on security and privacy, oakland, california, USA may 14–16, 2001* (pp. 38–49). doi:10.1109/SECPRI.2001.924286.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell Systems Technical Journal, 27*. 379–423 (Part I)623–656 (Part II)

Sikorski, M., & Honig, A. (2012). *Practical malware analysis: The hands-on guide to dissecting malicious software*. No Starch Press.

Smutz, C., & Stavrou, A. (2012). Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th annual computer security applications conference* (pp. 239–248). ACM.

Somayaji, A. (2004). How to win an evolutionary arms race. *IEEE security & privacy, 2*(6), 70–72.

Sorokin, I. (2011). Comparing files using structural entropy. *Journal in computer virology, 7*(4), 259–265.

Šrndic, N., & Laskov, P. (2013). Detection of malicious pdf files based on hierarchical document structure. In *Proceedings of the 20th annual network & distributed system security symposium*.

Stolfo, S. J., Wang, K., & Li, W. (2007). Towards stealthy malware detection. In *Malware detection* (pp. 231–249). doi:10.1007/978-0-387-44599-1-11.

Tabish, S. M., Shafiq, M. Z., & Farooq, M. (2009). Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD workshop on cybersecurity and intelligence informatics, paris, france, june 28, 2009* (pp. 23–31). doi:10.1145/1599272.1599278.

Tam, K., Khan, S. J., Fattori, A., & Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *Proc. of the symposium on network and distributed system security (ndss)*.

Tamersoy, A., Roundy, K., & Chau, D. H. (2014). Guilt by association: Large scale malware detection by mining file-relation graphs. In *Proceedings of the 20th acm sigkdd international conference on knowledge discovery and data mining KDD '14*. ACM.

Xu, J., Sung, A. H., Mukkamala, S., & Liu, Q. (2007). Obfuscated malicious executable scanner. *Journal of Research and Practice in Information Technology, 39*(3), 181–198.

Xu, W., Qi, Y., & Evans, D. (2016). Automatically evading classifiers. In *Proceedings of the 2016 network and distributed systems symposium*.

Ye, Y., Li, T., Jiang, Q., & Wang, Y. (2010). CIMDS: Adapting postprocessing techniques of associative classification for malware detection. *IEEE Transactions on Systems, Man, and Cybernetics, Part C, 40*(3), 298–307. doi:10.1109/TSMCC.2009.2037978.

Zhang, M., Duan, Y., Yin, H., & Zhao, Z. (2014). Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 acm sigsac conference on computer and communications security* (pp. 1105–1116). ACM.