Forensic Science International: Digital Investigation 37 (2021) 301192

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2021 USA - Proceedings of the Twenty First Annual DFRWS USA

Malware family classification via efficient Huffman features

Stephen O'Shaughnessy [a, *], Frank Breitinger [b, 1]

[a] Department of Informatics, Technological University Dublin, Blanchardstown Campus, Dublin 15, Ireland
[b] School of Criminal Justice, University of Lausanne, 1015, Lausanne, Switzerland

ARTICLE INFO

Article history:

Keywords:
Malware classification
Huffman encoding
Malware abstraction
Feature construction
Compression
Machine learning

ABSTRACT

As malware evolves and becomes more complex, researchers strive to develop detection and classification schemes that abstract away from the internal intricacies of binary code to represent malware without the need for architectural knowledge or invasive analysis procedures. Such approaches can reduce the complexities of feature generation and simplify the analysis process. In this paper, we present *efficient Huffman features* (eHf), a novel compression-based approach to feature construction, based on Huffman encoding, where malware features are represented in a compact format, without the need for intrusive reverse-engineering or dynamic analysis processes. We demonstrate the viability of eHf as a solution for classifying malware into their respective families on a large malware corpus of 15 k samples, indicative of the current threat landscape. We evaluate eHf against current compression-based alternatives and show that our method is comparable or superior for classification accuracy, while exhibiting considerably greater runtime efficiency. Finally we demonstrate that eHf is resilient against code reordering obfuscation.

1. Introduction

Developing adequate identification and eradication schemes to combat malware is a major challenge nowadays, due to the ever-increasing complexities of such malicious programs. From a malware forensics perspective, an essential step is the classification of malware, which helps to better understand its impact. Ideally, a classification scheme would arrange malware types into taxonomic groupings based on characteristic traits shared between members of the same class, such as shared code sections (Choi et al., 2008), similar API call sequences (Saxe et al., 2012) or API frequencies (Hansen et al., 2016). Much prior work has been presented in this field, demonstrating a wide range of approaches. By and large, these methods use signatures as feature vectors extracted from either the static code or dynamic behaviours of the malware samples. This requires intricate knowledge of the malware's internal binary structure, which can involve labor-intensive manual analysis to isolate the signatures. Furthermore, correct feature selection is a crucial step in this process, as choosing incorrect features will negatively impact the overall performance of the classification algorithms. New approaches that require no prior specialist subject-specific or background knowledge can help simplify the process by negating the need for specific feature extraction measures, while retaining important or relevant signature information.

The use of compression algorithms, such as LZMA, BZ2 and Zlib have been used in machine learning tasks such as clustering and classification, due to their properties which reduce the need for explicit feature extraction and selection. Prior work has demonstrated that compression algorithms map input data sequences to a defined and reduced feature space. This infers that the outputs from compression tools can be used to represent feature vectors for machine learning tasks. In this regard, compression has been be applied to machine learning tasks in many domains, such as image classification (Sanchez and Perronnin, 2011), text classification (Marton et al., 2005) and DNA sequencing (Vinitha et al., 2016).

In particular, prior work has focused on the formulation and application of similarity metrics based on compression. One such metric that has featured predominantly in this domain is Normalized Compression Distance (NCD) (Li et al., 2004). The theoretical

* Corresponding author.
E-mail addresses: stephen.oshaughnessy@tudublin.ie (S. O'Shaughnessy), Frank.Breitinger@unil.ch (F. Breitinger).
[1] https://www.FBreitinger.de

https://doi.org/10.1016/j.fsidi.2021.301192

justification for NCD is founded upon an approximation of the Kolmogorov (1998) complexity. NCD computes the similarity distance between two input sequences using a fixed formula expressed in terms of the compressed versions of the sequences both separately and combined. NCD's developers have demonstrated its application to a wide variety of domains, such as text processing, genomics, virology, music, and languages (Cilibrasi and Vitanyi, 2005). In the malware domain, NCD has been applied areas such as phylogeny (Walenstein et al., 2017) and detection (Alshahwan et al., 2015).

Despite its desirable properties, NCD-based approaches have proven infeasible for large data corpora, due to the computational complexity in pair-wise comparisons where each comparison involves calculating both the compression of each input data sequence and the compression of the concatenated pair. Moreover, Schuller Borbely (2016) demonstrated the performance of NCD is heavily dependent on the choice of compression algorithm and increasing file size is a factor that hampers the performance of NCD with these compression algorithms.

In this paper, we present a compression-based technique, based on the Huffman coding algorithm: an efficient, unambiguous statistical code that is commonly used for lossless data compression (Huffman, 1952). Although this technique can be applied to multiple domains, we focus on its application to malware family class classification. We evaluate our eHf method against a malware dataset comprising approximately 15 k samples from recent VirusTotal (VT) repositories, which are representative of current malware strains. For classification, k-nearest neighbors (KNN) classifier is used to validate our approach. Using the standard Minkowski distance metric, we demonstrate our Huffman features provide superior classification performance over NCD with a computation time of up to 3 orders of magnitude faster. We also evaluate our method against *Lempel-Ziv Jaccard Distance*, a recent compression-based metric proposed by Raff and Nicholas (2017a) and show our method produces comparable classification performance but is approximately five times faster in generating features and over twice as fast in classification model training on the VT dataset. Our code is open-source and fully available.[2] Additionally, the full or partial VT malware dataset is available to researchers on request. In summary, this research makes the following contributions:

- *Efficient Huffman features* (eHf): fixed-length feature vectors derived from Huffman encoding which provide a feature representation of input binary data in a compressed format;
- *Non-domain-specific*: since no internal or structural knowledge of the data is required, our eHf method can be applied to multiple domains outside of malware classification;
- *Machine learning ready*: eHf features are stored in an array structure, so can be applied to a wide range of machine learning algorithms.

The paper is organized as follows: Sec. 2 discusses the related works in compression-based approaches to malware detection and classification; Sec. 3 introduces the Huffman coding algorithm and outlines our feature construction method, including how we optimize the feature sets; Sec. 4 details our methodology; Sec. 5 presents our analysis approach and results. In this section, we test the efficacy of our Huffman feature generation method as a means of malware classification. We also perform comparative analyses with state-of-the-art compression techniques and finally demonstrate our method is resilient against code reordering obfuscation; Sec. 6 provides a discussion on the significance of our findings and finally we conclude in Sec. 7.

[2] Repository: https://anonymous.4open.science/r/068ed7f6-7957-4440-87cf-350b71e8feae/

2. Background and related work

One of the challenges in machine learning classification is the choice of an appropriate representation for the data. With the proliferation of malware and in particular the increasing complexities of recent campaigns, many classification schemes must be adapted to meet these demands. Malware researchers are moving away from the reliance on pattern matching schemes to generate signatures for classification by developing ways to represent malware in an abstract format that does not require invasive measures such as reverse engineering. Previously, techniques such as approximate matching or image processing have been considered in this regard. For example, Breitinger et al. (2014) presented saHash: an algorithm whose final fingerprint is composed of four sub-hash functions values."Given the hash values of two byte sequences, saHash returns a lower bound on the number of Levenshtein operations between the two byte sequences as their similarity score." Nataraj et al. (2011) used image representations of malware to classify samples into their family classes. Dubbed "byteplots", the images were generated using a byte-to-grayscale pixel mapping, resulting in textural images from which salient features were extracted for classification.

In contrast, this paper focuses on compression-based applications, particularly applied to the malware space. Compression represents data in a compact or reduced format, without any *subject-specific knowledge*, so it has potential as a feature space representation. The fact that compression can be applied to data in any format means it can be adapted to a spectrum of domains. Several previous works exist in this field, either in malware detection (differentiating between benign and malicious programs) or malware classification (grouping similar malware into groups, based on common structural or behavioural traits).

2.1. Compression as a distance metric

Many previous compression-based approaches in the malware domain are based on using compression as a distance metric to measure similarity, in particular, the normalized compression distance. NCD works on the idea that the similarity of two objects can be measured by the ease with which one can be transformed into the other. NCD is formally measured in Eq. (1), where C denotes a compression function and x, y are inputs:

$$NCD(x,y) = \frac{C(xy) - min(C(x), C(y))}{max(C(x), C(y))}. \quad (1)$$

If $x = y$, then the following is true: $C(xy) = C(y) = C(x)$ where xy denotes the concatenated inputs. NCD generates a non-negative value $0 \leq NCD(x,y) \leq 1$ where 0 indicates identical sequences and 1 denotes complete dissimilarity.

Wehner (2005) used NCD to analyse and detect different species of worms. Firstly, NCD was used as a similarity metric to cluster worms by family type. Unknown samples were compared to known worms for detection purposes. Next, detected anomalies in traffic by comparing the compressibility of each network session capture. Finally, compression was used to detect malicious network sessions that exhibited similarities to known intrusion attempts. Two Snort IDS plugins were provided to demonstrate the former approaches. Wehner does not report on the computational run-time performance, nor attempts to implement any classification scheme for the methods presented.

Alshahwan et al. (2015) built a malware classification scheme by constructing approximated features, based on NCD measurements between benign and malicious samples. The first feature in each vector was the ratio of the program's compressed size to its uncompressed size. The remaining n features were obtained by calculating the program's NCD to n other reference programs, comprising $n/2$ benign and $n/2$ malicious samples, chosen at random. The resulting feature set was trained on a corpus of labelled malware and benign-ware using a Random Forest classifier. The authors note the run-time computational complexity of NCD was non-trivial and addressed this by reducing the number of compression comparisons through a lower bound threshold. This threshold was derived from the compression property that states $Z(xy) = Z(x)$, where Z is some compression algorithm and x is an input sequence that fully contains a shorter sequence y, giving $1 - Z(x)/Z(y)$. A classification accuracy of 97.1% was reported using the described method.

Besides computational complexity issues, the choice of compression algorithm has impact on NCD as an effective distance metric. Cilibrasi and Vitanyi (2005) placed several attributes that compression algorithms must possess to ensure the properties of NCD as a similarity metric. Definition 3.1 by Cilibrasi and Vitanyi (2005) states a compressor is "normal" if it exhibits the following qualities:

1. *Idempotency*: $C(xx) = C(x), and C(\lambda) = 0$
2. *monotonicity*: $C(xy) \geq C(x)$
3. *Symmetry*: $C(xy) = C(yx)$
4. *Distributivity*: $C(xy) + C(z) \leq C(xz) + C(yz)$

where x, y are data input sequences and λ is an empty string. However, Schuller Borbely (2016) demonstrated that several compression algorithms, e.g., LZMA, BZ2, Zlib, and PPMZ, fail to satisfy the properties of a normal compressor for use with NCD, as defined above. Using a KNN classifier, it was shown that for larger malware binaries up to 15 MB in size, LZMA performed best, though classification accuracy was still low, at 59.7%. The best results were produced from the BZ2 algorithm, yielding 89.7% classification accuracy with smaller files up to 200 KB in size. The author offered a method to improve performance through interleaving and string sequence alignment, but this was still limited to order-dependent similarity and had a time complexity proportional to the product of the file sizes input and thus was quite slow with large files.

Apel et al. (2009) evaluated several distance metrics, including Levenshtein, Manhattan and NCD algorithms for malware similarity detection. The compression algorithms used to compute the NCD methods were LZMA and prediction by partial matching (PPM). The distance metrics were tested under the following categories: order sensitivity, appropriateness and run-time performance. The authors reported that the compression-based NCD metrics performed worst, even failing to detect shared behaviours in a malware corpus of 1195 samples. Additionally, run-time performance was infeasibly slow for larger samples. They recommended NCD should not be used as a similarity metric for malware detection.

Raff and Nicholas (2017a) presented an alternative metric to NCD, called the Lempel-Ziv Jaccard Distance (LZJD), based on the LZ77 algorithm and Jaccard distance. LZJD makes use of the Lempel-Ziv technique for creating a compression dictionary of byte subsequences, dubbed a LZSet. The authors note that the set of subsequences extracted by LZJD requires memory proportional to the size of the input strings, which makes storage impractical. To overcome this, they use min-hashing to create compact representations of the input strings, converting each sub-string to integers (hashes) and then choosing the k-smallest features. An error rate of 3% was achieved with $(k = 1024)$. The similarity metric is derived from an approximation of the Jaccard distance metric. The authors evaluated their method against NCD on several datasets including the Kaggle Malware Classification Challenge (Ronen et al., 2018) and Drebin Android datasets (Arp et al., 2014). A KNN classifier was used with 10-fold cross-validation on the data. Results reported showed LZJD outperformed NCD on both datasets, with performance speedup of up to four magnitudes faster.

2.2. Compression as a feature space

The research discussed thus far comprised different compression-based approaches applied to the malware space. In particular, they used compression algorithms to determine similarity through the NCD distance metric, or Jaccard index in the case of LZJD. However, none has explicitly considered the outputs of these compression algorithms as features for classification. Sculley and Brodley (2006) showed that compression algorithms possess such properties, which allow mappings from input sequences to vectors within an implicit feature space. For example, they showed the Lempel-Ziv-based compressors, such LZ77 and LZ78, compress data into dictionaries of substrings, which can be engineered as feature vectors. Paskov et al. (2013) build on these findings, presenting a dictionary-based compression scheme to represent features for unsupervised text classification. The scheme operates by scanning each document from left to right, consuming characters until it has found the longest prefix matching a previously seen substring. It then outputs a pointer to that previous instance, which is interpreted as a feature. The process continues with the remaining input string and if no prefix matches, the single next character is output. Using 10-fold cross validation on the training data, they report classification accuracies of 83% and 90.4% on the 20 Newsgroups (Crawford, 2016) and IMDb (Maas et al., 2011) datasets, respectively.

Raff and Nicholas (2017b) presented an improvement on LZJD called Stochastic Hashed Weighted Lempel-Ziv (SHWel). The byte sequences produced by the original LZJD algorithm were fixed and a weighting applied. To improve runtime calculation, feature hashing via the hashing trick was used. The hashing trick is a fast and space-efficient way of turning arbitrary features into a sparse binary vector. It enables dimensionality reduction because the hash function is used to determine the feature's location in a vector of lower dimension. Finally, similarly to LZJD, the k-smallest hashes were chosen. A stochastic component was also introduced to the vectorization, so that the algorithm could be robust against imbalanced data. By comparing the two algorithms for performance in malware family classification and detection, using the Kaggle and Drebin datasets, it was demonstrated the SHWel algorithm outperformed LZJD on balanced and unbalanced data. Out of the research reviewed, only the SHWel method provides a feature vector set that can be used for classification.

The majority of the compression-based approaches reviewed use the Lempel-Ziv (LZ) algorithm or one of its variations in their calculations. However, the findings of Sculley and Brodley (2006) showed that the LZ feature space is very large. For example, the Lempel-Ziv variant LZ77 encodes substrings as a series of output codes that reference previously occurring substrings in a sliding dictionary window. The result of this encoding is a high dimensional feature space of $O(2^m)$, where m is the maximum length of the repeated substrings found by LZ77. The LZW algorithm, another LZ variant, builds and stores an explicit substring dictionary on the fly, selecting new substrings to add to the dictionary using a greedy selection heuristic. The maximum length of a substring in the LZW dictionary, using the LZW substring selection heuristic, is $O(2^c)$, where c is the fixed-length output code produced by the LZW encoding. The vector space has one dimension for each possible substring with maximum length $O(2^c)$, that is, $O(2^{(2^c)})$ dimensions, so is considerably larger than that of LZ77. Evidently, to prepare LZ-based features for classification purposes, additional processing

must be performed to reduce the feature set dimensionality, such as feature hashing in the case of the LZJD and SHWel methods. However, this increases computational time complexity and for very large corpora or large file sizes, can render these approaches infeasible.

3. Efficient Huffman features

The goal of this research was to devise a method of representing malware in an abstract way, such that the characteristics or features of the representation could be used to accurately classify each sample into their respective family classes. Compression is an intuitive approach, since it provides a way of representing information in a reduced, compact form. Motivated by the limitations of current research, we chose the Huffman coding algorithm as it provides lossless compression, without being overly complex. Since the number of codewords in the Huffman set is equal to the number of alphabet of symbols in the input sequence, feature dimensionality is generally small. This is computationally advantageous as it requires less time and space to train a classifier and make inferences (Guyon and Elisseeff, 2003).

3.1. Huffman coding

The Huffman procedure is based on optimum prefix codes. In an optimum code, symbols that occur more frequently, i.e., have a higher probability of occurrence, will have shorter codewords than symbols that occur less frequently. To compress an input sequence, the Huffman coding algorithm constructs a tree structure as follows: The frequencies of the n symbols in the input sequence alphabet are used as the initial weights attached to a set of leaf nodes, one per alphabet symbol. The two lowest weighted nodes are identified and removed from the set, and combined to make a new internal node that is given a weight equal to the sum of the weights of the two nodes. That new node-weight pair is then added back to the set, and the process repeated. After $n - 1$ iterations of this cycle the set contains just one node that incorporates all of the original source symbols, and has an associated weight that is the sum of the original, at which point the process stops. The binary codewords usually use the convention that a left edge corresponds to a 0 bit and a right edge to a 1 bit and are found by constructing the binary number following from the root node to the leaf representing the symbol. Fig. 1 illustrates a Huffman tree for the string

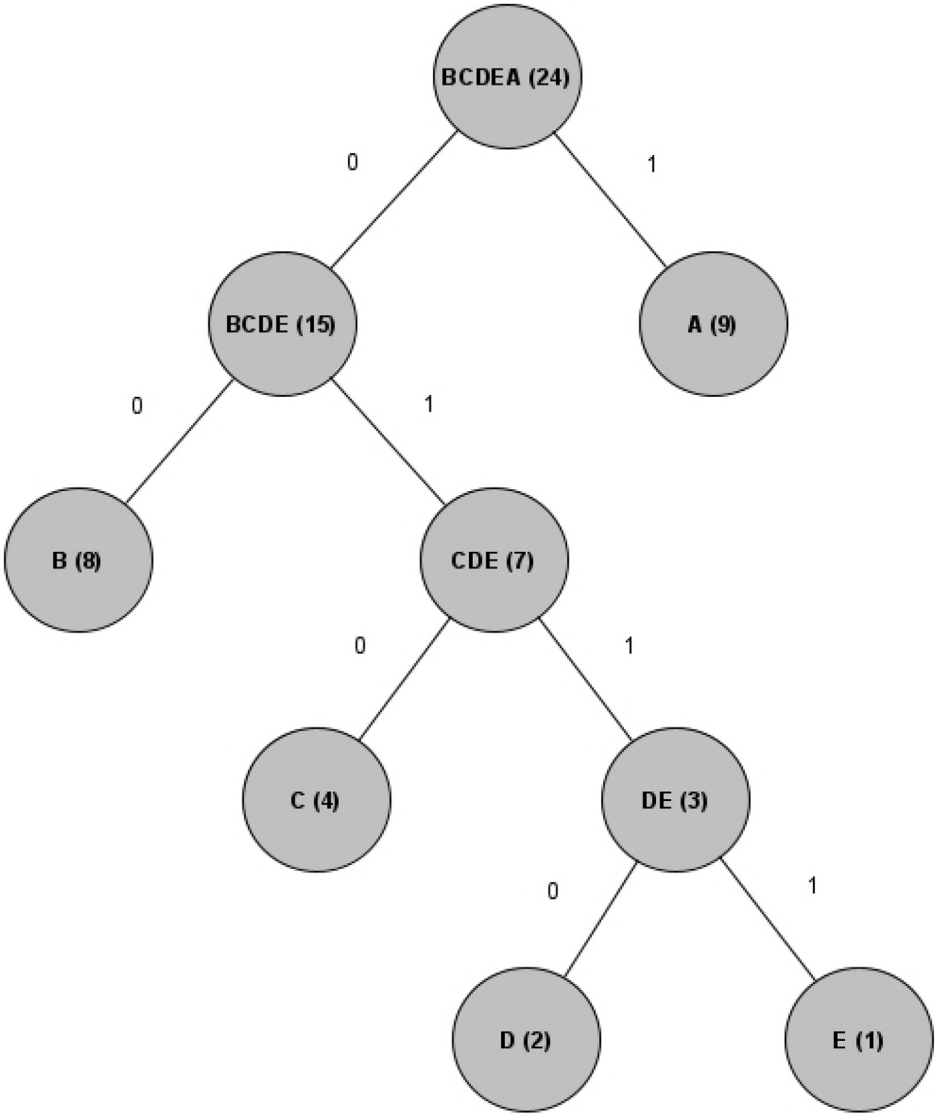

Fig. 1. Huffman tree for the string: AAAAAAAAABBBBBBBBCCCCDDE.

AAAAAAAAABBBBBBBBCCCCDDE. The frequencies of each node are shown in brackets. From the previous explanation, the order of the tree node construction is:

$$D_{(2)}E_{(1)}, C_{(4)}DE_{(3)}, B_{(8)}CDE_{(7)}, BCDE_{(15)}A_{(9)}$$

The Huffman codeword for each character is calculated by following the path from root to the character leaf, e.g. the codeword for D is: 0110. Table 1 shows the frequency and corresponding codeword for each symbol.

3.2. Huffman feature construction

Each feature in the eHf vector is constructed by combining the symbol, its frequency and Huffman codeword into a composite feature. The alphabet symbol and Huffman binary codeword are first converted to integers and the three values are added to produce a single feature. Malware variants by definition will share many common alphabet symbols, frequencies and codewords, so in turn will share many common composite features. Although we can consider the symbols, frequencies and codewords as separate dimensions in the feature vectors, we chose to combine them, which reduced feature set dimensionality by 66% and thus improved classification run-time complexity. It is of note that we initially calculated the features using the Huffman codewords only, but the models did not generalize as well on unseen data. By utilizing this composite feature approach, we were able to increase prediction performance by approximately 10%. This experimentation is not documented in the paper.

The eHf algorithm is constructed using a priority queue data structure, via a heap data structure. Priority queues are run-time efficient, with insertion and removal operations having a computational complexity of $O(n\ log(n))$. With priority queues, every item in the queue has a priority assigned to it and elements with a high priority are de-queued before an element with low priority. The alphabet symbol frequencies are assigned to determine the priority variables and in this way we can choose the two lowest weighted alphabet symbols (the first two items on the priority queue) to construct the nodes for the Huffman tree.

The pseudo code for our algorithm is shown in Algorithm 1. The sequence s denotes the data sequence to be encoded, in our case, a malware binary. The first step is to generate the frequency for each alphabet symbol in the binary. This is then stored in a dictionary f where each alphabet symbol is the key and its frequency is the associated value. A heap ,h, is then constructed containing a tuple of each symbol's frequency ν with a nested array containing the alphabet symbol κ, and an empty string c. The string c will store the Huffman codeword for each symbol.

To construct the Huffman codewords, the two lowest frequency weight pairs are de-queued from the heap, representing the left and right nodes in the tree. Zero is assigned to c in the left node representing the 0 binary bit for the left node edge and a one to c in the right node, representing the 1 binary bit for the right node edge. If a node has previously been processed, the node edge (1 or 0) is concatenated to c. The accumulation of the node edge values will eventually form the binary Huffman codeword. Next, the sum of the frequencies, which represents the parent node weight in the tree, along with the array containing the left and right nodes with their node edge values are pushed back onto the queue. The next two lowest frequency weight pairs are de-queued from the heap and the process is repeated until it reaches the root node. At this time, the heap consists of a single element, containing the sum of all the weights and the *symbol:codeword* pair array for each alphabet symbol.

The complete tree is next dequeued from the heap, with the sum of the weights removed as this is not considered in the calculation of the eHf features. The tree is then sorted in ascending order of codeword size. Finally, the code loops through the tree, summing symbol:codeword pairs and their corresponding symbol value ν, feature in the algorithm. On each pass, the feature is appended to the eHfset vector. Once the vector is constructed, it is returned.

Since the feature dimension of the eHf set is equivalent to the number of alphabet of symbols contained in the input data, it is possible to encounter cases where the resulting feature vectors differ in dimension. To produce a uniform length set of vectors suitable for classification, we first determine the vector with least dimensions, k. We then and resize all vectors in the feature set to k dimensions by choosing the first k features in each vector. Since each vector has been previously sorted by codeword value in ascending order, this ensures we are removing the largest codewords from the vectors, which represent the symbols with the lowest probability of occurrence and so inflict the least disturbance on the feature vectors.

Algorithm 1. Efficient Huffman Features

Input: sequence, s ;
Result: eHfSet, vector of Huffman features
Initialize: $f = \{\kappa : \nu\}$, $s \rightarrow s[\nu, [\kappa, c = \emptyset]]$, $ehfSet = \{\emptyset\}$, $h = \{\emptyset\}$
$h \leftarrow s$
while $length(h) > 1$ **do**
  $left = h.dequeue()$ `// left child node`
  $right = h.dequeue()$ `// right child node`
  for c *in* $left[1:]$ **do**
    $c[1] \leftarrow 0 + c[1]$ `// concat 0`
  end
  for c *in* $right[1:]$ **do**
    $c[1] \leftarrow 1 + c[1]$ `// concat 1`
  end
  $h.enqueue(\nu = (left[0] + right[0]),$
    $[left[1:], right[1:]])$
end
`// pop constructed tree`
$tree = sorted(h.dequeue())$
for t *in* $tree$ **do**
  $feature = t[0] + t[1] + f[t[1]]$
  $eHfSet \leftarrow eHfSet + feature$
end
return $eHfSet$

Table 1
Huffman codewords. This table corresponds to Fig. 1 and shows each symbol, their corresponding frequency and the resulting Huffman codeword.

Symbol	Freq.	Codeword
A	9	1
B	8	00
C	4	010
D	2	0110
E	1	0111

3.3. Feature optimization

Feature reduction can be advantageous when performing machine learning tasks. Datasets with less dimensions have better runtime efficiency and in some cases can improve classification performance due to a possible reduction in multicollinearity. We tested the effect of different values for k dimensions on run time

efficiency and classification performance, illustrated in Fig. 2. Since the number of dimensions is quite low, there was little change in the classification accuracy, however we were able to decrease classification runtime considerably. The graph shows an example where we reduced the number of feature dimensions in the VirusTotal subset from 229 (the smallest Huffman feature vector computed by our method for the VirusTotal dataset) to 150. By doing so, we decreased classification computation time from 104.18 s to 83.49 s, a reduction of approximately 20%, which is denoted by the shaded area under the top (compute time) curve in the graph. In contrast, the classification error rate in this case increased just 0.4%, from 0.018% to 0.022%, denoted by the shaded area under the bottom (error rate) curve. This demonstrates that eHf can be configured to accelerate runtime computations, while maintaining a high level of classification performance.

4. Methodology

The research presented in this paper provides a study on the efficacy of implementing Huffman coding to represent malware features that can be used to classify them according to their family class. The method of the study comprises four distinct parts: data gathering & pre-processing, Huffman feature generation, classification and model validation.

4.1. Data collection and pre-processing

For initial method testing, we used a subset of the VT data, which comprised 8232 portable executable (PE) samples from 12 distinct malware families. The sample sizes varied from 13.8 KB up to 61.7 MB, totalling 11.4 GB in size. The samples from this repository are not labelled, but have an associated JSON file which contains the detection outputs from the anti-malware scanners on VT, so there is some pre-processing necessary to assign the correct family label. In order to ensure correct labeling, we used a tool called AVClass Labeller to cluster malware samples into their family classes based on the output of the VT JSON reports (Sebastián et al., 2016). Only samples with a distinct label detection output from 10 or more scanners were chosen as true labelled samples. For comparative analysis with LZJD and NCD, we extended the VT dataset to 23 family classes, increasing the corpus to 14,694 samples, totalling 20.4 GB. The samples ranged from 8.2 KB up to 63.4 MB. Testing with this data also allowed us to evaluate eHf on a larger corpus to demonstrate its viability for malware family classification. The breakdown of the dataset is given in Table 2. Note: eHf's classification performance on the extended VT dataset is discussed in Sec. 5.3.

Table 2
Malware dataset comprising samples collected from the 2018–2021 VirusTotal academic share repositories.

Family	Type	Count
Agen	Virus	676
Allaple	Worm	339
Autoit	Worm	966
Berbew	Backdoor/Downloader	1692
Bitman	Trojan	1090
Dinwod	Trojan	893
Dorkbot	Info. stealer	92
Dridex	Banker Trojan	523
Emotet	Banker Trojan	497
Fsysna	Cryptominer	614
Hematite	File virus	562
InstallMonster	Adware Installer	364
Oberal	Spyware	224
Picsys	Worm	534
Salgorea	Backdoor	607
Scar	Trojan downloader	305
Sfone	Worm	1053
Shifu	Banker Trojan	475
Socks	Virus	495
Sytro	Worm	1140
Agent.BDMJ	Adware installer	97
Vilsel	Trojan	752
Vobfus	Worm	704
Total		**14,694**

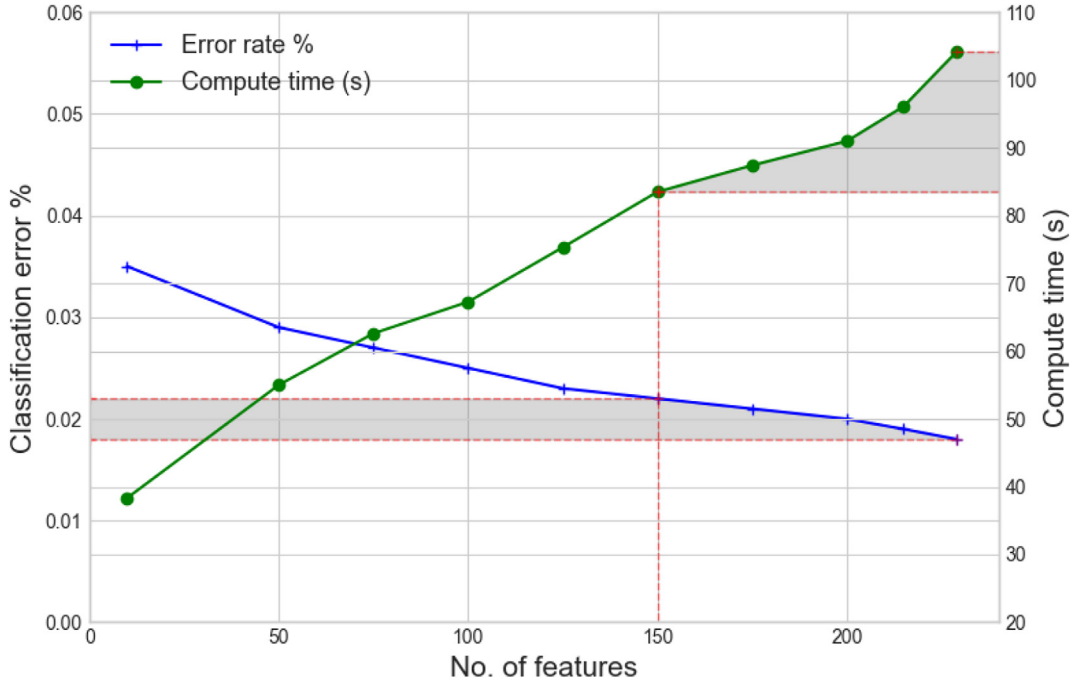

Fig. 2. Measuring the effect of varying features on classification error rate and computation time (based on VirusTotal dataset). By reducing the features from 229 to 150, compute time is reduced by 19.9%, with a classification error penalty of 0.4%.

4.2. Classification

To evaluate our methods, we chose the supervised machine learning algorithm KNN, since the class labels were already known, as described before. Supervised learning maps an inferred function between the training data and the corresponding class label and uses this function to identify class labels for unseen samples. For the KNN classifier, the most important input hyper-parameters are k, the neighbourhood size considered around each data point and the distance metric. Varying the size of k will directly influence the performance of the classifier, since it determines the label of a data point using a majority vote of the k most similar instances, or neighbours. KNN measures the neighborhood based on a distance metric such as Euclidean, Minkowski or Cityblock. Most significantly, KNN stores all available cases and classifies new cases based on a similarity measure, which allowed us to test our method directly against the NCD distance metric. The optimal combination of hyper-parameters was determined using *GridSearchCV*, which is part of the Python *Sci-kit Learn* library. GridSearchCV calculates all possible combinations of hyper-parameter input values for the classifier, and the best combination is retained. The optimum parameters for our KNN classifier was found to be $k = 3$, measured with the Minkowski distance metric.

4.3. Performance metrics

We performed extensive testing of our Huffman feature generation method to ensure robust classification models that could generalize well to previously unseen samples. We used the metrics precision, recall and accuracy to gauge each classifiers performance on the malware data. In the context of our research: Precision, in this context, tells us *'for all the malware labelled as a particular family, how many were correct?'*; recall tells us *'for each malware family, how many that should have been labelled as that family, were labelled correctly?'* and accuracy tells us *'how many samples were labelled correctly out of all the predictions?'*. A confusion matrix was used to provide a graphical view of how well each model performed.

4.4. Underfitting and overfitting

Throughout the machine learning phase, efforts were made to reduce the possibility of underfitting and overfitting in the classification models. Underfitting can occur when there is not enough data to build a generalized model and the model will fail to identify important signatures or patterns in the data. We compiled a suitably large dataset to reduce the risk of underfitting. Overfitting occurs when the model fits the data too well, mainly due to it capturing noise along with the underlying signatures or patterns in the data. A model that is trained to fit slightly inaccurate data can infect it with substantial errors and reduce its predictive power. In an effort to minimize overfitting, we implemented stratified 5-fold cross-validation. Stratification is the process of dividing members of the data population into homogeneous subgroups before sampling. In stratified cross-validation, folds are stratified such that they contain approximately the same proportion of samples as the original dataset. Cross-validation helps identify if overfitting is present in the data by repeating the training and testing phases multiple times, using all the different folds of the training set as validation sets. During our parameter tuning phase, we also tested 10-fold cross-validation, but our results showed 5-fold produced superior results.

5. Experimental analysis and results

Our initial model training was carried out on a subset of the VT dataset, previously described in Sec. 4.1. Approximately 10% of the data for each family class was held out for the validation phase where we evaluated the robustness of the classification model against previously unseen data. The model was trained using the following procedure:

1. Compute the Huffman feature vector for each sample.
2. Assign the family label to each vector.
3. Pass the feature set (vectors and labels) to the KNN classifier.
4. Perform stratified 5-fold cross-validation.
5. Compute precision, recall and accuracy.
6. Compute the confusion matrix.

The KNN classification performance metrics are presented in Table 3. The KNN model produced excellent results in the training phase, with the weighted average precision, recall and accuracy scores over all classes of 98.2%. The weighted average calculates the metrics for each class label and finds their average weighting by the support or the number of true instances for each label. This method accounts for any label imbalance as in the case of the VT dataset, where class sizes ranged from 92 to 1692. The highest metrics were returned for the banker Trojan Dridex precision, recall and accuracy of 100%. The lowest performing family was the Trojan downloader Scar, with metrics of 85.7%, 85.4% and 85.5% for precision, recall and accuracy, respectively.

The confusion matrix in Fig. 3 illustrates the classification metrics reported in Table 3. The actual class values are represented in the rows, while the columns represent the predictions. The intersection of the rows and columns represent the true positives, i.e., where the model predicted actual values correctly. It can be seen from the confusion matrix that the model predicted most classes correctly, with 11 out of 12 class prediction true positive rates (TPR) of 97% or above. As per the findings in Table 3, the lowest performing family, Scar, returned a TPR of 85%. On closer inspection of Scar, the classifier returned 60 false positives, where samples from other families were predicted incorrectly as Scar. In particular, 18 samples from the Berbew family were predicted as Scar. Scar and Berbew both possess downloader functionality, which may have caused the misclassification. Additionally, 14 false positives were returned for the Vilsel family. It was found that, in several Vilsel samples uploaded to VirusTotal, the AV scanners detected Scar as Vilsel, so this may have attributed to the incorrect

Table 3
Classification performance for the KNN model on the VT subset. The best metrics were returned for the banker Trojan Dridex, highlighted in bold.

Family	Precision	Recall	Accuracy
Agent.BDMJ	0.989	1.000	0.994
Autoit	0.961	0.976	0.969
Berbew	0.993	0.986	0.990
Dinwod	0.994	0.983	0.988
Dorkbot	0.977	0.988	0.982
Dridex	**1.000**	**1.000**	**1.000**
Oberal	0.976	1.000	0.988
Scar	0.857	0.854	0.855
Sfone	0.987	0.996	0.991
Socks	0.991	0.980	0.986
Sytro	0.994	0.999	0.997
Vilsel	0.985	0.971	0.978
Weighted avg.	**0.982**	**0.982**	**0.982**

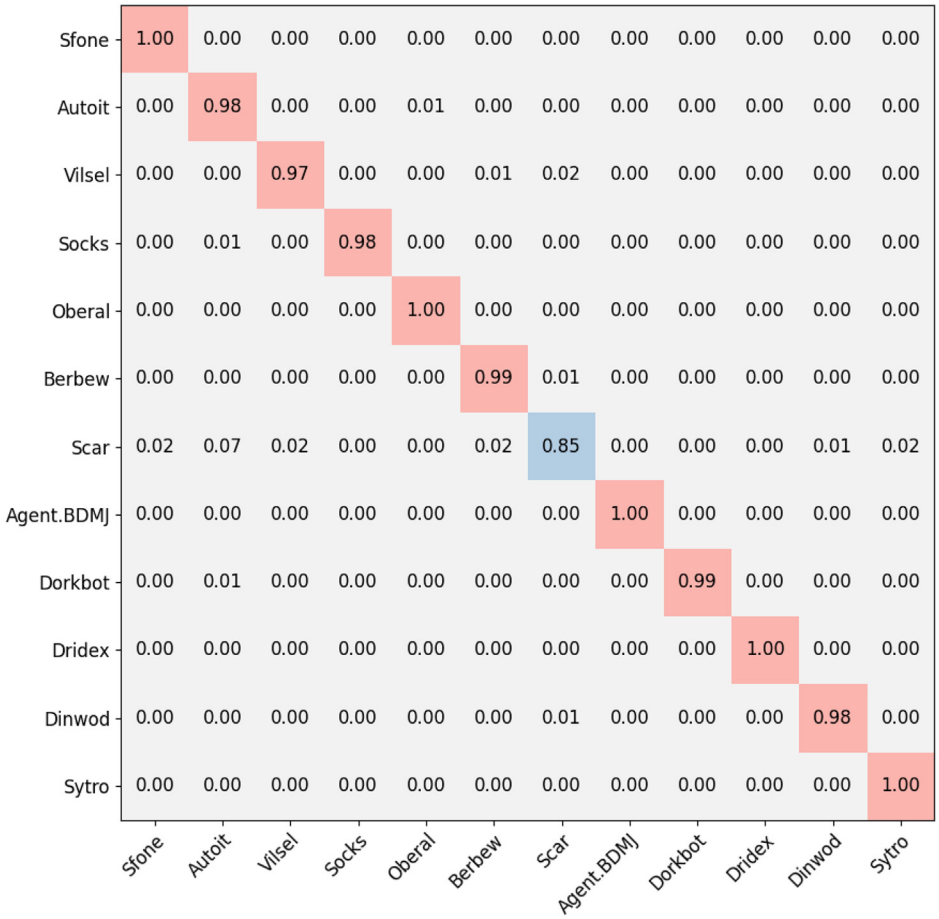

Fig. 3. Confusion matrix for the KNN model trained on the VT subset.

predictions (VirusTotal, 2021b). The false negatives for Scar, where actual samples of Scar were classified as other families, were 39. The highest misclassification rate was for Autoit, in 7.2% of cases or 22 out of 305 samples. Similarly to Vilsel, we identified that some AV scanners detect Scar as Autoit (VirusTotal, 2021a). Furthermore, we noted some variants of Scar, e.g., Win32.Scar.A, possess worm propagation functionality, which could have contributed to the confusion with the Autoit worm.

5.1. Model validation

A classification model's ability to generalize to new data is central to its success. A correctly trained model can process new data samples and make accurate predictions. The eHf KNN classification model was evaluated on the 10% holdout set that was removed from the VT dataset before the training phase. Stratified random sampling was used to select the evaluation set in order to provide an accurate representation of the original sample data population. The model predicted a class average of 97.4% for precision, recall and accuracy respectively. The results show that the eHf KNN model is robust in that it generalized to the new data samples. This indicates the model did not overfit the training data and adapted to predict similar data samples, in this case variants from the same malware family, providing an effective solution for malware family classification.

5.2. Comparison with NCD

As NCD has been featured heavily in previous compression-based approaches, it necessitated a comparative study with our eHf method. While eHf is not a similarity distance metric, our aim here was to determine if standard distance measures could be used with eHf in place of NCD for compression-based similarity, given its runtime complexity limitations. For comparisons, we chose Jaccard, Minkowski and Euclidean distance measures to test against NCD. All four distance metrics were tested on the extended VT dataset. The results are shown in Table 4. In terms of runtime, Minkowski and Euclidean yielded similar run times of approximately 1 ms per distance calculation, with Jaccard slightly slower at 1.4 ms. NCD was over 3 orders of magnitude slower at 1.2 s per calculation. NCD also performed considerably worse as a distance measurement, approximately 20% lower for each classification performance

Table 4
Distance metric comparisons showing the average runtime for calculating the distance measurement between 2 data samples and the average classification metrics, based on the extended VT data. Best results were returned for eHf using the Minkowski distance, highlighted in bold.

Dist. metric	Runtime (secs)	Prec.	Recall	Acc.
eHf-Jaccard	1.42×10^{-3}	0.969	0.968	0.968
eHf-Minkowski	$\mathbf{1.02 \times 10^{-3}}$	**0.972**	**0.973**	**0.972**
eHf-Euclidean	1.06×10^{-3}	0.970	0.971	0.969
NCD	1.2	0.782	0.774	0.772

metric. Our findings reiterate previous work concluding that NCD is not a feasible distance metric for malware classification at scale.

5.3. Comparison with LZJD

In this section, we evaluate our eHf technique against the LZJD algorithm, since it is the closest approach to our own in terms of compression-based feature generation. In the interests of equity, we used PyLZJD, the Python version of LZJD (Raff, 2018), considering our code was also written in Python. Within LZJD, there are two modes for hash set creation, the default hash and *'SuperMinHash'* which, according to its developers, is slower than the default implementation, but converts feature sets to a more compact vectorized representation for classification (Edward Raff et al., 2019). We could not locate code for the faster SHWel algorithm, but it is noted in LZJD's source code that the algorithm contains *'SHWel-style hashing'*. We tested eHf against both LZJD implementations, to demonstrate a more comprehensive comparison of the two approaches. For clarity, we refer to the superhash implementation of LZJD as LZJD-sh. For our comparative analyses, we used the extended VT dataset with 14,694 samples. As in our initial model training, we held out a 10% subset of the data for model validation testing.

We first tested the methods on run-time complexity in terms of feature generation and classification times. The results of the comparison are shown in Table 5.

From the results, eHf was 2.42 times faster than LZJD in generating feature vectors and just over one order of magnitude faster than LZJD-sh. In classification training, eHf was 6.85 times faster than LZJD and 3.66 times faster than LZJD-sh. These results are further discussed in Section 6.

Next, we compared eHf to the LZJD methods on classification performance. We used the KNN classifier and 5-fold cross-validation as in our initial model training on the VT subset data. The results are shown in Table 6. In the model training phase, the LZJD-sh method produced marginally better results than eHf, with precision, recall and accuracy of 97.7%, 97.3% and 97.4% as opposed to 97.2%, 97.3% and 97.2% for eHf. The default LZJD results were approximately 2% lower. In the validation phase, eHf proved to be significantly superior to both LZJD methods, returning classification metrics that were approximately 10% better than LZJD-sh and approximately 14% better than the default LZJD. The results indicate that eHf KNN provided a more robust classification model than either LZJD method in adapting to previously unseen samples. However, we note here that the LZJD methods were used as a comparison to gauge the validity of eHf for feature representation and in this regard, we did not perform extensive feature selection or parameter tuning, though we did test the LZJD methods on different values for k in the KNN classifier.

Table 5
Comparison of runtime efficiency for eHf and LZJD, where *feat. gen.* = *feature generation, training* = *classification training and feat. dims.* = *dimension of each feature vector*. The times shown are averages per sample in milliseconds.

	Run-time efficiency (ms)		Size
	feat. gen.	training	feat. dims.
LZJD-sh	5.15×10^{-1}	9.92×10^{-2}	1024
LZJD	1.21×10^{-1}	1.86×10^{-1}	1024
eHf	5.0×10^{-2}	2.71×10^{-2}	**229**

Table 6
Comparison of classification performance for eHf and LZJD methods. The scores represent precision, recall and accuracy averaged over all classes for the training and validation phases. The best performing scores are highlighted in bold.

	Training			Validation		
	precision	recall	accuracy	precision	recall	accuracy
LZJD-sh	**0.977**	**0.973**	**0.974**	0.882	0.878	0.873
LZJD	0.951	0.950	0.950	0.745	0.752	0.746
eHf	0.972	0.973	0.972	**0.974**	**0.974**	**0.974**

5.4. Resilience to obfuscation (reordering)

It is of vital importance that malware detection and classification schemes are resilient against obfuscation employed by malware developers. In this section, we demonstrate how eHf is robust against code reordering obfuscation. For simplicity, we chose to use the import tables extracted from each malware sample to test our approach. Import tables are structures contained within executable files comprising a list of functions that programs import in order to communicate with the operating system. Malware programs that contain similar code and have been compiled in the same way will generally have the same import table and in this way can be used as signatures to identify related malware samples. For example, Mandiant (2016) developed imphash, which comprises an MD5 hash generated from the extracted malware binary. Tests have shown that family variants generally share the exact same function code and therefore can be clustered together on the basis of their imphash. Furthermore, different strains of malware have been shown to be attributed to a single threat group, through identical imphashes. imphash suffers the same limitations of cryptographic hashing in that any reordering of the code will result in a completely different hash. Our test here demonstrates eHf is resilient against such tactics.

To perform our evaluation, we first extracted the import tables from each sample in the VT dataset. We then trained our KNN classifier on the import tables in the same way as detailed previously in this section. Next, we used SCYTHE, an obfuscation tool presented by Balles and Sharfuddin (2019), to perform a random reordering routine on the import tables of a sub-sample of approximately 3 k malware binaries. This produced a different (reordered) import table for each sample. We then evaluated the KNN model against the reordered import tables. The average precision, recall and accuracy scores returned were 99.8%, 99.7% and 99.7%, respectively. The results show that, despite the random reordering, eHf is not adversely affected. The reason for this lies in the way in which the Huffman encoding generates its compression dictionary. As described in Huffman coding, input data is stored in a dictionary structure according to their frequency of occurrence. Therefore, the ordering of the input sequence is not a consideration in the generation of the codewords. As long as the input data, in our case the malware binary code, contains the same or similar "alphabet" of symbols and frequencies, then reordering has little effect.

6. Discussion

The eHf method described in this paper provides a representation of data in a compressed feature space that does not require invasive feature extraction processes and does not suffer the complexities and scalability issues of such techniques. Furthermore, eHf operates without the need for domain-specific knowledge, which means it can be applied to a multitude of domains apart from malware classification. Prior related work has focused predominantly on the normalized compression distance (NCD), a compression-based distance metric for measuring similarity. Although NCD has shown promise in terms of classification

performance, it has proven to exhibit poor runtime efficiency, due to the calculation requirement for compression of both the data inputs individually and concatenated, which has been demonstrated to be infeasible at scale. In this regard, we have shown that standard distance metrics such as Minkowski and Euclidean produce far superior results over NCD. The results of our testing show that eHf is a viable malware classification scheme, returning average precision, recall and accuracy scores of approximately 97% using a KNN classifier and Minkowski distance metric trained on a corpus of approximately 15 k malware samples. These results mean that eHf has the ability to accurately predict malware variants from the same family it has been trained on.

By comparing eHf to LZJD, a similar compression-based approach using the Lempel-Ziv compression algorithm and the Jaccard index, we demonstrated that our method performed comparatively well in classification model training, but proved to produce more robust models, with improved prediction capabilities of 10–14% over both LZJD methods. In terms of runtime efficiency, eHF exhibited greatly and reduced computational complexity over LZJD. This is attributed to several factors. First, the Huffman encoding algorithm is less complex than the LZ77 algorithm used by LZJD and so affords a much faster feature computation time. Second, by implementing our code using a min-heap data structure, we could optimize data processing times to a complexity of $O(n\ log(n))$. Third, with eHf we can represent the data within a considerably reduced feature space (229 vs. 1024 dimensions for LZJD) which reduces classification time. It should be noted that the feature set size mode in LZJD can be configured to smaller dimensions, but we did not test how this would influence its performance.

Through our experiments, we observed that we could further reduce the eHf feature set dimensionality, which resulted in a considerable decrease in computation time, with a small penalty in increased classification error. This property would be beneficial in time critical scenarios or classification of large corpora and where a slight increase in error rate is acceptable. Our evaluation also shows that eHf is resilient against code reordering obfuscation. This is due to the Huffman encoding procedure, where each codeword is based on the frequencies of the input data and so the order of the data has no influence on the features generated. This property infers that eHf could potentially be applied to digital forensics situations, for example, where modified data needs to be examined for similarity.

7. Conclusions and future work

Due to the limited related work in this domain, our intention for this research was to investigate the efficacy of compression as a solution to represent data in such a way that it abstracts away from the need to understand intricate structural or analysis procedures. In doing so, we introduced a novel method of feature set generation using a modified implementation of the Huffman encoding algorithm, which we dubbed efficient Huffman features or eHf. We note here that, while we have not tested eHf on other data types, such as in text file classification, our method is not limited to a specific domain, so it can potentially be applied to a variety of fields other than malware classification. Although we limited our research to KNN classification, the feature sets generated by eHf are constructed as an array structure. Thus, they are in a suitable format for testing in a variety of other machine learning algorithms. Our evaluation of eHf with other compression-based approaches demonstrated a greatly improved runtime efficiency over NCD and LZJD. We have also shown that eHF produced more robust classification models than LZJD, in predicting previously unseen sample data. Further testing of eHF has shown it is resilient to code reordering obfuscation.

Overall, the research presented here has shown that eHf exhibits the desirable properties of code reordering obfuscation resilience, non-invasive data feature extraction techniques and fast computation times, without the need for domain-specific knowledge, making it a simple, practicable and scalable solution suited to the classification of large malware corpora.

For future work, we aim to investigate the robustness of eHf against other common forms of obfuscation such as packing and encryption. Furthermore, we intend to extend our testing to evaluate eHf at scale. Our intention is to use the SOREL–20 M dataset, which comprises 10 million disarmed malware samples made available for research purposes (Harang and Rudd, 2020). To make eHf more feasible at this scale, we also intend to make further improvements on the performance of the source code, for example, utilizing Cython for faster computation of the feature sets (Behnel et al., 2011).

References

Alshahwan, N., Barr, E.T., Clark, D., Danezis, G., 2015. Detecting malware with information complexity. CoRR, abs/1502.07661. arXiv:1502.07661. http://arxiv.org/abs/1502.07661.

Apel, M., Bockermann, C., Meier, M., 2009. Measuring similarity of malware behavior. In: 2009 IEEE 34th Conference on Local Computer Networks, pp. 891–898. https://doi.org/10.1109/LCN.2009.5355037.

Arp, D., Spreitzenbarth, M., Gascon, H., Rieck, K., 2014. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket.

Balles, C., Sharfuddin, A., 2019. In: Breaking Imphash. CoRR, abs/1909.07630 arXiv:1909.07630. http://arxiv.org/abs/1909.07630.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S., Smith, K., 2011. Cython: the best of both worlds. Comput. Sci. Eng. 13, 31–39.

Breitinger, F., Ziroff, G., Lange, S., Baier, H., 2014. Similarity hashing based on levenshtein distances. In: IFIP International Conference on Digital Forensics. Springer, pp. 133–147.

Choi, Y., Kim, I., Oh, J., Ryou, J., 2008. Pe file header analysis-based packed pe file detection technique (phad). In: International Symposium on Computer Science and its Applications, pp. 28–31. https://doi.org/10.1109/CSA.2008.28.

Cilibrasi, R., Vitanyi, P.M.B., 2005. Clustering by compression. IEEE Trans. Inf. Theor. 51, 1523–1545. https://doi.org/10.1109/TIT.2005.844059.

Crawford, C., 2016. 20 newsgroups dataset. https://www.kaggle.com/crawford/20-newsgroups.

Raff, Edward, Joe, Aurelio, Nicholas, Charles, 2019. PyLZJD: an easy to use tool for machine learning. In: Lippa, Dillon Niederhut, Shupe, David (Eds.), Chris Calloway, David, Proceedings of the 18th Python in Science Conference, pp. 101–106. https://doi.org/10.25080/Majora-7ddc1dd1-00e.

Guyon, I., Elisseeff, A., 2003. An introduction to variable and feature selection. J. Mach. Learn. Res. 3, 1157–1182.

Hansen, S.S., Larsen, T.M.T., Stevanovic, M., Pedersen, J.M., 2016. An approach for detection and family classification of malware based on behavioral analysis. In: 2016 International Conference on Computing, Networking and Communications (ICNC), pp. 1–5. https://doi.org/10.1109/ICCNC.2016.7440587.

Harang, R., Rudd, E.M., 2020. Sorel-20m: A Large Scale Benchmark Dataset for Malicious Pe Detection arXiv:2012.07634.

Huffman, D.A., 1952. A method for the construction of minimum-redundancy codes. Proceedings of the IRE 40, 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898.

Kolmogorov, A., 1998. On tables of random numbers. Theor. Comput. Sci. 207, 387–395. https://doi.org/10.1016/S0304-3975(98)00075-9. http://www.sciencedirect.com/science/article/pii/S0304397598000759.

Li, M., Chen, X., Li, X., Ma, B., Vitanyi, P.M.B., 2004. The similarity metric. IEEE Trans. Inf. Theor. 50, 3250–3264. https://doi.org/10.1109/TIT.2004.838101.

Maas, A., Daly, R.E., Pham, P.T., Huang, D., Ng, A.Y., Potts, C., 2011. Learning word vectors for sentiment analysis. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, pp. 142–150.

Mandiant, 2016. Tracking malware with import hashing. https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html.

Marton, Y., Wu, N., Hellerstein, L., 2005. On compression-based text classification. In: Losada, D.E., Fernández-Luna, J.M. (Eds.), Advances in Information Retrieval. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 300–314.

Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S., 2011. Malware images: visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security VizSec '11. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2016904.2016908.

Paskov, H.S., West, R., Mitchell, J.C., Hastie, T., 2013. Compressive feature learning. In: Burges, C.J.C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K.Q. (Eds.), Advances in Neural Information Processing Systems, vol. 26. Curran Associates, Inc., pp. 2931–2939. In: https://proceedings.neurips.cc/paper/2013/file/

5e1b18c4c6a6d31695acbae3fd70ecc6-Paper.pdf
Raff, E., 2018. Pylzjd. https://github.com/EdwardRaff/pyLZJD.
Raff, E., Nicholas, C., 2017a. An alternative to ncd for large sequences, lempel-ziv jaccard distance. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining KDD '17. ACM, New York, NY, USA, pp. 1007–1015. https://doi.org/10.1145/3097983.3098111.
Raff, E., Nicholas, C., 2017b. Malware classification and class imbalance via stochastic hashed lzjd. In: Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security AISec '17. Association for Computing Machinery, New York, NY, USA, pp. 111–120. https://doi.org/10.1145/3128572.3140446.
Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M., 2018. Microsoft malware classification challenge. CoRR, abs/1802.10135. arXiv:1802.10135. http://arxiv.org/abs/1802.10135.
Sanchez, J., Perronnin, F., 2011. High-dimensional signature compression for large-scale image classification. In: CVPR 2011, pp. 1665–1672. https://doi.org/10.1109/CVPR.2011.5995504.
Saxe, J., Mentis, D., Greamo, C., 2012. Visualization of shared system call sequence relationships in large malware corpora. In: Proceedings of the Ninth International Symposium on Visualization for Cyber Security VizSec '12. Association for Computing Machinery, New York, NY, USA, pp. 33–40. https://doi.org/10.1145/2379690.2379695.
Schuller Borbely, R., 2016. On normalized compression distance and large malware. Journal of Computer Virology and Hacking Techniques 12, 235–242. https://doi.org/10.1145/1671970.1921702. https://doi.org/10.1007/s11416-015-0260-0.
Sculley, D., Brodley, C.E., 2006. Compression and machine learning: a new perspective on feature space vectors. In: Data Compression Conference (DCC'06), pp. 332–341. https://doi.org/10.1109/DCC.2006.13.
Sebastián, M., Rivera, R., Kotzias, P., Caballero, J., 2016. Avclass: a tool for massive malware labeling. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (Eds.), Research in Attacks, Intrusions, and Defenses. Springer International Publishing, Cham, pp. 230–253.
Vinitha, p., Gopalakrishnan, G., Muralikrishnan, K., 2016. An optimal seed based compression algorithm for dna sequences. Advances in Bioinformatics 3, 1–7. https://doi.org/10.1155/2016/3528406.
VirusTotal, 2021a. Virustotal scan of the win32.scar trojan downloader. https://www.virustotal.com/gui/file/d915b5633b6a39f8943d3ab1cc960229737d741a6c0f1fedde5b1e3028a8ceef/detection.
VirusTotal, 2021b. Virustotal scan of vilsel. https://www.virustotal.com/gui/file/20fdcd9f6a53011a1065014bd4649f58d4143cb2debb70a4a279acb920c276fc/detection/.
Walenstein, A., Hayes, M., Lakhotia, A., 2017. Phylogenetic comparisons of malware. In: Virus Bulletin Conference, vol. 39, p. 41. https://www.virusbulletin.com/uploads/pdf/conference_slides/2007/WalensteinVB2007.pdf.
Wehner, S., 2005. Analyzing worms and network traffic using compression. CoRR, abs/cs/0504045. arXiv:cs/0504045. http://arxiv.org/abs/cs/0504045.