Toward the Detection of Polyglot Files

Luke Koch
Bredesen Center
kochlr@ornl.gov

Mary Adkisson
Tennessee Tech University
adkissonma@ornl.gov

Brian Weber
Oak Ridge National Laboratory
weberb@ornl.gov

Sean Oesch
Oak Ridge National Laboratory
oeschts@ornl.gov

Sam Erwin
Pacific Northwest National Laboratory
Samantha.erwin@pnnl.gov

Amul Chaulagain
Oak Ridge National Laboratory
chaulagaina@ornl.gov

***Abstract*—Standardized file types play a key role in the development and use of computer software. However, it is possible to abuse standardized file types by creating a file that is valid in multiple file types. The resulting polyglot (*many languages*) file can confound file type identification, allowing elements of the file to evade analysis. This is especially problematic for malware detection systems that rely on file type identification for feature extraction. Although work has been done to identify file types using more comprehensive methods than file signatures, accurate identification of polyglot files remains an open problem. Since malware detection systems routinely perform file type-specific feature extraction, polyglot files need to be filtered out prior to ingestion by these systems. Otherwise, malicious content could pass through undetected. To address the problem of polyglot detection we assembled a data set using the `mitra` tool. We then evaluated the performance of the most commonly used file identification tool, `file`. Finally, we demonstrated the accuracy, precision, recall and F1 score of a range of machine and deep learning models. Malconv2 and Catboost demonstrated the highest recall on our data set with 95.16% and 95.45%, respectively. These models can be incorporated into a malware detector's file processing pipeline to filter out potentially malicious polyglots before file type-dependent feature extraction takes place.**

I. Introduction

A. Overview

A polyglot is a form of steganographic file that can be successfully interpreted in two or more types [1]–[3]. In other words, a JPG+JAR polyglot is one that presents an image when interpreted by an image viewer and executes self-extracting Java code when fed to the Java Runtime Environment. Many combinations are possible, including a polyglot that can be interpreted as an image, a video, a PDF, and a video game depending on which program interprets the polyglot [1]. As we demonstrate in our results (section IV), common utilities for identifying files do not reliably identify polyglots. Ergo, malware detection systems that rely on common utilities for file type identification prior to feature extraction are vulnerable to polyglot files. If a detector recognizes only the JPG portion of a JPG+JAR polyglot, then two problems can arise.

- For ML/DL-based detecters, feature extraction will only extract fields from the JPG portion of the file
- For signature-based detectors, only malware signatures associated with JPG files will be matched [4]

In either instance, malicious code in the JAR portion passes through without accurate analysis. In 2019, researchers at Oak Ridge National Laboratory established that multiple commercial off-the-shelf (COTS) malware detection systems failed to detect 100% of polyglot malware in the data set [5]. These polyglots were created by Assured Information Security as part of their red team campaign. Motivated by this failure, we compiled a data set of normal and polyglot files, trained multiple classifiers on this data set, and evaluated the recall and F1 score of these classifiers. Our objective is to protect malware detection systems that rely on file type identification for feature extraction or signature sub-selection by developing a model to filter polyglots out of the file processing pipeline. This will allow COTS tools to filter out potentially malicious polyglots while still benefiting from the rich information provided by file type-specific feature extraction [6].

B. Motivation

Malware detection systems commonly rely on file type-specific features in order to classify an unknown file as either malicious or benign [6]. For example, the header of a PE file contains a great deal of information about the file, e.g., the read/write/execute flags of all sections, the offset to the import function table, and the address of the entry point. In a PDf file the XREF table, if present, is a useful source of information for determining which portions of the PDF could contain malicious activity. If the XREF is not present or consistent with the file contents, then malicious code can be found by scanning for *action* objects that launch Javascript code [7], [8]. In any event, knowing the file type allows the detector to extract far more information than otherwise possible. If the file type is not known, then only generic features like byte

entropy, n-grams, and strings can be used for classification [6].

In some instances, malware detectors will ignore files if the file type has no known malicious capability. If a polyglot file is identified as having only one file type, only that type will be used for feature extraction and further analysis. File sections from the undiscovered type will simply be ignored, allowing potential malicious activity to go undetected or misunderstood [4]. One solution is to forego file type-specific feature extraction. However, this greatly reduces the number of available manual features [6], [9]. Deep learning could be used to perform automatic feature extraction that is effective across all desired file types. However, this would require the complete overhaul of a file type-specific malware detection system. Instead, we propose simply adding a polyglot filter as a pre-processing step for existing malware detection systems. Provided the polyglot filter has sufficiently high recall, this solution would require minimal alteration to existing solutions. Furthermore, existing solutions would continue to benefit from the rich features that can be extracted once a file is known to contain only one file type.

Aside from the afore-mentioned trial at ORNL, additional evidence of the pressing need for polyglot detection is provided by a 2019 attack on DICOM files [10], [11]. DICOM files are commonly used medical imaging files. The attack involves the creation of a PE+DICOM polyglot that functions as expected when loaded into medical imaging software, yet is also capable of execution as a Windows PE file. The DICOM type is intentionally flexible; the authors anticipated the creation of TIFF+DICOM polyglots for non-malicious purposes [12]. However, this flexibility lends itself well to more malicious pursuits when the second file contains malicious code.

Due to the wide variety of file types that exist, we felt that a rule-based approach to polyglot detection would require excessive revision and, at best, only account for previously-encountered forms of polyglots. There are also a large number of discrepancies in terms of malformed files that make the rule-based approach difficult [4].

In an attempt to create a solution that learns the general problem and has the possibility of protecting against novel polyglots, we decided to explore machine and deep learning classifiers. This required the creation of a sizable training set that includes a wide variety of existing polyglots. Before we discuss this data set, we first provide a detailed look at the various types of polyglots that are included within it.

II. Background

A. Polyglot Mechanics

Although there has been some academic research into file type identification methods (discussed in the related works section II-B), polyglot creation has largely been driven by researchers from industry [1], [2]. Ange Albertini demonstrated multiple methods for creating polyglots at industry presentations and released the *python*-native *mitra* tool on Github to enable fellow researchers. We utilized this tool to create our polyglot dataset. Albertini classified polyglots based on the method used to combine donor files into the polyglot. Ergo, *mitra* attempts to create stacks, zippers, parasites, and cavities from pairs of donor files.

1) Stacks: A stack is the simplest type of polyglot. The second file is simply appended to the end of the first file. The caveat is that any byte offsets in the second file may need to be adjusted (increased by the length of the first file) in order for the second file to function as expected. Although there is no restriction on the first file, the second file must not strictly enforce the *magic number at offset zero* rule. A **magic number** is an arbitrary (hence the *magic* moniker) hexadecimal value that uniquely identifies a file type. This number is used for fast file type identification, since a utility need only scan the first few bytes of the file in order to identify it. However, there are many file types that do not enforce this rule.

PDF readers commonly accept files as valid PDFs if the magic number is anywhere within the first 1024 bytes of the file [1], [13] despite the requirement listed in the official documentation [14]. Additionally, some types have no requirement at all for a magic number at offset zero. Zip files commonly begin with a magic number. However, this number is part of the local file header for the first file contained within the Zip archive. According to the specifications [15], the Zip file is indexed via a central directory at the end of the file. This central directory has its own magic number for identification. Ergo, Zip files have no requirement that any magic number be the first byte. This makes them an ideal candidate for the second file in a stack polyglot.

2) Parasites: In a parasite polyglot, the second file is added within comment sections—offset by comment markers—of the first file. Many file types allow for comment sections that are not displayed when the file is interpreted. These comments are only visible when the file is opened for editing by a hex editor like *vim*. In order for the second file within the comment sections of the first file to remain functional, it must not have the strict *magic number at offset zero* rule. The byte offsets for both files must be updated. In the case of the first file, the byte offsets must now account for the second file contents that are now hidden within the comment sections. In the case of the second file, the offsets must account for the fact that the second file contents are now scattered in comment sections of the first file. As long as these offsets are updated correctly, both files will continue to function normally. Although this update process is more complex than the updates necessary for a stack polyglot, there are many file types than can combine to create a parasite.

3) Zippers: Zippers are a more complex version of a parasite. In a zipper, both files are contained within each other's comment sections. This means that the two donor files must use different markers to begin and end their comment sections. This arrangement is rather unusual in practice, so donor files for zippers are much harder to find. In our data set, only DCM files combined with either GIF or PDF files were able to create zipper polyglots.

4) Cavities: Cavities are created when the second file is hidden within null-padded areas of the first file. This arrangement is only possible when the first file is of an executable or ISO type, wherein memory is allocated in chunks. Since these areas are often null-padded to a standard size, executable or ISO files may contain enough null-padded memory to hide a second file in the padded areas. This is very similar to the classic *code caving* technique often used by malware authors to hide malicious code. The distinction between a cavity polyglot and code caving is that the caved material of a cavity polyglot is a complete file that can function correctly when interpreted by an appropriate program. Since the first 16 sectors of an ISO file are left empty, the contents of another file can be placed in the beginning of the ISO file. For executable files, like Windows PE files, the second file would be written into the null-padded trailing areas at the end of sections. No updates will be needed for the first file, but any byte offsets in the second file will need to be updated if the second file does not begin at offset zero of the first file.

B. Related Work

File type identification has historically been addressed through signatures. The Linux utility `file` matches files by examining magic bytes in unison with other structural elements. The magic number is an arbitrary hexadecimal value that uniquely identifies a file type. Some file types can be strictly identified simply by their magic number, while others require additional elements to be scanned. JAR files, for instance, are simply Zip files with a MANIFEST.INF file present in the archive. Ergo, `file` scans for the presence of this file in order to distinguish a JAR file from a Zip file. In either event, `file` matches each input file to a unique signature. Note, `file` scans until it detects a match, then halts. There is a `--keep going` flag, but in our examination it did not allow `file` to detect polyglots. The results of running `file` (with this flag active) on our polyglot data set is included in the results section. The signatures that `file` utilizes are extremely accurate and efficient in identifying conventional (monoglot) files, which means little research has been done to improve on this winning formula.

McDaniels and Heydari used byte histograms in concert with three different algorithms to identify file types [16]. Under the first algorithm (*byte frequency analysis* or BFA), they converted files into a fixed length feature vector. All files, regardless of type, can be represented as a sequence of hexadecimal values. Therefore, we can compactly summarize a file's contents by placing the number of times each possible hex value occurs into a 256 (all possible hexadecimal values) character vector where the index corresponds to the byte value. Ergo, the 7th value stored in the vector is the number of times the value 0x7 occurred in the file. This feature is referred to as a byte occurrence vector, unigram, or byte histogram in literature. McDaniels and Heydari then normalized each byte occurrence vector and calculated an average vector for each file type. Each average vector represented a unique file type and was referred to as a *fileprint*. Test files were compared to the *fileprint* to calculate correlation scores. Finally, each test file received its file type label from the most correlated *fileprint*.

For the *byte frequency cross-correlation* or BFC algorithm, they calculated cross-correlation scores to measure the average relationships between bytes per file type. This process resulted in a 256x256 matrix that represented each byte co-occurrence in the file. As with the previous method, an average vector for each file type was calculated and files were labeled according to their correlation with this average vector.

Lastly, the *file header/trailer* or FHT method collected H bytes from the beginning of the file and T bytes from the end of the file. For each position in this string of bytes, a one-hot encoding is produced. A one-hot encoding creates a vector 256 characters long (all possible hexadecimal values) where all values are zero except the value that actually occurs. Ergo, if the first byte was 0x7, then the 7th value in the vector would be a 1 while the rest would be 0. Since this encoding is applied at each offset, the resulting matrix is of size $(H + T)$x256. Again, an average *fileprint* per file type was calculated using this representation. The algorithms were tested on 120 files representing 30 file types. The results are as follows:

- BFA: 27.5%
- BFC: 45.83%
- FHT: 95.83%

The relatively high accuracy of the header/footer algorithm demonstrates that the location of a byte value plays a significant role in classification accuracy. The byte histogram, on its own, was not very discriminative.

Li et al. extended Mcdaniel's work by using centroids rather than an average vector as the representation of each file type [17]. Files were classified based on their Mahalanobis distance from a centroid. The centroids were chosen using the K-means algorithm. Note, Manhattan distance is the metric the authors chose when applying K-means. The authors further experimented with two variations. In the first, they used multiple clusters per file type since some file types are assumed to be quite diverse. In the second variation they randomly select 80% of their training files for use as exemplars. Under this approach, instead of calculating the Mahalanobis distance from an individual file to a cluster, they find the Manhattan distance to the nearest exemplar. This approach had the highest accuracy (99.6%) by a slim margin. Interestingly, these authors found the highest accuracy by truncating the input down to the first 20 bytes of each file rather than examine the entire file. The lowest accuracy was 82%, which corresponded to the single cluster with no truncation method. Their data set consisted of 800 files across 8 file types.

Karresand [18], Veenman [19], Fitzgerald [20], Beebe [21], [22] all trained machine learning classifiers that used a wide variety of statistical features extracted from files and fragments, which included unigrams, bigrams, bag-of-words, byte rate-of-change, Kolmogorov complexity, and common/longest strings/bytes. Models developed included support vector machines, K-nearest neighbors, and hierarchical clustering. Beebe released an open-source tool, Sceadan, which achieved 73.7%

accuracy on 30 file types and 8 data types by utilizing a linear SVM trained on an input vector of unigrams concatenated with bigrams. Unigrams simply the natural language processing (NLP) term for the byte occurrence count vector or byte histogram. Bigrams are a 256x256 matrix of co-occurrence counts. This data is usually sparse and slow to compute [23].

More recently, deep learning has been used to detect the type of file fragments. File fragments may contain a partial signature or no signature at all, making rule-based approaches useless. Moreover, deep learning has an advantage over other learning algorithms thanks to its automatic feature extraction, which allows deep models to train on raw bytes. Mittal et al. built a deep learning model, FiFTY, to identify file fragments in 2021 [23]. They trained a one-dimensional convolutional neural network to identify 75 different file types. Although the authors experimented with training neural networks and convolutional neural networks on a vector of statistical features that included Shannon entropy, Kolmogorov complexity, deviation, skewness, kurtosis, and mean values (arithmetic, geometric, harmonic) as features, raw bytes as the only input proved the most accurate. Their model outscored Sceadan (77.5% acc vs 69.0% acc) on a data set of 75 file types, which was released along with the FiFTY model.

Image classification techniques have also been applied to the fragment classification problem [24], although this approach is theoretically unsound. Interpreting a stream of bytes from a binary file as a two-dimensional image introduces a width parameter that assumes an underlying 2D structure which is not present in a binary file.

In terms of cybersecurity, polyglot files are the subject of a small number of research papers [4], [10], [11]. Jana and Shmatikov detailed a wide variety of obfuscation methods that target discrepancies between how a malware detector parses a file and how the OS interprets the file [4]. Some of these discrepancies cause the detector to misidentify the file type of the incoming file, resulting in the wrong subset of malicious signatures being applied to the file. These methods are referred to as Chameleon attacks [4]. Werewolf attacks [4], on the other had, exploit discrepancies in how the file is parsed. Jana and Shmatikov describe one type of Werewolf attack as 'ambiguous files conforming to multiple types' [4]. These are polyglot files. They note that a malicious TAR+ZIP stack-type polyglot is incorrectly classified by 20 out of 36 malware detectors hosted on Virustotal at that time.

III. Methodology

A. Polyglot Data Set

Our dataset, described in Table I, consists of 7 monoglot or normal files types as the negative class and 21 polyglot file types as the positive class with an 80/20 train/test split. We should note that these polyglots do not contain malicious code. Our objective is a polyglot filter, not a malware detector. Additionally, the use of benign polyglots greatly simplifies data sharing.

TABLE I: Data Set Overview

Data Set	Train	Test
Monoglot	31,199	7,799
Polyglot	25,210	6,303

There are 3120 of each file type in the monoglot training set and 780 of each file type in the test set. The monoglot file types are as follows:

- PDF
- PNG
- GIF
- TIFF
- JPG
- DCM
- JAR
- Zip
- PE
- ISO

Feeding a subset of the above files to *mitra*, we created 21 different polyglot combinations. There are 1200 of each file type combination in the training set and 300 in test set.

- DCM+GIF
- DCM+JAR
- DCM+ISO
- DCM+PDF
- DCM+Zi[
- GIF+ISO
- GIF+JAR
- GIF+Zip
- JPG+JAR
- JPG+Zip
- PE+ISO
- PE+JAR
- PE+Zip
- PNG+ISO
- PNG+JAR
- PNG+PDF
- PNG+Zip
- TIFF+ISO
- TIFF+JAR
- TIFF+PDF
- TIFF+Zip

Since each pair of file types can only be combined in certain ways (stack, zipper, parasite, cavity), the number of each type of polyglot in this data set is not balanced. Our future work includes the production of a data set that is balanced according to polyglot combination method rather than file type. Table II contains the breakdown of polyglot combination methods in our training and test sets.

TABLE II: Data Set Polyglot Combination Method

Polyglot Method	Train	Test
Stack	10267	2574
Parasite	10301	2542
Zipper	1795	463
Cavity	2847	724

B. Model Selection

We trained a wide variety of models to identify polyglot files. Traditionally, machine learning models used in cyber security applications rely on manual features that are specific to each file type [25], [26]. Since polyglot files can confound file type-specific feature extraction, our models are file type agnostic. In the case of ML models trained through `scikit-learn`, the input vector consists of a byte histogram. In other words, the feature vector is 256 integers in length where the value at each index corresponds to the number of times that index occurs as a byte value in the file. Since files are stored internally as hexadecimal values (hence 256 possible values for each byte), this feature can be used regardless of file type. The models we tested include random

forest, support vector machine, stochastic gradient descent, light GBM, gradient boosting, and CatBoost.

We also trained and tested a deep learning model. Deep learning via a convolutional neural network (CNN) utilizes automatic feature extraction [27]–[30], making it ideally suited for our task. We follow in the footsteps of Mittal et al. in utilizing a deep learning model trained on raw bytes [23]. The deep learning model we chose is designed for binary (two class) classification of binary (compiled) files, namely, Malconv2 [29]. This model is a one-dimensional convolutional neural network developed for malware detection. The first Malconv model demonstrated that CNNs can be effective malware classifiers [28]. However, it had some issues that prompted the authors to develop a second, more effective model.

Since Malconv reads in raw bytes as features, the input file must be truncated if it exceeds the maximum capacity of the model. In the second iteration of the model, Malconv2, the authors exploited the sparse nature of temporal max pooling. This allowed them to partition the varying size files into N sections, then collect only the bytes that would actually get updated with nonzero gradient during the backward pass of the training process. This temporal max pooling allows Malconv2 to intake much larger files than the original Malconv. Moreover, temporal max pooling is used in conjunction with a gating mechanism to provide an attention mechanism. This allows Malconv2 to correlate features that are distant from one another in the byte space. This is especially important for polyglots, where two separate headers might be in the same file at a great distance from one another.

We note that one criticism of Malconv is that it pays far more attention to headers than data or code [31]. Although this was an undesirable trait in a malware detector, it is useful for a polyglot detector since the header structure is closely tied to the file type. Note, some file types utilizes footers instead of headers or have no distinct header/footer, so learning which parts of a file are most relevant is a challenging problem for a polyglot detector.

Our final method concatenated the byte histogram vector with the mime-type output of the utility `file`. Since `file` is very accurate on normal files, we theorized that adding `file`'s output to the feature vector would lessen the learning load on a model.

IV. Results

Firstly, Table III shows that the most common method for identifying file types, `file`, has poor recall overall on the test set of polyglot and monoglot files. Although `file` does perform well specifically on zippers and cavities, these two types are the rarest form of polyglot (in our experience) due to the requirements for mutual comment markers and padding bytes, respectively. We used the `--keep-going` flag when running `file`.

TABLE III: `file` Results on Test Data

	Accuracy	Precision	Recall	F1 Score
Overall	67.82%	99.61%	28.11%	43.85%
Stack	75.75%	90.41%	2.56%	4.99%
Parasite	80.68%	98.75%	21.68%	35.55%
Zipper	99.92%	98.51%	100%	99.25%
Cavity	99.54%	98.99%	95.58%	97.26%

Table IV shows the accuracy, precision, recall and F1 score of a variety of machine learning models and one deep learning model (Malconv2). The ML models are trained on the byte histogram of each file while the deep learning model is trained on the raw bytes.

TABLE IV: ML/DL Results on Test Data

Model	Accuracy	Precision	Recall	F1 Score
Malconv2	96.23	96.35	**95.16**	95.75
Random Forest	91.85	92.54	88.94	90.70
CatBoost	88.19	89.06	83.86	86.39
LightGBM	85.93	85.13	83.02	84.06
GradientBoost	80.11	76.49	80.12	78.26
Support Vector Classifier	62.30	72.29	25.37	37.56

Table V shows the accuracy, precision, recall, and F1 score of the ML models when the feature vector is a concatenation of the byte histogram and the mime type (generated by `file`) for each file. This additional feature boosted the recall of our machine learning models across the board. Adding the mime type to the deep learning model's raw byte input, however, only degraded its output. Therefore, Malconv2 is not included in this table.

TABLE V: ML with `file` Results on Test Data

Model	Accuracy	Precision	Recall	F1 Score
Tuned CatBoost	96.47	96.61	**95.45**	96.03
CatBoost	95.60	95.37	94.75	95.06
Random Forest	95.45	95.85	93.89	94.86
LightGBM	93.98	93.29	93.24	93.26
GradientBoost	89.14	88.96	86.44	87.68
Support Vector Classifier	62.29	72.25	25.37	37.55

V. Discussion and Future Work

We demonstrated that the most common method for file type identification fails to reliably detect polyglots. We then evaluated machine and deep learning models for detecting polyglots and found that Malconv2 had the highest recall. If mime type is included as a feature, then a hyper-parameter tuned version of Catboost has the highest recall. Our work is merely the beginning, however. We need to demonstrate that the above models have high enough throughput to be incorporated into a production-grade malware detection system. We also plan to conduct extensive hyper-parameter tuning to improve model performance. Additionally, we plan to generate a larger data set with a much wider variety of donor types to establish whether our approach is valid for a detection system than can

encounter a huge variety of file types. Finally, we need to address the instances where polyglots are deployed in a non-malicious manner, as in a DICOM+TIFF.

REFERENCES

[1] A. Albertini, "Funky file formats," *International Journal of Proof-of-Concept or Get The Fuck Out*, March 2015. [Online]. Available: https://github.com/angea/pocorgtfo/blob/master/contents/issue07.pdf#page=18

[2] S. Shah, "Weaponized polyglots as browser exploits," *International Journal of Proof-of-Concept or Get The Fuck Out*, June 2015. [Online]. Available: https://www.alchemistowl.org/pocorgtfo/pocorgtfo08.pdf

[3] M. Ortiz, "HIPAA-protected malware? exploiting dicom flaw to embed malware in ct/mri imagery," *Cylera Labs*, 2019.

[4] S. Jana and V. Shmatikov, "Abusing file processing in malware detectors for fun and profit," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 80–94.

[5] R. A. Bridges, S. Oesch, M. E. Verma, M. D. Iannacone, K. M. Huffer, B. Jewell, J. A. Nichols, B. Weber, J. M. Beaver, J. M. Smith *et al.*, "Beyond the hype: A real-world evaluation of the impact and cost of machine learning-based malware detection," *arXiv preprint arXiv:2012.09214*, 2020.

[6] H. S. Anderson and P. Roth, "Ember: An open dataset for training static pe malware machine learning models," *arXiv preprint arXiv:1804.04637*, 2018.

[7] D. Maiorca, B. Biggio, and G. Giacinto, "Towards adversarial malware detection: Lessons learned from pdf-based attacks," *ACM Computing Survey*, vol. 52, 2019.

[8] D.-S. Popescu, "Hiding malicious content in pdf documents," *arXiv preprint arXiv:1201.0397*, 2012.

[9] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," *CoRR*, vol. abs/1508.03096, 2015. [Online]. Available: http://arxiv.org/abs/1508.03096

[10] B. Desjardins, Y. Mirsky, M. Ortiz, Z. Glozman, L. Tarbox, R. Horn, , and S. Horii, "Dicom images have been hacked! now what?" *American Journal of Roentgenology*, vol. 214:4, pp. 727–735, 2020. [Online]. Available: https://www.ajronline.org/doi/pdfplus/10.2214/AJR.19.21958

[11] S. G. Selvaganapathy and S. Sadasivam, "Malware attacks on electronic health records," in *Congress on Intelligent Systems*, 2021, pp. 589–599.

[12] R. Graham, R. Perriss, and A. Scarsbrook, "Dicom demystified: A review of digital file formats and their use in radiological practice," *Clinical Radiology*, vol. 60, no. 11, pp. 1133–1140, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0009926005002199

[13] J. Wolf, "Omg wtf pdf," Presentation at the 27th Chaos Communication Congress, 2010.

[14] "Document management—portable document format—part 1: Pdf 1.7," 2008. [Online]. Available: https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf

[15] "Zip file format specification," *PKWARE Inc.*, 2020. [Online]. Available: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

[16] M. McDaniel and M. Heydari, "Content based file type detection algorithms," in *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, 2003, pp. 10 pp.–.

[17] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog, "Fileprints: identifying file types by n-gram analysis," *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, pp. 64–71, 2005.

[18] G. Mittal, P. Korus, and N. Memon, "Fifty: Large-scale file fragment type identification using convolutional neural networks," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 28–41, 2021.

[19] C. J. Veenman, "Statistical disk cluster classification for file carving," in *Third International Symposium on Information Assurance and Security*, 2007, pp. 393–398.

[20] S. Fitzgerald, G. Mathews, C. Morris, and O. Zhulyn, "Using NLP techniques for file fragment classification," *Digital Investigation*, vol. 9, pp. S44–S49, 2012, the Proceedings of the Twelfth Annual DFRWS Conference. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287612000333

[21] N. L. Beebe, L. A. Maddox, L. Liu, and M. Sun, "Sceadan: Using concatenated n-gram vectors for improved file and data type classification," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 9, pp. 1519–1530, 2013.

[22] N. Beebe, L. Liu, and M. Sun, "Data type classification: Hierarchical class-to-type modeling," in *Advances in Digital Forensics XII*, G. Peterson and S. Shenoi, Eds. Cham: Springer International Publishing, 2016, pp. 325–343.

[23] G. Mittal, P. Korus, and N. Memon, "Fifty: Large-scale file fragment type identification using convolutional neural networks," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 28–41, 2021.

[24] T. Xu, M. Xu, Y. Ren, J. Xu, H. Zhang, and N. Zheng, "A file fragment classification method based on grayscale image," *J. Comput.*, vol. 9, pp. 1863–1870, 2014.

[25] A. Souri and R. Hosseini, "A state-of-the-art survey of malware detection approaches using data mining techniques," *Human-centric ComputComputing and Information Sciences*, vol. 8, 2018. [Online]. Available: https://hcis-journal.springeropen.com/articles/10.1186/s13673-018-0125-x

[26] H. E. Merabet and A. Hajraoui, "Survey of malware detection techniques based on machine learning," *International Journal of Advanced Computer Science and Applications*, vol. 10, 2019.

[27] E. Raff and C. Nicholas, "A survey of machine learning methods and challenges for windows malware classification," *arXiv preprint arXiv:2006.09271*, 2020.

[28] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole exe," in *AAAI Workshops*, 2018.

[29] E. Raff, W. Fleshman, R. Zak, H. S. Anderson, B. Filar, and M. McLean, "Classifying sequences of extreme length with constant memory applied to malware detection," *arXiv preprint arXiv:2012.09390*, 2020.

[30] D. Gibert, C. Mateu, J. Planes, and R. Vicens, "Using convolutional neural networks for classification of malware represented as images," *Journal of Computer Virology and Hacking Techniques*, vol. 15, 2019. [Online]. Available: https://link.springer.com/article/10.1007/s11416-018-0323-0

[31] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining vulnerabilities of deep learning to adversarial malware binaries," *arXiv preprint arXiv:1901.03583*, 2019.