Using Your Head: Identifying Windows Malware by Deep Learning on PE Headers

Jasmine Cai
jlcai@umass.edu

Jeremy Dunn
jeremydunn@umass.edu

Timafei Hushchyn
thushchyn@umass.edu

University of Massachusetts Amherst

Abstract

In this work, we address the problem of identifying unknown malware samples using a limited section of their program: the header of the Windows PE file. Most methods do not utilize deep learning techniques and instead opt for rule-based machine learning algorithms or supervised learning algorithms like the Random Forest algorithm, if any machine learning is utilized at all. In our work, we analyze the effectiveness of utilizing deep learning neural nets such as multi-layer perceptrons (MLPs) and fully-connected neural networks (FCNets) that we create and train for this malware identification task. Our results demonstrate that a generic Random Forest classifier is effective at predicting malware based on its header (with 98.93% accuracy) and deep learning approaches are also effective, with a MLP having 94.98% accuracy and a FCNet having 95.36% accuracy. We also explore the rate in which our three models can classify samples per second across two different types of hardware to ensure flexibility of utilizing our methods across many potential security appliances, both of which obtain high rates of classification.

1. Introduction

Antivirus and intrusion detection systems utilize many different techniques to determine if files are malicious or not. Among these techniques, machine learning algorithms have shown promise, specifically in the realm of supervised learning. However, not all machine learning techniques have been explored. In this paper, we explore deep learning techniques involving neural networks to perform malware detection. By limiting the scope of the data we look at to a few specific header fields, we allow for simple and fast classification of malware samples while maintaining the flexibility that a machine learning based approach allows. This fast classification makes the approach practical in situations that require a fast turnaround, such as scanning incoming files through a network-based Intrusion Detection System.

We classify samples as ”clean” or ”malicious” using a limited section of their program - namely the header of the Windows PE file. The Windows PE file is a file format for executable files in the Windows operating system. Previous work has indicated that the header of PE files can determine if the file is malware or not, since it contains the main information about a process runtime [5]. The format of a file PE header can be seen below, along with some examples of section header names (i.e. .text, .data, .rsrc, etc.) being some of the key features we looked at alongside unpictured optional headers like magic numbers.

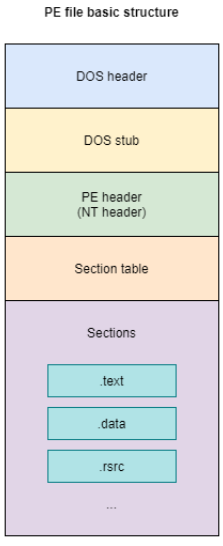


Figure 1. Figure of Windows PE file basic structure from [2]

We explore several types of classifier algorithms: random forests (RFs), multilayer perceptrons (MLPs), and fully-connected neural networks (FCNets). We assign importance to each feature by assigning embeddings and de-

termining how a model places its own importance on each feature, as well as performing our own frequency statistic tests on each feature to determine if any specific headers have values of note.

Our dataset is comprised of Windows PE files, with our target features all being extracted from their headers. We source our "clean" files from a fresh Windows distribution installed in a virtual machine. Several common applications were installed on the system, such as VMware, Google Chrome, and Office365. The test system was scraped for all PE files it contained, and each valid file was processed. The "malicious" files were sourced from Vx-Underground [9], a well known repository of malware that is commonly used by analysts when developing tools. We downloaded four "collections" - the Bazaar, InTheWild, VirusShare, and VirusSign collections. These collections contained tens of thousands of files and totaled roughly 100 GB of data.

Each file was processed using the Python 'pefile' library and had all of its relevant header information extracted. The header values and hashes were stored in a JSON object indexed by the MD5 hash of the file. This ensured there were no duplicate files in the dataset. Once extraction was complete, our dataset consisted of 36,037 clean samples and 26,979 malicious samples.

Our results will show the rate of false positives where files are predicted as containing malware but are "clean", false negatives, and true positives and negatives. We have high truth rates where each predicted label is correct and have low false rates. This was be achieved by having high precision and recall scores in addition to general accuracy.

Our results evaluation is based on each model's ability to correctly predict the label of the testing data after training, as well as the speed with which the classification was made. Our model's effectiveness will be evaluated on the rates stated above.

2. Related work

2.0.1 Dataset Collection

Collection of both malware and non-malware samples has traditionally been performed via web scrapers which download samples from the Internet.

Another common tactic in past literature to collect malware samples is to use crowd-sourced datasets that already exist [4] [6]. Non-malware samples, commonly called benign samples in literature, were often collected from default Windows files [3], although another technique was to get samples from popular programs [6]. An issue pointed out by Kumar et al. is that it would be useful to have a common dataset to do comparisons with past work, but such datasets are often unavailable. Unfortunately, this is still an issue, despite the fact that some papers claim to make their respective dataset available. On the other hand, old samples might not be as useful for considering modern efficacy, though it would be useful to do inter-method comparison.

2.0.2 Malware Sample Identification

Previous approaches to identifying unknown malware samples rarely attempt deep learning techniques and instead rely on manual sorting procedures, rule-based machine learners, or other supervised learning algorithms like Random Forest.

Liao utilized their own rule-based algorithm without machine learning to detect malware in files. They wrote their own collection of rules and evaluated different combinations of five PE header features to see which is most effective at detecting malware in that file. Their highest correct detection accuracy was 99.5% with all five features. This approach is limited to the five features that were studied. In our approach, we aim to be able to use our models to find features of note themselves in PE headers to determine the likelihood of a file being malware after having a feature of note present in its header.

Schultz et al. utilized the rule-based machine learner RIPPER and probabilistic classifier Naive Bayes as their models for detecting malware in files. The overall accuracy of their RIPPER model of correctly detecting malware in their files was 89.36% with DLL function calls. Their Naive Bayes classifier fared better with an accuracy of 97.11%. Their RIPPER model had a peak detection rate of 71.05% and their Naive Bayes model had a detection rate of 97.43%. We seek to try to get higher detection rates, at least compared to the RIPPER model.

Markel and Bilzor used three non-deep learning classifying approaches: Naive Bayes, decision trees, and logistic regression. They used metadata extracted from Windows PE Headers as the input data. The decision tree classifier was the most effective followed by logistic regression and finally the Naive Bayes classifier. Though, when they lowered the prevalence of the malware samples in the training and test data, they noticed a decline in the performance in all three classifiers, especially in the logistic regression classifier. This is of interest because in actual uses, there will be a bias in the data toward having lots of benign samples.

Firdausi et al. was of interest because in addition to using various non-deep learning methods such as SVMs and decision trees, they utilized MLPs. For their data input, they generated XML files that described the behavior of benign and malware samples and then trained classifiers based on those XML files as input with some preprocessing steps. With feature selection, they found that all the techniques used scored approximately 92%, including the MLP model.

Overall, the literature suggests that using machine learning to classify malware, including from Windows PE Headers, is effective. One paper also showed that using deep

learning is effective, though they used a different approach to input data. Thus, it remains to be explored if deep learning is a good approach to classifying malware via Windows PE Headers. This is the gap that we will address in our paper.

3. Method

3.1. Dataset Collection

To gather clean data samples, we set up a Virtual Machine with a fresh copy of Windows 11 Pro Version 24H2. Several common applications were then installed on the VM, such as VMware, Office365 and Google Chrome. We then created a Python script that scanned the C: drive for every PE file on the system and processed every file it found.

To gather the malware samples, we utilized VxUnderground's extensive malware repository, namely their Bazaar, InTheWild, VirusShare, and VirusSign collections. Each collection was downloaded to a sandboxed malware analysis VM, decrypted, and verified for integrity through validation of checksums. The same processing script was used as for the clean files, except this time it was limited to the malware collection as opposed to the operating system.

For each PE file found, the following procedure was run. The specific values were obtained using the Python pefile package, which is a well established tool for extracting data from PE files.

Algorithm 1 Algorithm for processing a Windows PE file to get the data from headers of note.

```
procedure PROCESSFILE(file)
    data = ()
    for section_name in file.section_names do
        data["sections"] ← section_name
    end for
    for optional_header in file.optional_headers do
        data["optional"] ← {optional_header.name: optional_header.size}
    end for
    return data
end procedure
```

The data was then stored in JSON format for easy retrieval of information.

In all, we processed 36,037 unique clean samples and 26,979 unique malicious samples, for a total of 63,016 samples. This provided a varied and robust training set from which we obtained our results.

3.2. Feature Extraction, Embeddings, and Preprocessing

In the dataset of 63,016 files, we identified 2003 unique headers. This was too many to train on directly, as our model would have run into issues with dimensionality.

After extensive testing, we settled on a fifteen parameter model. Fourteen of those parameters were from the optional headers section of the binary, and one parameter was a combined representation of the section names.

To determine the most impactful optional headers to use, we calculated the mean and standard deviation of each header across the respective clean and malicious datasets. The results of this are included in Table 4. We manually reviewed the data, then selected the headers that appeared the most different between the clean and malicious datasets. The fourteen optional headers we chose to focus on are: SizeOfCode, SizeOfInitializedData, SizeOfUninitializedData, BaseOfCode, SectionAlignment, FileAlignment, SizeOfHeaders, SizeOfStackCommit, SizeOfHeapCommit, LoaderFlags, NumberOfRvaAndSizes, DllCharacteristics, MajorImageVersion, and CheckSum.

If a dataset sample did not have one or multiple of these optional section headers, the optional header representation was set to the placeholder 0xFFFFFF, or 4294967295. This is a value not found in any of the optional headers that were present, and allows for the model to identify if one of the headers is missing and potentially use that in its classifications.

Literature has also indicated that the section names of a binary can provide insight into whether it is malicious. In particular, it has been noted that uncommon section names are far more common in malware than in clean files. There are thousands of potential section header names, with ones that are present across all binaries, such as ".rdata", ".data", ".pdata", ".rsrc", and ".reloc", but also miscellaneous unique ones that can be set by a developer.

To account for this in a reasonably space efficient manner, we created a fifteenth "header" alongside the fourteen optional headers called "SectionHeaders". This header consists of the bitwise XOR of the UTF-8 representation of a binary's section header names. We chose the XOR operation because of it's commutative and associative properties, which will allow the XOR of headers with the same section names to look the same, while headers with unique names will have a different pattern.

The data that we passed into our models ultimately became a tensor for each binary, where each tensor consisted of the values of each of these optional headers and the XOR'ed section names. For example, our first binary had the information in Table 1.

To allow all of the values to have the same type and make processing the model easier, we converted the XOR of the section headers to an integer – see Algorithm 2. This would be our new 'SectionHeaders' value: 309388837382. As such, our tensor representation for the above would be [65536, 57344, 0, 4096, 4096, 4096, 4096, 8192, 4096, 0, 16, 49504, 10, 168463, 309388837382].

Table 1. Binary #1 (md5: a9f8e7392f8f8661997d67fbde92eed5) optional header values and section header names prior to transforming into dataset tensor.

Header	Value
SizeOfCode	65536
SizeOfInitializedData	57344
SizeOfUninitializedData	0
BaseOfCode	4096
SectionAlignment	4096
FileAlignment	4096
SizeOfHeaders	4096
SizeOfStackCommit	8192
SizeOfHeapCommit	4096
LoaderFlags	0
NumberOfRvaAndSizes	16
DLLCharacteristics	49504
MajorImageVersion	10
Checksum	168463
SectionHeaders	fothk, .rdata, .data, .pdata, .rsrc, .reloc

Algorithm 2 Algorithm for creating our fifteenth data point for a binary, the "SectionHeader" representation.

```
procedure XORALGORITHM(section_names)
    hex_names = ()
    result = "\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    for section_name in section_names do
        hex_names ← hex(section_name)
    end for
    for item in hex_names do
        result ← item ⊕ result
    end for
    return int(result)
end procedure
```

The labels were, as mentioned before, 0 denoting a "clean" binary and 1 denoting a "malware" binary.

3.3. Models

We investigated two different types of deep learning neural network models to apply to the task of malware identification via PE headers: a multi-layer perceptron (MLP) and a fully-connected net (FCNet). We also created a generic Random Forest classifier, which was seen in many other related works tackling the same problem. Our RF classifier utilized Scikit-Learn [7] and our MLP and FCNet models utilized Pytorch [1].

The MLP is comprised of two linear layers with one non-linearity (i.e. ReLU) in-between. Our FCNet has a similar structure but with more fully-connected layers in between, which we tuned. Both of these models act like a binary classification model to predict the label of our inputs.

For the MLP and FCNet models, we also used min-max scaling because the different features had wildly different magnitudes. This leads to poor gradient calculations (either very big or very small) and leads to our models having basically the same performance as a coin-flip. Min-max scaling fixes this because it gets all of our features to be in a range of [0,1] by rescaling so that 0 as the minimum value of a feature and 1 as the maximum value of a feature which thus leads to more stable calculations.

4. Results

4.1. Data Collection

We have successfully obtained 36,037 clean and 26,979 malware Windows PE binaries, making a total of 63,016 binaries for our dataset with 2003 unique headers.

4.2. Data Preprocessing

We successfully preprocessed the data of all of these binaries, extracting the fourteen optional headers of note as well as the section header representation for each binary.

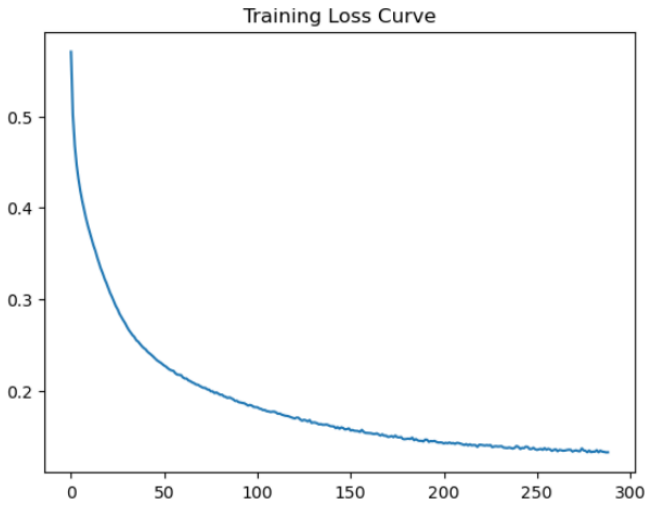
4.3. Model Accuracy, Precision, and Recall

With the RF classifier, we tested many different combinations of parameters. We did not adjust the amount of either clean or malware samples very much during training for the milestone. We varied the hyperparameters, number of estimators and max depth, to obtain preliminary results of our classifier. When there were 654 clean samples, 266 malware samples, 10 estimators, and a max depth of 40, we were able to obtain 92.12% accuracy, 95.5% precision, and 63.69% recall. The high precision, which was close to 100%, and lower recall tell us that the RF classifier tends to be cautious in predicting and generally correct when it does, but might be missing many true classifications. After the milestone, we increased the number of samples a lot so we had 36k clean samples and 27k malware samples. We had the best accuracy with max depth set to none and number of estimators set to 100 with 97% for precision and 97% recall with an overall accuracy of 96%. The train-test split we used was 0.3 for test size, 0.7 for training size. High precision is indicative of very few false positives/FPs and high recall is indicative of very few false negatives/FNs, so our RF model is performing very well and near optimally.

Our MLP model has two linear layers with a ReLU layer in-between. The output size was 1 since our goal was binary classification. During the milestone, it seemed that we ran into issues that would drastically change the model as we changed the various parameters (i.e. which loss function to use, which learning rate to choose, etc.). After printing our results, we noticed the predictions were decent in early epochs, but steadily became biased to only predicting "0"

(i.e. that a given header was clean) as we continued through the other epochs. Our model would consistently have high accuracy, with an average of 92.44% accuracy. Since our input dataset was primarily clean data to begin with (since our dataset is unbalanced) predicting ”0” would be generally correct. We attributed this to potentially being an issue with our dataset sampling. We fixed this issue as aforementioned by obtaining a larger dataset of clean and malware samples and adjusted the way that we preprocess the data. With 44111 data samples (both clean and malware), hidden layer size of 128, and learning rate of 1e-3, we got an accuracy of 94.9% with 94.37% precision and 96.95% recall. Although slightly worse than our RF model, our MLP model is performing near optimally with our high accuracy, precision, and recall.

Figure 2. Training loss curve of the Multi-layer Perceptron (MLP) Model.



Our FCNet model is comprised of linear layers with non-linearities (ReLU) between with a varying number of hidden layers, essentially a more complex MLP model. Like our MLP model, the output size was 1 due to our goal of binary classification. With 44111 data samples (both clean and malware), hidden layer sizes of 256, 128, and 64, learning rate of 1e-3, epoch count of 50, and batch size of 64, we got an accuracy of 95.36% with 98.74% precision and 93.21% recall. So, our FCNet also performs slightly worse than our near-optimal RF classifier. Despite this, our FCNet performs marginally better than our MLP, with our FCNet seemingly having less false positives but more false negatives. We tried a couple of different optimization methods, Adam seemed to be the most effective. We also attempted to use dropout at a couple different levels of activation but it seems that with our dataset it only introduced unwanted noise.

Figure 3. Training loss curve of the Fully-connected Neural Net (FCNet) Model.

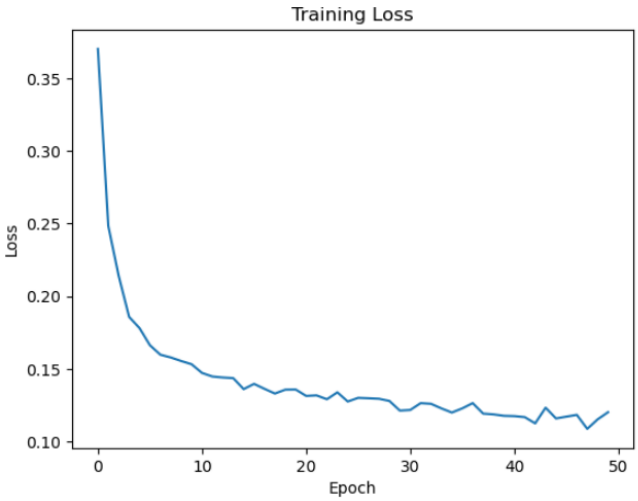


Table 2. **Accuracy, precision, and recall**: Accuracy, precision (correct positives classified / all guessed positives), and recall (correct positives classified / all actual positives) for the Random Forest (RF) classifier, Multi-layer Perceptron (MLP), and Fully-connected Neural Net (FCNet).

	RF	MLP	FCNet
Accuracy	0.9892621	0.949855	0.953610
Precision	0.9892989	0.943724	0.987360
Recall	0.991953	0.969476	0.932106

4.4. Model Performance

A major concern for this kind of malware identification is performance, as many of the situations in which a binary malware classifier is used can be considered as ”fast turnaround”, such as during network scanning with an IDS or when downloading a file from the internet. If a classifier is too slow, it limits its use.

In Table 3, we show that each type of model we tested was fast enough to be used in these fast turnaround situations. Each set of timings was obtained using the models in ”CPU only” mode to better simulate the kinds of hardware that would be available to a security appliance. Although the Random Forest model was the fastest, all three model types can classify in a fraction of a second and with minimal resources.

Additionally, we ran these three models across two different types of systems to get a better idea of the performance differences in potential security appliances with varying hardware. We ran experiments on a typical laptop equipped with a Intel i5-10210U (8) @ 4.200GHz as well as an enterprise-like server equipped with a Intel(R)

Table 3. **Performance**: Number of samples classified per second for the Random Forest (RF) classifier, Multi-layer Perceptron (MLP), and Fully-connected Neural Net (FCNet) across two different environments (Intel i5-10210U (8) @ 4.200GHz vs. Intel(R) Xeon(R) Gold 6148 CPU (80) @ 2.40GHz).

	RF	MLP	FCNet
Intel i5-10210U (8) @ 4.200GHz	6301.6667	420.1111	138.2328
Intel(R) Xeon(R) Gold 6148 CPU (80) @ 2.40GHz	7693.1840	285.8842	215.8043

Xeon(R) Gold 6148 CPU (80) @ 2.40GHz. Based on an investigation of a process viewer 'htop' while running the RF model, we believe RF runs on a single-core, making the RF model run slower (fewer samples classified per second) on a typical laptop over an enterprise-like server. Additionally, the laptop had more samples classified per second than our enterprise-like server, which we believe is due to the laptop's fewer cores but faster clock. The FCNet model, due to its heavy computation, produced fewer samples classified per second on the laptop than the enterprise-like server.

5. Conclusion

In this work, we explored the idea of utilizing deep learning to classify Windows malware samples utilizing only PE header information. We utilized three different models and a large dataset collected and preprocessed in a unique way.

Our results indicated that not only is deep learning on PE headers a viable method for identification of malware, but by limiting the amount of information looked at, we can get very close to state of the art results with minimal overhead.

Although our deep learning methods were effective, the Random Forest classifier still performs the best. We reason that this is due to the structured, relatively few feature, discrete nature of the dataset. Neural networks deal best with continuous data like images, video, and audio, or with complex data like natural language which has an expansive vocabulary and grammar. On the other hand, this dataset only has a few features and they only have so many different values. Thus, a Random Forest classifier has a better architecture to take advantage of this dataset because it is able to directly leverage the structure of it. Meanwhile, the MLP and FCNet create some noise while trying to learn the dataset so that it can be more generalizable which is useful when you have a cat in different positions or different lighting but not when you do not really have these distortions in a more abstract piece of data such as a header file.

We also explored running each of the three models on different types of hardware to accommodate for the differing hardware that intrusion detection systems (IDSs) run on. Both of these types of hardware were able to classify large amounts of samples per second.

In future works, we would like to look at different headers and perform further analysis to see which headers are the most impactful on classification in the hopes of improving our performance even further. In addition, taking into account section sizes as well as names could provide another indicator a model could train on.

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, 2024. 4

[2] cocomelonc. Windows shellcoding - part 3. pe file format. 1

[3] Ivan Firdausi, Alva Erwin, Anto Satriyo Nugroho, et al. Analysis of machine learning techniques used in behavior-based malware detection. In *2010 second international conference on advances in computing, control, and telecommunication technologies*, pages 201–203. IEEE, 2010. 2

[4] Ajit Kumar, K.S. Kuppusamy, and G. Aghila. A learning model to detect maliciousness of portable executable using integrated feature set. *Journal of King Saud University - Computer and Information Sciences*, 31(2):252–265, 2019. 2

[5] Yibin Liao. Pe-header-based malware study and detection. Accessed: 2025-03-14. 1, 2

[6] Zane Markel and Michael Bilzor. Building a machine learning classifier for malware detection. Accessed: 2025-03-14. 2

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 4

[8] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables, 2001. 2

[9] vx underground. 2

Table 4. All optional headers found in our sample binaries, with the mean and standard deviation of appearance of each field in either clean or malware binaries.

OPTIONAL HEADER		MEAN	STD. DEV
Magic	clean	398.7543635707745	127.94492859890322
	malware	333.81107528077393	112.42737874603714
MajorLinkerVersion	clean	15.140050503649027	10.364115192992088
	malware	10.318506986915748	8.89393303909284
MinorLinkerVersion	clean	27.775536254405196	16.576888164262762
	malware	9.500611586789725	15.047849524212433
SizeOfCode	clean	523135.0131808974	4476454.801637794
	malware	772660.7794580971	22058735.64974974
SizeOfInitializedData	clean	381318.63451452676	3596108.151768076
	malware	1287262.0976314913	8153821.424978578
SizeOfUninitializedData	clean	6262.271332241862	213992.5431117511
	malware	449410.21713184327	1189689.8421901606
AddressOfEntrypoint	clean	271583.6142020701	2264092.3408621377
	malware	1801769.4346343451	69914096.20842524
BaseOfCode	clean	4889.680273052696	32304.27350265249
	malware	449934.82278809446	1191853.4522491896
ImageBase	clean	3583276193291366.5	2.5707038688337206e+17
	malware	37694272213.22777	2266001763618.479
SectionAlignment	clean	4748.085356716708	1660.347758138156
	malware	154771.0790244264	24693885.228453256
FileAlignment	clean	1525.6976995865361	1614.1102269020255
	malware	79079.36843470848	12769428.345477177
MajorOperatingSystemVersion	clean	7.5792935038987705	2.705877159475324
	malware	6.883316653693614	350.06679980168525
MinorOperatingSystemVersion	clean	0.12748008990759496	0.46839675823816446
	malware	1.9254976092516403	290.9388170090119
MajorImageVersion	clean	66.704248411355	1111.5546360172523
	malware	5.302568664516847	384.30532968456095
MinorImageVersion	clean	70.85015400838027	1189.2707267359413
	malware	3.141295081359576	302.9636582487618
MajorSubsystemVersion	clean	7.081943558009823	2.400623703037888
	malware	5.935208866155158	178.04256201264815
MinorSubsystemVersion	clean	0.33557177345506006	0.7967545677358044
	malware	0.5462396678898402	64.11376264450891
Reserved1	clean	0.0	0.0
	malware	149386.124244783	24536592.415149458
SizeOfImage	clean	984301.0292754669	6922714.68150624
	malware	2924184.126468735	25125907.316255175
SizeOfHeaders	clean	1733.167355773233	1503.0860490250893
	malware	10223.187145557657	1334443.536601544
CheckSum	clean	931112.7579987235	6964823.554262584
	malware	2018312.588531821	37282581.4800254
Subsystem	clean	2.552987207592197	0.708705533121903
	malware	4.359724229956633	354.1415732395719
DllCharacteristics	clean	18053.380248078363	14512.904923390875
	malware	17294.384669557803	17493.374886524613
	clean	916377.1110802786	992116.6869331253

Table 4 continued from previous page

SizeOfStackReserve	malware	1330521.401238	12629691.18946378
SizeOfStackCommit	clean	5080.79229680606	21344.88628088939
	malware	107496.25616220022	14024984.131094605
SizeOfHeapReserve	clean	1013956.8936370952	191874.9164820748
	malware	1300325.3446384224	23975571.335348666
SizeOfHeapCommit	clean	4090.9989177789494	931.5417599199343
	malware	54201.69887690426	8226110.497778804
LoaderFlags	clean	8700.064489274911	95115.65433612066
	malware	89494.38192668372	14699435.204358535
NumberOfRvaAndSizes	clean	15.999445014845852	0.074495258459515
	malware	4032620.09844694	61102414.934546836