The Workshops of the Thirty-Second
AAAI Conference on Artificial Intelligence

Malware Detection by Eating a Whole EXE

Edward Raff,[1,3,4] **Jon Barker,**[2] **Jared Sylvester,**[1,3] **Robert Brandon,**[1,3,4]
Bryan Catanzaro,[2] **Charles Nicholas**[4]
[1]Laboratory for Physical Sciences, [2]NVIDIA, [3]Booz Allen Hamilton, [4]University of Maryland, Baltimore County
{edraff,jared,rbrandon}@lps.umd.edu, {jbarker,bcatanzaro}@nvidia.com, nicholas@umbc.edu

Abstract

In this work we introduce malware detection from raw byte sequences as a fruitful research area to the larger machine learning community. Building a neural network for such a problem presents a number of interesting challenges that have not occurred in tasks such as image processing or NLP. In particular, we note that detection from raw bytes presents a sequence problem with over two million time steps and a problem where batch normalization appear to hinder the learning process. We present our initial work in building a solution to tackle this problem, which has linear complexity dependence on the sequence length, and allows for interpretable sub-regions of the binary to be identified. In doing so we will discuss the many challenges in building a neural network to process data at this scale, and the methods we used to work around them.

1 Introduction

The detection of malicious software (malware) is an important problem in cyber security, especially as more of society becomes dependent on computing systems. Already, single incidences of malware can cause millions of dollars in damages (Anderson et al. 2013). Anti-virus products provide some protection against malware, but are growing increasingly ineffective for the problem. Current anti-virus technologies use a signature-based approach, where a signature is a set of manually crafted rules in an attempt to identify a small family of malware. These rules are generally specific, and cannot usually recognize new malware even if it uses the same functionality. This approach is insufficient as most environments will have unique binaries that will have never been seen before (Li et al. 2017) and millions of new malware samples are found every day. The limitations of signatures have been recognized by the anti-virus providers and industry experts for many years (Spafford 2014). The need to develop techniques that generalize to new malware would make the task of malware detection a seemingly perfect fit for machine learning, though there exist significant challenges.

To build a malware detection system, we must first determine a feature set to use. One intuitive choice is to use features obtained by monitoring program execution (APIs called, instructions executed, IP addresses accessed, etc.). This is referred to as dynamic analysis. While intuitively appealing, there are many issues with dynamic analysis in practice. To conduct dynamic analysis, malware must be run inside a specially instrumented environment, such as a customized Virtual Machine (VM), which introduces high computational requirements. Furthermore, in some cases it is possible for malware to detect when it is being analyzed. When the malware detects an attempt to analyze it, the malware can alter its behavior, allowing it to avoid discovery (Raffetseder, Kruegel, and Kirda 2007; Garfinkel et al. 2007; Carpenter, Liston, and Skoudis 2007). Even when malware does not exhibit this behavior, the analysis environment may not reflect the target environment of the malware, creating a discrepancy between the training data collected and real-life environments (Rossow et al. 2012). While a dynamic analysis component is likely to be an important component for a long term solution, we avoid it at this time due to its added complexity.

We instead take a static analysis approach, where we look at information from the binary program that can be obtained without running it. In particular, we look at the raw bytes of the file itself, and build a neural network to determine maliciousness. Neural nets have excelled in learning features from raw inputs for image (Szegedy et al. 2015), signal (Graves, Mohamed, and Hinton 2013), and text (Zhang and LeCun 2015) problems. Replicating this success in the malware domain may help to simplify the tools used for detecting malware and improve accuracy. Because malware may exploit bugs and ignore format specifications, parsing malicious files and using features that require domain knowledge can require significant and nontrivial effort. Since malware is written by a real live adversary, such code will also require maintenance and improvement to adjust to changing behavior of the malware authors.

Since we desire to learn a system from raw byte inputs, from which higher level representations will be constructed, we choose to use a neural network based approach. However, there exist a number of challenges and differences for this domain that have not been encountered in other tasks. These challenges make research in malware detection intrinsically interesting and relevant from a machine learning perspective beyond merely introducing these techniques to a novel domain. For Microsoft Windows Portable Executable

(PE) malware, these challenges include but are not limited to:

1. The bytes in malware can have multiple modalities of information. The meaning of any particular byte is context sensitive, and could be encoding human-readable text (e.g., function names from the import table), binary code, arbitrary objects such as images (from the resource/data sections of a binary), and more.

2. The content of a binary exhibits multiple types of spatial correlation. Code instructions in a function are intrinsically correlated spatially, but this correlation has discontinuities from function calls and jump commands. Further, the contents at a function level can be arbitrarily rearranged if addresses are properly corrected.

3. Treating each byte as a unit in a sequence, we are dealing with a sequence classification problem on the order of *two million time steps*. To our knowledge, this far exceeds the length of input to any previous neural network based sequence classifier.

4. Our problem has multiple levels of concept drift over time. The applications, build tools, and libraries developers use will naturally be updated, and alternatives will fall in and out of favor. This alone causes concept drift. But malware is written by a real-life adversary, and is often intentionally adjusted to avoid detection.

Our contributions in this work are the development of the first, to our knowledge, network architecture that can successfully process a raw byte sequence of over two million steps. Others have attempted this task, but failed to outperform simpler baselines or successfully process the entire file (Anderson 2017), in part because the techniques developed for signal and image processing do not always transfer to this new domain. We identify the challenges involved in making a network detect malware from raw bytes, and the initial methods one can employ to successfully train such a model. We show that this model learns a wider breadth of information types compared to previous domain-knowledge free approaches to malware detection. Our work also highlights a failure case for batch-normalization, which initially rendered our model unable to learn.

2 Related work

There are two primary themes of past work: the application of neural networks to ever longer sequences, and the application of neural networks to malware detection. The use of Recurrent Neural Networks (RNNs) has been historically prevalent in any work involving sequences, but the processing of raw bytes far exceeds the scale attempted in previous work by orders of magnitude. For malware detection, all of these previous applications use a significant amount of domain knowledge for feature extraction. In contrast, our goal is to minimize the use of such domain knowledge, and explore how much of the problem can be solved without specifying any such information.

2.1 Neural Networks for Long Sequences

Little work has been done on the scale of sequence classification explored in this work. The closest in terms of pure sequence length is WaveNet (Oord et al. 2016). WaveNet attempts to advance the state-of-the-art in generative audio by ignoring previous feature engineering, and instead using the raw bytes of the audio as the input feature and target. This results in a sequence problem with 16,000 time steps per second of audio. Wide receptive fields for this task (4,800 steps) were obtained through the use of dilated convolutions (Yu and Koltun 2016) and by training a very deep architecture. Ultimately, their work is still on the order of two magnitudes smaller in sequence length compared to our malware detection problem.

The use of dilated convolutions to handle sequence length has become a common trend, as for example in the ByteNet model for machine translation (Kalchbrenner et al. 2016). While translation can result in relatively long sequences, their sequence length is smaller than WaveNet's audio generation. While we did explore dilated convolutions in this work, we did not find them to perform any better or worse than standard convolutions for our problem. We suspect this is due to the different nature of locality in binaries, that the values in the "holes" of the dilation are easier to assume or interpolate for spatially consistent domains like image classification, but are not obviously interpolated for binary content.

We note another trend when working with long sequences: the use of RNNs that operate at different frequencies. Mehri et al. (2017) used such an architecture for audio classification, but exploited the generative nature of the task to train on sub-sequences of only 512 time steps. Other works that have made use of RNNs operating at multiple frequencies have similarly worked on sequences that do not exceed thousands of time steps (Koutnik et al. 2014; Neil, Pfeiffer, and Liu 2016).

In addition to the difficulties in dealing with the unusually long sequences that we confront, we must also contend with a lack of information flow. When making a benign/malicious classification of a binary we obtain only one error signal, which must be used to inform decisions regarding all 2 million time steps. In contrast, neural translation models and autoregressive models such as WaveNet are attempting to predict not an overall classification, but the next word or byte. This provides them with frequent label information at each time step, resulting in a near 1:1 mapping between input size and labels from which to propagate errors. Such frequent gradient information is not available for our problem, increasing the learning challenge even before considering sequence length.

2.2 Neural Networks for Malware Detection

There has been little work thus far in applying neural networks to malware detection, and no current work we are aware of that attempts to do so from the raw bytes of the entire binary. It has recently been demonstrated that fully connected and recurrent networks are able to learn the malware identification problem when trained on just 300 bytes from

the PE-header of each file (Raff, Sylvester, and Nicholas 2017). Based on the positive results obtained, the current work extends those results by training networks on entire, several million byte long executables, and encounters a wide breadth of potential byte content.

The work of Saxe and Berlin (2015) is closest to ours at a feature level, as it uses a histogram of byte entropy values for features. This is in addition to a histogram of ASCII string lengths, PE imports, and other meta-data that can be obtained via static analysis. This approach produces some small level of information from the whole file, but discards most information about the actual content of the binary in the process, as it creates a fixed length feature vector to use as input to the network.

Most recent work in the application of deep learning to malware detection has used features extracted via dynamic analysis, where the binary is run in a virtualized environment to obtain information about its execution. Kolosnjaji et al. (2016) tackled the related problem of malware family classification (i.e., which family does a particular malicious file belong to?) using a combination of convolutions followed by LSTMs to process the sequence of API calls a malware file generated under dynamic analysis. This was after down-selecting to just 60 kernel API calls to track.

Huang and Stokes (2016) performed manual feature engineering of API calls into 114 higher-level concepts, and combined these API events with input arguments to the original function calls as well as tri-grams. Rather than just predict maliciousness, they performed malware detection and family classification with the same model (i.e., weights shared between two tasks). This approach improved the performance of the model on both tasks, and would be compatible with our design in this work.

These prior works in malware detection tend to use significant manual feature engineering, which requires a significant if not rare level of domain expertise. Those using dynamic analysis often rely on sophisticated non-public emulation environments to mitigate the challenges with dynamic analysis, which significantly increases the effort to reproduce work. Our proposed approach eliminates this domain knowledge-specific code and feature processing, reducing the amount of specialized code and reducing the barrier to reproduction and extension.

We note one unfortunate aspect of much of the previous work in malware detection, including some of our own, namely, the use of data that is not available to the public, for various reasons. Data that can be readily obtained by the public is often not of a sufficient quality to be usable in practice, as we will discuss in 3. This also means we cannot meaningfully compare accuracy numbers across works, as different datasets are used with different labeling procedures.

3 Training data

For this work we use the same training and testing data as in Raff et al. (2016). Specifically, we use the Group B training data, and Group A & B testing data. Group B data was provided by an anti-virus industry partner, where both the benign and malicious programs are meant to be representative of files seen on real machines. The Group B training set consists of 400,000 files split evenly between benign and malicious classes. The testing set has 77,349 files, of which 40,000 are malicious and the remainder are benign.

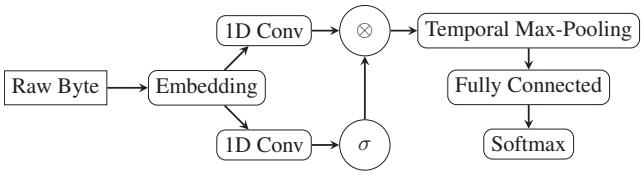

Figure 1: Architecture diagram of MalConv model.

The Group A data was collected in the same manner as most work in the malware identification literature (Schultz et al. 2001; Kolter and Maloof 2006) , which is available to the public. The benign data (or "goodware") comes from a clean installation of Microsoft Windows, with some commonly installed applications (e.g., Firefox, Flash, etc) and the malware comes from the VirusShare corpus (Roberts 2011). The Group A test set contains 43,967 malicious and 21,854 benign testing files.

It was found that training on the Group A-style data results in severe overfitting (Raff et al. 2016), learning to recognize "from Microsoft" instead of "benign", which does not generalize to new data. That is to say, a model trained on Group A doesn't generalize to Group B, but a model trained on Group B does generalize to Group A. For this reason we only perform our experiments with the Group B training data, and test on both groups. Testing in this manner allows us to better quantify generalization ability, as the data are from different sources. This minimizes shared biases, and gives us a potential upper and lower-bound on expected accuracy.

We use both group's test sets, as this allows us to better judge the generalization ability of the models. Group B's test performance is important, as it is supposed to represent data in the wild, but may have a shared common bias due to how Group B data was collected. Testing on the Group A data, which is collected in a different manner, then is a stronger test of generalization as the data has fewer common biases with Group B. Because of this, we consider Group A's test performance more interesting than Group B's. We also want our model to have similar performance on both test sets, which would indicate the features learned are widely useful.

In addition, reaching out to the authors and original company, we have obtained a larger training corpus of 2,011,786 binaries, with 1,000,020 benign and 1,011,766 malicious. We use this larger dataset to show that our new MalConv architecture continues to improve with increased training data, while the byte n-gram approach appears to have plateaued in terms of performance.

4 Model Architecture

When designing our model three features were desired: 1) the ability to scale well with sequence length, 2) the ability

to consider both local and global context while examining an entire file, and 3) an explanatory ability to aid analysis of flagged malware. A block diagram of this model, which we refer to as MalConv, is given in 1, and a more detailed diagram is in the supplemental material.

Our architectural choices were influenced in large part by the need to address the high amount of positional variation present in executable files. At a high level, the contents of a PE binary can be rearranged in almost any arbitrary ordering. The only fixed constant is the MS-DOS header, which ends with a pointer to the beginning of the PE-Header. The PE-Header can then be anywhere, and parts of it can be located throughout the file. The PE-Header itself contains pointers to all other contents of the binary (code, resources, etc). This allows a macro-reorganization of the byte contents without ever changing the meaning. Similarly, even within the code sections of a binary, the definition of functions can be re-ordered so long as address of sets used in the code are correctly adjusted. This is another level of spatial restructuring that can occur. This macro-level reordering represents one of many types of spatial properties within a binary, but we consider it to be the most important to tackle. Spatial discontinuities at a function level will remain difficult, but are not insurmountable for the model to learn around. Correlations across large ranges will likely be missed; we hope to capture that information in future work.

To best capture such high level location invariance, we choose to use a convolution network architecture. Combining the convolutional activations with a global max-pooling before going to fully connected layers allows our model to produce its activation regardless of the location of the detected features. Rather than perform convolutions on the raw byte values (i.e., using a scaled version of a byte's value from 0 to 255), we use an embedding layer to map each byte to a fixed length (but learned) feature vector. We avoid the raw byte value as it implies an interpretation that certain byte values are intrinsically "closer" to eachother than other byte values, which we know a priori to be false, as byte value meaning is dependent on context. Training the embedding jointly with the convolution allows even our shallow network to activate for a wider breadth of input patterns. This also gives it a degree of robustness in the face of minor alterations in byte values. Prior work using byte n-grams lack this quality, as they are dependent on exact byte matches (Kolter and Maloof 2006; Raff et al. 2016).

We note a number of difficult design choices that had to be made in developing a neural network architecture for such long input sequences. One of the primary limitations in practice was GPU memory consumption in the first convolution layer. Regardless of convolution size, storing the activations after the first convolution for forward propagation can easily lead to out-of-memory errors during back-propagation. We chose to use large convolutional filters and strides to control the memory used by activations in these early layers.

Attempts to build deep architectures on such long sequences requires aggressive pooling between layers for our data, which results in lopsided memory use. This makes model parallelism in frameworks like Tensorflow difficult to achieve. Instead we chose to create a shallow architecture with a large filter width of 500 bytes combined with an aggressive stride of 500. This allowed us to better balance computational workload in a data-parallel manner using PyTorch (Paszke, Gross, and Chintala 2016). Our convolutional architecture uses the gated convolution approach following Dauphin et al. (2016), with 128 filters.

Regularization A consistent result across tested architectures is a propensity for overfitting. This is not surprising given the large size of our input feature space (2 million time steps) from which we must learn the benign/malicious classification based on a single loss. In particular we note the difficulty in generalizing from both the Group B training data to the Group B testing data, as well as the Group B training data to the Group A test data. In development we found the DeCov regularization (Cogswell et al. 2016) to be most helpful, which penalizes correlation between the hidden state activations at the penultimate layer.

One of the significant challenges in our work was the discovery that batch-normalization was preventing our models from learning the problem. Batch Normalization has become a common tool in the deep learning literature for both faster convergence and a regularizing effect that often improves generalization (Ioffe and Szegedy 2015). This makes the failure of batch-norm on our data an interesting and unique result, which we discuss in 5.3.

4.1 On Failed Architectures

A large number of alternative architecture designs were tested for this problem, including up to 13 layers of convolution, using various (Bidirectional) RNNs, and with different attention models. The MalConv architecture presented performed best amongst many candidates. We review the other high level alternative architecture strategies here, the reasons why they failed to outperform our simpler MalConv, and how these relate back to out final design. Additional details can be found in the appendix.

Adding more layers is possible at the cost of decreased batch size, due to the aforementioned large memory use for backpropagation. We tested this with up to 13 layers of convolutions, and found performance only decreased. Many of these experiments tried smaller convolutional fields, so that the total receptive field of a neuron was on the scale of 500 to 1000 time steps. The problem with these approaches, beyond increasing training time to an untenable degree, is that it is not possible to due the standard approach of doubling the number of convolutional filters after each round of pooling to keep the amount of state per layer roughly equivalent. The state of the convolutions after 2 million steps is simply too large to reasonably compute on. Thus a rapid compression of state size per layer is necessary, but this ends up inhibiting learning. In our approach we have moved large amounts of information into the wide filter width in a single convolution, allowing us to exercise and retain information without exploding memory use.

Another design choice was in processing the entire file simultaneously in one large convolution. An appealing notion would be to break up the input into chunks of 500 to 10,000

bytes, and process each chunk independently, as this would greatly reduce the training requirements. We tested this approach, and while it achieved reasonable accuracies up to 95%, it often failed to generalize to new data — obtaining test accuracies in the 65-80% range. This is because much of the contents of a given binary may be fully non-informative to a maliciousness decision, and training on random chunks and assuming a malicious label then encourages the model to overfit to the training data, and memorize the contents to produce correct decisions. Our MalConv model has access to the entire file which allows the model to detect the few informative features regardless of location. This is necessary to avoid the above variety of overfitting, and is objectively necessary to work in situations where normally benign programs have had malware injected into them. In this common situation most of the file should correctly indicate a benign program, while only a small fraction of the content is malicious.

The issue of information sparsity is also a factor in our choice to use temporal max-pooling rather than average-pooling. Beyond providing better interpretability, max-pooling also provided superior performance relative to average-pooling. The latter enforces a prior that informative features should be widely occurring in the underlying file. But many features will occur only once in the file, and so when combined with average-pooling, that feature's high response in one region of the binary will be washed-out by the remaining majority of the file that produces a low activation. Max-pooling avoids this problem, while still allowing us to tackle the variable-length issue.

While RNNs are a common tool for any sequence related task, we found they reduced test accuracy when applied after our convolutions, by breaking the output after each convolution into a number of fixed sized chunks (with the last chunk containing padding). While an intuitive step to take, this also imposes a prior into the model that data coming from the convolution must regularly produce the same activation patterns at fixed frequencies. This is because the input to the RNN is re-shaping the temporal outputs of the CNN into a non-temporal matrix multiplication, and thus mandates the temporal information appear in consistent locations with a period equal to whatever chunk size was determined. This is not something the CNN can reasonable learn, and so performance is reduced.

5 Results

We now present the results of our neural network model. To evaluate its performance and effectiveness, we will look at standard measures of accuracy in 5.1, investigate the generalization capability of the learned features in 5.2, and address batch-normalization issues in 5.3. We will also take a moment to note the computational constraints required to build this model. To get the model to converge in a timely manner, we had to use a relatively larger batch size of 256 samples per batch. Due to the extreme memory use of the architecture, this could not be performed on a single GPU. We were able to train this model on the 400k Group B set using data parallelism across the 8 GPUs of a DGX-1 in 16.75 hours per epoch, for 10 epochs, and using all available GPU memory. Training on the larger 2 million set took one month on the same system.

5.1 Malware classification

In evaluating the predictive performance of our models, we use Balanced Accuracy (Brodersen et al. 2010) (i.e., accuracy weighted so that benign and malicious samples count evenly) and AUC (Bradley 1997) . We use balanced accuracy so that our results across the Group A and Group B tests sets are directly comparable, as they have differing proportions of benign and malicious samples. AUC is an especially pertinent metric due to the need to perform malware triage, where a queue of binaries to look at is created based on a priority structure (Jang, Brumley, and Venkataraman 2011) . It is desirable to have the most malicious files ranked highest in the queue, so that they are identified and quarantined sooner rather than later. An analyst's time is expensive, and characterizing a single binary can take in excess of 10 hours (Mohaisen and Alrawi 2013). A high AUC score corresponds to a successful ranking of most malware above most goodware, making it a directly applicable metric to evaluate. We pay particular attention to the accuracy on the Group A test set, as it has the fewest correlations with the Group B training set. Thus accuracy performance on Group A serves as a stronger measure of *generalization* performance. In this vein we are also interested in which models have the smallest difference in performance between Groups A and B, which would indicate a model hasn't overfit to the source distribution.

Despite the difficulty of the task at hand, we found that our networks tend to converge quickly, after only three epochs through the training corpus. This is in some ways beneficial, as the training time per epoch is significant. We believe this fast convergence may be due in part to the small size of our architecture, which has (only!) 134,632 trainable parameters. The accuracy results are shown in 1. Our model is able to achieve high AUCs when trained with and without regularization, indicating they would be useful for malware triage to help ranking of work queues.

Looking at the results, we can see our MalConv model is best or second best in performance on both metrics and test-sets. It also has the smallest performance difference between Group A and B test sets, indicating the model is using features that generalize well across the distributions. The byte n-gram model has high accuracy and AUC on the Group B test set, but the model also has a wide gap between Group A and B performance, indicating overfitting (Raff et al. 2016). The byte n-gram model is also fragile to single-byte changes in the input, which will cause a feature to effectively "disappear" form the model's consideration. This is important when we consider that malware is written by an adversary capable of effecting such changes, making byte n-gramming a suboptimal approach. Our MalConv architecture does not have this same issue, and would require considerably more work to circumvent. Using a model trained on the PE-Header generalized well to the Group A test data, achieving a slightly higher accuracy than MalConv, but has significantly reduced performance on Group B in terms of accuracy and AUC. This shows some robustness, but in-

Table 1: Performance of models on Group A and Group B test sets. Best results in **bold**, second best in *italics*.

	MalConv		MalConv w/o DeCov		Byte n-grams		PE-Header Network	
Test Set	Accuracy	AUC	Accuracy	AUC	Accuracy	AUC	Accuracy	AUC
Group A	*88.1*	**98.5**	83.3	*98.4*	87.0	*98.4*	**90.8**	97.7
Group B	*89.6*	*95.8*	86.6	94.3	**92.5**	**97.9**	83.7	91.4

Table 2: Performance of models on Group A and Group B test sets, when using new 2 million training corpus. Best results in **bold**

	MalConv		Byte n-grams	
Test Set	Accuracy	AUC	Accuracy	AUC
Group A	**94.0**	**98.1**	82.6	93.4
Group B	90.9	**98.2**	**91.6**	97.0

dicates the same features aren't being used equally across domains. Overall, MalConv provides the most encouraging balance in performance across all data and metrics.

The application of DeCov regularization significantly improves the accuracy of the model for both Group A and B test sets. This is a somewhat unusual property, as it appears that the DeCov's primary impact is to improve the calibration of the decision threshold, rather than the underlying concept learned by the model. This was a problem noted in Raff, Sylvester, and Nicholas (2017) for their PE-header network. Applying DeCov has successfully improved the calibration of the model's output probabilities, increasing the accuracy by up to 4.8 points.

Using a larger corpus of 2 million files, we can also see that the MalConv model improves its performance, increasing Group A and B accuracy by 5.9 and 1.3 points, and Group B AUC by 2.4 points. We have also replicated the byte n-gram model that the original Group B training data used, and found that performance dropped on the Group A test set by 4.4 points for accuracy and 5.0 points for AUC. Group B test performance was also reduced, though not significantly. This highlights the predicted brittleness and propensity for overfitting of byte n-grams for malware detection (Raff et al. 2016). Our MalConv network's improvement with more data highlights its superiority, and that it has greater capacity to tackle this problem than prior domain knowledge free approaches.

5.2 Manual Analysis

Using our architecture design, we are able to perform a modest manual analysis of what the model has learned. We do this by adapting the approached used by Zhou et al. (2016), which produces a *class activation map* (CAM) for each class in the output. We use a global max-pooling layer in our work, rather than the average pooling originally proposed. Doing so produces a naturally sparse activation map which aids interpretability, which we call a sparse-CAM. This is a critical design choice given the extreme sequence length of our binaries, as it would be impractical to examine all 2 million bytes. This sparse-CAM design will return one 500 byte region as "important" for each convolutional filter; since our model uses 128 filters, there are at most 128 regions marked for each binary.

This approach enables us to produce CAM mappings for regions that are indicative of benignness or maliciousness to the learned network. This preference towards benign or malicious is determined by the sign of the produced activation map. Using the PE-file library (Carrera 2007), we can parse most of our binaries into different regions. These regions correspond to different portions of the binary format. For example, there is a PE-Header that specifies the regions of the file. We expect any approach to learn significant information from this region, as it is the most structured and accessible portion of a binary. The PE-Header then identifies which sections of the binary store the executable code (*.text* or *CODE* sections), global variables (*.data*), and others. By determining which region each sparse-CAM occurred in, we can gain insights about what our model is learning. We show the results of this applied to 224 (7 mini-batches) randomly selected binaries from the Group A test set. This allows us to best evaluate the generalized knowledge of the network, and the results are shown in 3.

Previous work building byte n-gram models on this data found that byte n-gram's obtained almost all information from the PE-Header (Raff et al. 2016). Based on the sparse-CAM locations, we find that only 58-61% of information MalConv is using also comes from the PE-Header, indicating a larger diversity of information types are being used. The *.rsrc* section indicates use of the resource directory, where contents like file icons (but also executable code) may be stored. Importantly we also see the *.text* and *CODE* sections activating, indicating that our model is using some amount of executable code as a feature. Similarly, application data found in *.data* and *.rdata* indicates our model may be detecting common structural patterns between binaries.

We note in particular that the *UPX1* section has been indicative of both benign and malicious binaries, as learned by our network. The *UPX1* section indicates the use of packing, specifically the widely used UPX packer (Oberhumer, Molnár, and Reiser 1996). Packing will compress or encrypt most of the binary into a single archive which is extracted at runtime. This makes simple static analysis difficult, and packing is prevalent among malware authors to hinder malware analysis. However, packing alone is not a reliable malware indicator, as many benign applications are also packed (Guo, Ferrie, and Chiueh 2008). The prevalence of packing in malicious executables leads to many models learning a direct (but unhelpful) equivalence between "packed" and "malicious". Our results indicate that

Table 3: Important features as determined by section, as determined by the non-zero regions of the sparse-CAM mapped to the output of PE-file.

Section	Total	PE-Header	.rsrc	.text	UPX1	CODE	.data	.rdata	.reloc
Malicious	26,232	15,871	3,315	2,878	697	615	669	383	214
Benign	19,290	11,183	2,653	2,414	596	505	423	243	77

our model may have avoided such an association. We hope further advances in interpretable models will help us to confirm this behavior, and determine which minute details allow the model to change its inclination.

5.3 The Failure of Batch-Normalization

Our results are seemingly in conflict with what has been reported in numerous other works, since the addition of batch-normalization to MalConv consistently failed to learn after several epochs. At best models trained with batch-norm would obtain 60% training and 50% test accuracies. This phenomena occurred with all architecture design variants. Our surprise at this result lead us to implement this, and other, architectures using batch-normalization in PyTorch, Tensorflow, Chainer, and Theano. Batch-norm failed to converge or generalize in all cases.

To diagnose this problem, we started with the fact that batch-normalization assumes that data should be re-fit to a unit-normal distribution. We then plotted the pre-activation function response of layers in our network along with that of the Gaussian distribution, which can be seen in 2. The figure shows kernel density estimates of the responses from earlier layers in networks trained on images or on binary executables. Networks trained on image data display an approximately Gaussian distribution of activations (smooth and unimodal), while the activation distribution of our network exhibits much greater asperity. Since batch normalization assumes the data to be normalized is normally distributed, this may account for its ineffectiveness in our application. We recommend that any applications of batch-normalization to new problems produce similar such visualizations as a method to diagnose convergence issues.

We hypothesize that batch norm's ineffectiveness in our model is a product of training on binary executables. The majority of contemporary deep learning research, including batch-normalization, has been done in the image and signal processing domains, with natural language a close second. In all of these domains the nature of data is relatively consistent. In contrast, our binary data presents a novel multi-modal nature of the byte values. The same byte value can have drastically different meaning depending on the location, ranging from ASCII text, code, structured data, or even images stored for the icon. Our hypothesis is that this multi-modal nature produces multiple modes of activation, which then violates the primary assumptions of batch-normalization, causing degraded performance.

Our tests in using models trained on random chunks of only 500 to 10,000 bytes of the binary support this hypothesis. When trained on a random sub-region like this, the majority of bytes will be of a single modality when presented,

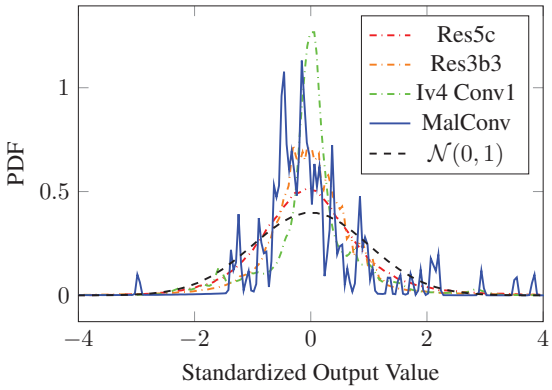

Figure 2: KDE plots of the convolution response (pre-ReLU) for multiple architectures. Red and orange: two layers of ResNet green: Inception-v4 blue: our network; black dashed: a true Gaussian distribution for reference.

and thus present a smoother unimodal activation pattern. This was the only case where batch-norm was able to reach high training accuracies above 60% for our data, but still did not generalize to the test data (obtaining only 50% random-guessing accuracy).

6 Conclusion

In this work we have described the use of neural networks on the raw bytes of entire executable files. This solution avoids a number of the issues with the more common byte n-gram approach, such as brittle features and over-focusing on the PE-Header as important information. It achieves consistent generalization across both test sets, despite the challenges of learning a sequence problem of unprecedented length.

In a broader machine learning context, we have identified a number of unique learning challenges and discussed techniques for addressing classification of extremely long sequences. Our work has extended the application of neural networks to a domain beyond images, speech, etc. to one with much more sophisticated spatial correlation behaviors. In doing so, we identify a potential pitfall with the very commonly used batch-normalization and suggest a way to check if the technique is appropriate (a normality test of pre-activation function response).

In future work we hope to further developed architectures that work in this domain, to further explore the batch-normalization issue, and determine what types of existing normalization or weight initialization schemes work with such multi-modal responses. Critical thought must also be given to ways in which the memory intensive nature of this

problem can be reduced, and what types of architectural designs may allow us to better capture the multiple modes of information represented in a binary. A general approach to byte level understanding of programs would have many applications beyond malware classification such as static performance prediction and automated code generation.

Acknowledgments Special thanks to Mark McLean of the Laboratory for Physical Sciences for supporting this work.

References

Anderson, R.; Barton, C.; Böhme, R.; Clayton, R.; van Eeten, M. J. G.; Levi, M.; Moore, T.; and Savage, S. 2013. Measuring the Cost of Cybercrime. In Böhme, R., ed., *The Economics of Information Security and Privacy*. 265–300.

Anderson, H. 2017. Your model isn't that special: zero to malware model in Not Much Code and where the real work lies. In *BSidesLV*.

Bradley, A. P. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition* 30(7):1145–1159.

Brodersen, K. H.; Ong, C. S.; Stephan, K. E.; and Buhmann, J. M. 2010. The Balanced Accuracy and Its Posterior Distribution. In *Proc. of the 2010 20th International Conference on Pattern Recognition*, ICPR '10, 3121–3124.

Carpenter, M.; Liston, T.; and Skoudis, E. 2007. Hiding Virtualization from Attackers and Malware. *IEEE Security and Privacy* 5(3):62–65.

Carrera, E. 2007. pefile. https://github.com/erocarrera/pefile.

Cogswell, M.; Ahmed, F.; Girshick, R.; Zitnick, L.; and Batra, D. 2016. Reducing Overfitting in Deep Networks by Decorrelating Representations. In *International Conference on Learning Representations*.

Dauphin, Y. N.; Fan, A.; Auli, M.; and Grangier, D. 2016. Language modeling with gated convolutional networks. *arXiv preprint arXiv:1612.08083*.

Garfinkel, T.; Adams, K.; Warfield, A.; and Franklin, J. 2007. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proc. of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, 6:1–6:6.

Graves, A.; Mohamed, A.-R.; and Hinton, G. 2013. Speech recognition with deep recurrent neural networks. In *Proc. of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 6645–6649.

Guo, F.; Ferrie, P.; and Chiueh, T.-C. 2008. A Study of the Packer Problem and Its Solutions. In *Proc. of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, 98–115.

Huang, W., and Stokes, J. W. 2016. MtNet: A Multi-Task Neural Network for Dynamic Malware Classification. In *Proc. of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

Ioffe, S., and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Jang, J.; Brumley, D.; and Venkataraman, S. 2011. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proc. of the 18th ACM Conf. on Computer and Communications Security (CCS)*, 309–320.

Kalchbrenner, N.; Espeholt, L.; Simonyan, K.; Oord, A. v. d.; Graves, A.; and Kavukcuoglu, K. 2016. Neural Machine Translation in Linear Time. *Arxiv*.

Kolosnjaji, B.; Zarras, A.; Webster, G.; and Eckert, C. 2016. Deep Learning for Classification of Malware System Call Sequences. In *The 29th Australasian Joint Conference on Artificial Intelligence*.

Kolter, J. Z., and Maloof, M. A. 2006. Learning to Detect and Classify Malicious Executables in the Wild. *Journal of Machine Learning Research* 7:2721–2744.

Koutnik, J.; Greff, K.; Gomez, F.; and Schmidhuber, J. 2014. A Clockwork RNN. In *Proc. of The 31st International Conference on Machine Learning*, 1863–1871.

Li, B.; Roundy, K.; Gates, C.; and Vorobeychik, Y. 2017. Large-scale identification of malicious singleton files. In *7th ACM Conference on Data and Application Security and Privacy*.

Mehri, S.; Kumar, K.; Gulrajani, I.; Kumar, R.; Jain, S.; Sotelo, J.; Courville, A.; and Bengio, Y. 2017. SampleRNN an unconditional end-to-end neural audio generation model. In *Proc. of the 5th International Conference on Learning Representations*.

Mohaisen, A., and Alrawi, O. 2013. Unveiling Zeus: Automated Classification of Malware Samples. In *Proc. of the 22Nd International Conference on World Wide Web*.

Neil, D.; Pfeiffer, M.; and Liu, S.-C. 2016. Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences. In Lee, D. D.; Sugiyama, M.; Luxburg, U. V.; Guyon, I.; and Garnett, R., eds., *NIPS*. 3882–3890.

Oberhumer, M. F.; Molnár, L.; and Reiser, J. F. 1996. Ultimate packer for eXecutables. https://github.com/upx/upx.

Oord, A. v. d.; Dieleman, S.; Zen, H.; Simonyan, K.; Vinyals, O.; Graves, A.; Kalchbrenner, N.; Senior, A.; and Kavukcuoglu, K. 2016. WaveNet: A generative model for raw audio. In *Proc. of the 9th ISCA Speech Synthesis Workshop*.

Paszke, A.; Gross, S.; and Chintala, S. 2016. PyTorch.

Raff, E.; Zak, R.; Cox, R.; Sylvester, J.; Yacci, P.; Ward, R.; Tracy, A.; McLean, M.; and Nicholas, C. 2016. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*.

Raff, E.; Sylvester, J.; and Nicholas, C. 2017. Learning the PE Header, Malware Detection with Minimal Domain Knowledge. In *Proc. of the 10th ACM Workshop on Artificial Intelligence and Security*, AISec '17, 121–132.

Raffetseder, T.; Kruegel, C.; and Kirda, E. 2007. Detecting System Emulators. In *Proc. of the 10th International Conference on Information Security*, ISC'07.

Roberts, J.-M. 2011. Virus Share.

Rossow, C.; Dietrich, C. J.; Grier, C.; Kreibich, C.; Paxson, V.; Pohlmann, N.; Bos, H.; and Steen, M. v. 2012. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *2012 IEEE Symposium on Security and Privacy*, 65–79.

Saxe, J., and Berlin, K. 2015. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, 11–20.

Schultz, M.; Eskin, E.; Zadok, F.; and Stolfo, S. 2001. Data Mining Methods for Detection of New Malicious Executables. In *Proc. 2001 IEEE Symposium on Security and Privacy. S&P 2001*, 38–49.

Spafford, E. C. 2014. Is anti-virus really dead? *Computers & Security* 44:iv.

Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; and Rabinovich, A. 2015. Going deeper with convolutions. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, 1–9.

Yu, F., and Koltun, V. 2016. Multi-Scale Context Aggregation by Dilated Convolutions. In *ICLR*.

Zhang, X., and LeCun, Y. 2015. Text understanding from scratch. *arXiv preprint arXiv:1502.01710*.

Zhou, B.; Khosla, A.; Lapedriza, A.; Oliva, A.; and Torralba, A. 2016. Learning Deep Features for Discriminative Localization. In *Conference on Computer Vision and Pattern Recognition*, 2921–2929.