What Can N-Grams Learn for Malware Detection?

Richard Zak and Edward Raff
Laboratory for Physical Sciences
Booz Allen Hamilton
{rzak,edraff}@lps.umd.edu

Charles Nicholas
University of Maryland, Baltimore County
nicholas@umbc.edu

Abstract

Recent work has shown that byte n-grams learn mostly low entropy features, such as function imports and strings, which has brought into question whether byte n-grams can learn information corresponding to higher entropy levels, such as binary code. We investigate that hypothesis in this work by performing byte n-gram analysis on only specific sub-sections of the binary file, and compare to results obtained by n-gram analysis on assembly code generated from disassembled binaries. We do this by leveraging the change in model performance and ensembles to glean insights about the data. In doing so we discover that byte n-grams can learn from the code regions, but do not necessarily learn any new information. We also discover that assembly n-grams may not be as effective as previously thought and that disambiguating instructions by their binary opcode, an approach not previously used for malware detection, is critical for model generalization.

1. Introduction

Malware detection is a growing problem due to the amount of new malware each year, which is increasing at an exponential rate. Current anti-virus (AV) technology is based on signatures, which are intrinsically specific to the malware for which they are written. Thus they will not recognize new malware that has not yet been seen, and likely miss known malware that has been slightly tweaked to avoid AV detection. To address this shortcoming, many have looked at applying machine learning algorithms to the task.

The focus of this work is to investigate the performance of one of the simpler machine learning approaches, n-grams. N-grams have been popular for malware detection, used with both byte [e.g., 1]–[3] and assembly [e.g., 4]–[6] features, with some works even using both in a single system [7]. Both byte and assembly n-grams can be extracted purely through static analysis, thereby avoiding many possible environmental pitfalls with dynamic analysis and overfitting [8]. Using static features can be desirable, as it makes the system as a whole simpler and faster, but it also increases the breadth of possible effective obfuscation techniques.

Despite widespread use of byte n-grams, recent work has called into question how effective the technique truly is [9]. Due to overfitting by using binaries from just Microsoft Windows installations, many prior works appear to overfit by learning to separate "Microsoft vs Other" instead of "Benign vs Malicious". In particular, Raff *et al.* [9] question whether byte n-grams could learn beyond the ASCII string and header content of a binary. We evaluate this hypothesis in this work by performing byte n-gramming on different regions of Microsoft Portable Executable (PE) binaries that correspond to different types of content. In particular, we consider the question of whether byte n-grams could learn from the code sections of a binary. For this reason we also evaluate and compare the accuracy of these models against ones created from assembly n-grams.

The contributions of our work are two results that help to further our understanding of what types of information byte and assembly n-grams can learn. First, we provide evidence that byte n-grams are able to learn from the code sections of a binary, contrary to the hypothesis. However, the information learned from the assembly regions seems to be highly correlated with that of the import and header sections of PE binaries, suggesting the actual amount of information learned about the assembly instructions may be limited. Second, we note that the discriminative ability of assembly n-grams is less than that of byte n-grams from the code section. By using multiple datasets that do not share common biases, we observe that assembly n-grams seem to suffer from severe overfitting, by being unable to generalize past data similar to the training distribution. This suggests that the disassembly process and n-gramming strategies commonly used may be losing important discriminative information, and not learning what was previously thought.

The remainder of this paper is organized as follows. In section 2 we review relevant related work. Next we give an overview of the byte n-gram approach in section 3, and detail which sub-regions we use and how we extracted them. In section 4 we will discuss the different ways in which assembly n-grams may be extracted, and introduce an additional novel representation. In section 5 we will review the classification algorithms that will be used in this work and training and parameter tuning procedures. The evaluation methodology and results will be presented in section 6, followed by a discussion of those results and conclusions in section 7 and section 8 respectively.

2. Related Work

Many works have looked at the use of byte n-grams for malware detection, and were considered in the first work on Microsoft Windows malware detection Schultz *et al.* [10]. One of the most thorough previous investigations of byte n-grams was done by Kolter and Maloof [2], who evaluated several different linear and non-linear classification and feature selection algorithms to use with byte n-grams. A number of works follow the same general approach of Kolter and Maloof when using byte n-grams: pick a feature selection method similar to Information Gain on 4 or 6-grams, followed by a non-linear classifier [11]–[13]. Work by Raff *et al.* [9] sought to explore this feature type more deeply, and discovered evidence that prior results were overfitting. We use the model building strategy suggested in their work in ours, using byte 6-grams and elastic-net regularization to perform implicit feature selection with a linear model.

This same general strategy has also been used for assembly (and opcode) based classification. In the static analysis case, the binary is disassembled using some tool, such as IDA Pro, and the extracted assembly features are used to create n-grams. It is often the case that creating n-grams from a whole instruction (with its arguments) is not efficient, and so a number of variants have been used in practice, which we discuss more in section 4. This basic strategy is similar to the byte n-gram approach but at a higher level of representation, and has been popular in practice [14]. Also popular for assembly instructions is to use Hidden Markov Models (HMMs) [15], [16]. The Markov assumption is that one item in a sequence can be predicted with information from only the previous m items. Thus a HMM approach of the m^{th} order has many functional similarities to the n-gram approach explored in this work.

There have been a few works attempting to combine assembly n-grams with byte n-grams and other feature types. These works have all focused on building a system with higher accuracy, where in our work our goal is to determine what kinds of information are being learned or used. Masud *et al.* combined these with function imports into one larger feature set, and found it to perform better than binary of assembly features independently. On one of their datasets, they report accuracies of 96.5%, 94.6% 87.1% when using the combined features, byte, and assembly features respectively. They obtained similar performance for both boosted decision trees and Support Vector Machines. Masud *et al.* 's work is similar to our own in combining features of different types, though it differs in method and reasoning. No analysis was given in their work to determine which type of feature had more or less impact on the improved performance. Similarly, Menahem *et al.* [18] combined a wider array of feature types and classification algorithms into one larger ensemble to maximize performance, but did not attempt to investigate which features contributed in which ways. Though not combining feature types, Yan *et al.* [19] also looked at both byte and assembly n-grams for malware detection. They performed a wide search over feature selection and classification methods to determine which configuration worked best, but used only 300 binaries for their experiments.

The aforementioned works all used static analysis, as we do in this work. It is also possible to obtain assembly instructions via dynamic analysis, which has been popular as well. More closely related to our work, Damodaran *et al.* [16] looked at using assembly instructions to train Hidden Markov Models (HMMs) from both static and dynamic analysis (among other approaches as well). While they did not perform the same type of malware detection, they found dynamic analysis could increase the detection effectiveness, as measured by Area Under the Curve (AUC), by as much as 20 percentage points. They also found that dynamic analysis reduced the number of distinct opcodes observed, which reduces the training time and indicates that many instructions present in a static analysis may not be relevant to functionality.

In conducting the literature review for this work, we did not find any previous malware detection work that uses data similar to Group B (i.e., production data from a corporation) and uses static assembly features[20]–[23]. This is important, as most works use benign data from Microsoft Windows installations, which can result in overfitting to the concept of “Microsoft vs not-Microsoft”. We believe the overfitting that occurs when using Microsoft binaries for training and testing may be the root of positive results with assembly-grams for malware detection that have been previously reported. This gives us some optimism that the issues we discover is not a problem with our data, but that a weakness with assembly-gram features was not published due to a bias against negative results [24], [25]. The lack of publicly available, high quality, datasets for this task will be a hindrance toward reaching a consensus on this issue.

3. Sectional Byte N-Gram Features

The primary effort of this work looks at byte n-grams and what types of information can be learned from them. To investigate this, we perform byte n-gramming on sub-sections of the PE file that correspond to different regions and thus types of data. The PE format specifies a variety of different sections types for storing information needed for the program to execute. Some typical sections found in many PE files are: `.text` for the section of an EXE that contains executable code, `.data` data for initialized variables, `.rodata` data for read-only variables, and `.idata` for the import table.

However, the name for any given section is arbitrary and does not impact how a section is loaded or used. Some compilers put the executable code in a `.code` section instead of the traditional `.text`. This makes the section name an unreliable method of determining type. Instead, one can use the information encoded in each section to more accurately determine a section's purpose. Each section has a number of flag bits which indicate properties of the section, particularly if it is: (1) Executable, (2) Read-only, or (3) Read-write. Newer versions of Windows don't allow a section that is marked executable to be also marked as read-write. To separate out the executable section, all that is needed is to find sections marked as executable. Most files have just one such section, although some have two or more. In the case of UPX-packed binaries, the executable sections are `.UPX0` and `.UPX1`. Using these section flags, and the other fields of the PE header, we perform byte n-gramming on four high level section types: PE-Header, data, imports, and executable code. We obtain the bytes for these four sections in the following manner.

For the parsing of the PE files, we use the PortEx library [26] to discover all sections and the section offsets in the raw file. This library was also used to process the section bit flags needed for the other portions of this work. Once identified, we concatenated all the bytes associated with the PE-Header into one longer sequence. This sequence was then used as the PE-Header feature source for byte n-grams. It generally corresponds to low-entropy information, but is encoded in variable length fields (some fields are single bits, some are multi-bit flags, and some are integers varying between 4 and 64 bits in length). This makes it a good match for byte n-grams in terms of being low entropy, but a poor match in terms of the variable length nature with respect to the fixed size n for n-gramming.

For the executable sections of a binary, we checked every section within the binary for whether or not its executable bit was set. All sections found with such a property were concatenated together. For most files there was only one section marked executable. This corresponds to the theoretical worst case for byte n-grams. The contents of the executable regions are higher entropy, and the encoding of x86 instructions is variable length. Our expectation is that this section will have the worst performance.

The remaining two region types we are interested in are imports and data. In general, the vast majority of sections that were not marked executable corresponded to either a data section or an import section. For benign applications, it is usually easy to separate the two, as the PE-Header will point to the section in which imports are stored. This was not true in all cases for goodware though, and was rarely true for malware. For example, many samples had the import table address point to an address beyond the file size, to areas partway into an existing section, or areas that did not appear to contain any imports at all after manual inspection. If the import table address was present, the address was used as an offset into the section. From the offset to the end of the section was treated as the import table.

To remedy the situation when the import table offset wasn't valid, we used rudimentary string matching to detect regions of the binary that appeared to be containing import information. This was done by comparing the byte content with frequent DLL and function names, and deciding that these sections are import sections. This approach successfully extracted 90% of the imports from the non-executable data, which was verified by manual inspection of randomly selected binaries.

Finally, after identifying the byte sequences corresponding to the PE-Header, executable, and import sections of a binary, all other sections were assumed to be data sections. This gives us all four regions of interest for our experiments. We note that these extractions are not perfect, but are more than accurate enough to allow for informative experiments.

4. Assembly N-Gram Features

Before creating n-grams of assembly instructions, we must first select a subset of base n-gram representations to choose from. In assembly code, each line is generally represented by the instruction name and a number of parameters for the instruction. Just as n-grams are a sliding window of consecutive bytes, we define n-grams of assembly as a sliding window of *lines* of assembly code. A number of options have been proposed, which we will review below.

The relationship between byte and assembly n-grams has been noted before [14], [17]. An important consideration not widely discussed is that not all instructions with the same name map to the same binary opcode[1]. To illustrate, the `cmp` instruction's binary opcode can begin with 0x3C, 0x3D, 0x3A, 0x3B, 0x80, 0x81, 0x83, 0x38, or 0x39, depending on the arguments given. In this and all previous works in malware detection known to us, these are all treated as the same instruction based on the common `cmp` name. We will refer to distinguishing instructions based on their binary opcode as disambiguation. Little work has been done with such disambiguated opcodes in related tasks of malware family classification[27] and function identification[28], but neither work quantifies the importance or significance of using opcodes. We perform the first such comparison to

[1]Some works have used the term "opcode" to describe the instruction name, such as `cmp`. We avoid this terminology, and instead use "opcode" only to refer to the binary encoding of the instruction.

determine the impact of disambiguation assembly n-grams in subsection 6.3, where we will show their impact is critical to obtaining generalizable results.

4.1. Instruction Only

The simplest approach is to capture only the instruction used as the base. If encountering the instruction `mov eax, 4` we simply reduce it to `mov`. This approach is used by Shabtai *et al.* [14]. They argue that this representation will generalize better, as small perturbations in the arguments (due to a change in location) can be functionally equivalent, but no longer found by an n-gram. For brevity, we will refer to this form of assembly n-grams as "OI" for "Only Instructions".

4.2. Instructions with Parameter Type

This method was used by Masud *et al.* [17], and generally appears to be a common preference [29], [30]. They noted that an instruction will have some number of parameters and each parameter is coalesced into a location type, either memory, register, or constant corresponding to where the parameter came from: either an access to memory, directly from a register, or the immediate value from the call to an instruction. For example, the instruction `mov eax, 4` would be coalesced to `mov.register.constant` and `mov [eax], 4` to `mov.memory.constant`. We note that in this form it does not matter that a register was used in the first parameter, it is that the parameter came from a memory accesses that determines the type. We will refer to this form of assembly n-grams as "IPT" for "Instructions with Parameter Type".

4.3. Instructions with Function Resolution

Many works have noted that the APIs called also have predictive power [17], [31], and we have found that byte n-grams tend to pick up on these features as well[9]. Inspired by these observations, we developed a novel feature representation of assembly where all matching constants are replaced with the function name being called, when available from the import table. All other operands are left in their raw form, and we attempt to match exact instruction sequences, with numerical constants replaced by the function name when a match is detected. This is a more viable alternative to using the raw pointer values, as the pointer to a function may change from one binary to the next, even if calling the same function. Our shorthand for this type of assembly n-gram will be "IFR" for "Instructions with Function Resolution". Doing so allows us to perform tests using as much of the raw disassembly as possible, and reducing many instructions to a canonical form. These instructions sequences are logically equivalent between binaries, but would not have been matched correctly if the addresses were not resolved.

```
call 0x401040 ; address not found in import table
call 'MSVCRT.dll:sprintf' ; direct call
mov edi, 'KERNEL32.dll:GetPrivateProfileStringA'
 ↪  ; indirect call to function, first loaded
call edi ; then called via register later
```

Figure 1. Excerpt of our diassembly with function resolution, extraneous instructions removed for brevity.

5. Machine Learning Models

Now that we have reviewed the features that will be used in this work, we will discuss the primary models that we will apply. First, we present our primary classification model which performs feature selection as part of the model construction process. Second, we present the ensemble method we will use to combine models for our byte n-gram results. Creating ensembles is a common method in machine learning to produce a more accurate model by exploiting the uncorrelated errors made by members of the ensemble [32], [33].

5.1. Elastic-Net Regularized Logistic Regression

Keeping our results consistent with [9], we use elastic-net regularized logistic regression [34] for most of our experiments. The objective function of this regression is given in equation (1). The $||w||_1$ term of (1) gives the model the property of performing feature selection automatically as part of the optimal solution. This allows us to consider many different feature set sizes in a computationally efficient process.

$$f(w) = \frac{1}{2}||w||_1 + \frac{1}{4}||w||_2^2 + C\sum_{i=1}^{N}\log(1+\exp(-y\cdot w^\mathsf{T}x_i)) \tag{1}$$

The value C in the loss function is the regularization parameter. Larger values of C decrease the strength of the regularization; as $C \to \infty$, (1) approaches the behavior of standard Logistic Regression. Smaller values of C reduce the effective degrees of freedom of the model, and force coefficients of w to become zero.

An important property of the elastic-net regularizer is robustness to the curse of dimensionality and irrelevant features[35]. The nature of using n-grams is such that, as n increases, the number of unique possible features grows exponentially. Even if all new features provided no information, they can have a significant negative impact on model performance. Using this elastic-net model allows us to reduce this impact.

5.2. Stacking

Given that we want to understand if the information being learned in each section is different, or some variant of the same information, we also apply the Stacking ensemble technique [36] to combine the models from all four byte regions. When performing Stacking, we have a set of base classifiers that make the ensemble, which are all trained independently. The predictions of these classifiers are then used to create a new feature set, of dimension equal to the number of base models used. Any other classifier can be used as the combiner, which uses this new feature set to learn the same problem. The combiner model can be as simple or complex as desired. It is common to use a linear model for the combiner, in which case stacking learns what is essentially a weighted average vote of the constituent base classifiers.

Stacking is often an effective method to increase the predictive performance for a problem, at the cost of using multiple models (and thus more memory and compute time). Though the connection to stacking was not made, the strategy has been applied to malware detection before [37]. Like most ensemble methods, it relies on the base classifiers having some degree of variation and performs best if their errors are uncorrelated. We tested this with a number of combiners, including Random Forests (RF) [38], Linear Support Vector Machines (SVM) [39], and a simple Neural Network (NN). Given this wide array of stacking models, if the errors are uncorrelated and useful, either in a linear or non-linear way, we should see a boost in performance.

6. Evaluation and Results

To evaluate the models and hypothesis of this work, we use the same corpus used in [9], and use the JSAT library for implementation [40]. This data is sub-divided into two primary groups, with Group A collected in a manner consistent with most works on malware detection, and Group B samples provided by an anti-virus company . The data and amounts are summarized in Table 1. The Group A benign data is collected from various versions of Microsoft Windows installations, and was found to be insufficient for training. The common bias of being from Microsoft resulted in strong over-fitting, and likely impacts many prior works[9]. The Group B data is supposed to better represent the general population of benign and malicious binaries found in the wild. For this reason we only perform training on the Group B data, but evaluate the models on all test sets. This helps us better judge the generalization ability of the model.

Table 1. Breakdown of the number of malicious and benign training and testing examples in each data group, along with the sources from which they were collected. "Misc." comprises `portablefreeware.com`, Cygwin and MinGW.

	training		testing	
Group A	malicious	benign	malicious	benign
Virus Share	175,875	—	43,967	—
Open Malware	—	—	81,733	—
MS Windows	—	268,236	—	21,854
Misc.	—	1,195	—	—
total	*175,875*	*269,431*	*125,700*	*21,854*
Group B				
Industry Partner	200,000	200,000	40,000	37,349
total	*200,000*	*200,000*	*40,000*	*37,349*

To evaluate the models discussed, we will use two metrics. The first metric is balanced accuracy [41], which down weights errors of the more populous class so that the benign and malicious samples have equal total weight toward the final accuracy. This avoids comparative issues due to differing levels of class imbalance in the test sets. We use the Area Under the ROC Curve (AUC) [42] as the second metric. The AUC considers the whole spectrum of false negative rates for each possible false positive as the decision threshold is varied.

During feature processing, many binaries could not be disassembled. Many of these errors occurred due to the binary in question having no sections as being marked executable. These binaries ended up being DLL files with translation strings for localized copies of Windows, or DLL files containing icons or other data for applications. Some errors can also occur due to the dissembler erring on challenging inputs. Any file with such an issue was removed from both the training and testing datasets. Since balanced accuracy and AUC are not sensitive to class proportion, we can meaningfully compare those metrics with the results from our byte n-gram experiments. We note that we have confidence in our extracted disassembly, as we are able to resolve addresses across binaries to which function they are calling. These would not resolve or make sensible disassembly if we had an error in our disassembly process. We also removed .Net files, as these files don't contain machine readable code, but rather are interpreted by the .Net or open-source Mono runtime environments.

6.1. Byte 6-Gram Results

We use byte 6-grams for our evaluation as they were found to perform best compares to 4-grams, and larger values are beyond our computational capacity. We compare using byte 6-grams on the entire file as the control, versus byte 6-grams extracted from one of only four sub-regions of the binary. All five models were trained in the same manner as outlined in Raff *et al.* [9], and the results can be found in Table 2. This also includes the results of using Stacking to combine the four different section types with various different combiner models.

Table 2. Accuracy and AUC when using byte 6-grams. First four rows are results using 6-grams from only one section of a PE file, with the fifth row showing result from 6-gramming the whole file. Best numbers in **bold**, second best in *italics*. Last three rows show results when using Stacking (with different models for the combiner) to combine models from all four feature sections.

	Group A		Group B		Open Mal
PE-Section	Acc (%)	AUC (%)	Acc (%)	AUC (%)	Acc (%)
PE Header	76.4	*98.2*	87.7	*95.6*	53.9
Data	69.6	94.9	84.4	92.9	52.7
Import	*83.8*	92.9	*88.7*	94.0	*74.3*
Code	80.5	94.6	88.1	95.2	60.7
Whole File	**87.0**	**98.4**	**92.5**	**97.9**	**81.2**
Stacking SVM	80.8	78.4	87.8	88.4	56.2
Stacking NN	83.1	81.4	89.0	89.8	60.7
Stacking RF	83.6	81.6	89.8	90.4	62.9

We note that in all cases the 6-grams of the entire binary performed best in both accuracy and AUC. For the second best model, the PE-Header was best when using the AUC metric, and the Import section best when considering accuracy. While it may seem unusual for these metrics to differ, this is not an uncommon scenario [43]. The best sub-region for 6-grams region was generally 3 to 5 percentage points behind that of using the whole file. We also note that the near 50% accuracies for Open Malware are not indicative of the model randomly guessing, as the Open Malware set contains only malware. It is likely those models are biased towards declaring a binary as benign, and so would get considerably higher accuracy rates if there were benign files to include with the Open Malware files. This is confirmed by looking at the precision on other datasets. For example, the PE-Header model had a precision of 99.7% and recall of 72.8%.

Regarding the hypothesis that byte n-grams only learn from the import and header sections, our evidence argues against this hypothesis . The 6-gram models were able to learn reasonable models from all four section types, though the models from the Import and PE-Header were the ones that performed best. That these two sections would perform best is not unreasonable, as the lower entropy content of these regions means that an n-gram found in the training set is more likely to be found in the testing set.

What is more interesting is that the Stacking models, which combine the predictions of each of the four section-based models into a final prediction, generally perform worse than models built on only the import and PE-Header sections. Due to how we portioned each binary into the four sections, the entirety of information available to the Stacking model and a byte-gram model trained on the entire binary should be equivalent. Theoretically, the Stacking model has an advantage in intrinsic information about region type from the constituent ensemble members.

The decrease in test accuracy from ensembling suggests that the predictions of the models are highly correlated, as we would expect performance to increase if the predictions were uncorrelated. Differences in prediction output (and confidence) then act only as a noise in the decision process, rather than as signal that helps improve accuracy. We suspect that this means the 6-grams from the code section of a binary are learning the same kind of information that is contained within the Import and Header sections of the binary. This information leakage could potentially be assembly instructions or operands that are correlated with particular functions or settings that may be found in those sections, respectively.

6.2. Assembly N-Gram Results

Given that we have evidence that byte n-grams can learn from the code sections of a binary, we wish to compare and understand the performance differences in what byte n-grams from code learn and what n-gramming of the disassembled code sections. We evaluate the three assembly n-gram strategies discussed in section 4 on the same datasets, using Group B for training. We emphasize that an assembly n-gram does not correlate well with a byte n-gram in terms of how much information is captured in a single features. A single assembly instruction can range in size from one byte to an extreme of 15 bytes, and so we do not concern ourselves with trying to compare bytes vs assembly based on the value of n. For each assembly n-gram type, we evaluate up to and including the largest value of n that we could manage in terms of memory on our workstation[2].

Table 3. Balanced Accuracy and AUC for each test set, with models trained on Group B. Using assembly n-grams of varying types.

Assembly-gram		Group A		Group B		Open Mal
Type	n	Acc (%)	AUC (%)	Acc (%)	AUC (%)	Acc (%)
IFR	1	67.6	44.0	76.5	86.8	30.0
	2	67.9	42.6	78.1	88.5	35.7
IPT	1	64.0	37.8	69.0	78.6	31.6
	2	68.1	44.2	76.8	87.3	36.6
	3	67.1	41.1	76.8	87.8	34.6
	4	64.8	36.0	76.3	87.9	20.9
OI	1	61.6	33.3	64.3	70.3	30.8
	2	65.0	38.4	73.6	85.4	26.5
	3	66.7	40.6	77.8	88.6	28.7
	4	65.5	37.5	77.6	88.3	24.6
	5	64.7	35.9	76.1	87.2	21.5
	6	64.0	34.2	75.5	87.3	19.2

The results for all assembly grams are given in Table 3. The most striking result of these accuracy numbers is a dra-

[2]The workstation used for this experiment has 128GB of RAM

matic drop in performance compared to the byte 6-grams. We also observe considerable overfitting when using the assembly features, where the Group B test set performance is reasonable (yet still lower than the byte grams), and drops in accuracy and especially AUC when evaluated against the other test sets. Our expectation would have been that assembly grams would perform equal to or better than byte grams of assembly, as the disassembled version is a higher level representation of the data. Assembly-grams obtaining an AUC lower than 50% (which would be the threshold of random guessing), combined with the behavior of marking most binaries as benign, indicates the assembly model trained on Group B has no actual generalization to the other datasets.

Given this surprising result, we hypothesize a number of ways in which discriminatory information may have been lost for assembly-grams compared to byte-grams. These stem from differences in assumption between performing byte n-gramming and assembly n-gramming.

First we note, as discussed in subsection 4.1, that different byte op-codes get mapped to the same higher level assembly instruction when performing the disassembly process. It is possible that the specific version of an instruction is in fact discriminative, and is thus lost when using the assembly grams. We will test this first hypothesis in the following section.

Second, we observe that byte n-grams may start and end in the middle of an assembly instruction, where as an assembly n-grams will cover exactly n assembly instructions. This gives byte n-grams an odd form of flexibility and specificity. A byte-gram could start at one instruction, and reach into only the op-code of the next instruction (touching some or none of the operands). Or a byte n-gram could start in the middle of a instruction, considering the lower order bits of the operands of one instruction and then whole or part of the preceding instruction.

6.3. Assembly-Grams with Disambiguation

Existing code infrastructure allows us to test the importance of opcode disambiguation with relative ease. The Capstone Engine API we used for disassembly allows us to obtain the first byte of the opcode for a particular instruction, and so we can produce an "enhanced" disassembly, an example of which can be seen in Figure 2. While the first byte of the opcode may not be sufficient in all cases, it already allows us to distinguish between multiple different versions of the same instruction. We treat these as a new instruction set, and repeat our assembly experiments on the same data.

We note that the disambiguation necessarily increases the size of the feature space. This intrinsically makes learning harder for the algorithm, as the impact of the curse of dimensionality is only increasing. Thus any additional discriminative information from disambiguation must be non-trivial

```
jmp_eb 0x4010eb
push_68 0x10024b78
lea_8d ecx, dword ptr [esp + 4]
call_ff dword ptr [MFC71.DLL:None]
push_53 ebx ; three different pushes
push_56 esi
push_57 edi
push_68 0x10024c05
lea_8d ecx, dword ptr [esp + 0x14]
call_ff dword ptr [MFC71.DLL:None]
lea_8d ecx, dword ptr [esp + 0x24]
mov_bb ebx, 1
push_51 ecx
mov_88 byte ptr [esp + 0x20], bl
call_e8 0x41f8ec
mov_8b edx, dword ptr [eax]
```

Figure 2. Example of disassembly with opcode disambiguation. Note that it is now clear four different `push` instructions are being called, two different `call` instructions, and three different `mov` instructions.

in order to increase performance. This also increases computational burden and memory use, which prevents us from testing n-gram sizes as large as the preceding section.

The results can be seen in Table 4, where the disambiguated opcodes had an almost uniformly positive impact. Most notably, the AUC on the Group A test data improved dramatically, by at least 25 points in every case. A large positive impact was also obtained on the Open Malware test set, and in general for every statistic on every test set when considering 1-grams. This indicates that the specific opcode of the instruction, and not just the instruction type, contains significant discriminative information for malware detection. The disambiguation has improved assembly grams from overfitting to the training data with almost *zero* generalization ability, to being able to show moderate generalization, but still subject to a non-trivial amount of overfitting.

7. Discussion

We have now tested byte n-grams by section type, to help us better understand what byte n-grams have learned. In this process we also tested assembly n-grams as a comparison point to byte n-grams of the executable sections of a binary. In doing so we have discovered a number of interesting results not previously reported, as far as we are aware, for both byte and assembly n-grams.

By byte n-gramming different sections of the binary, we were able to show that they can learn discriminative information from executable regions, which prior work hypothesized was not possible [9]. However, a surprising result is that they do not appear to be learning much about the code contents, but rather, leaked information about imports, strings, and any other lower entropy feature content that was discovered

Table 4. Balanced Accuracy and AUC for each test set, with models trained on Group B. Using assembly n-grams of varying types with partial opcode disambiguation. Difference in scores from Table 3 in parentheses.

Assembly-gram		Group A		Group B		Open Malware
n-gram type	n	Accuracy (%)	AUC (%)	Accuracy (%)	AUC (%)	Accuracy (%)
IFR	1	65.5 (+2.1)	69.2 (+25.2)	78.1 (+01.6)	86.4 (–00.4)	48.4 (+18.4)
	2	69.8 (+1.9)	69.2 (+26.6)	78.8 (+00.7)	85.4 (–03.1)	44.7 (+09.0)
IPT	1	66.1 (+2.1)	73.7 (+35.9)	78.1 (+09.1)	88.4 (+09.8)	46.9 (+15.3)
	2	70.8 (+2.7)	73.7 (+29.5)	80.4 (+03.6)	90.9 (+03.6)	42.9 (+06.3)
	3	66.0 (–1.1)	66.9 (+25.8)	79.6 (+02.8)	90.4 (+02.6)	31.1 (–03.5)
OI	1	61.9 (+0.3)	69.4 (+36.1)	74.4 (+10.1)	84.5 (+14.2)	46.1 (+15.3)
	2	69.7 (+4.7)	72.9 (+34.5)	76.6 (+03.0)	89.9 (+04.5)	43.8 (+17.3)
	3	65.8 (–0.9)	68.3 (+27.7)	80.1 (+02.3)	91.3 (+02.7)	39.2 (+10.5)
	4	64.9 (–0.6)	63.5 (+26.0)	77.6 (+00.0)	89.2 (+00.9)	26.9 (+02.3)

previously. We can draw evidence for this conclusion from the lack of improved accuracy (and in-fact, degraded accuracy), when creating an ensemble of classification models. If the information content used was different, errors should be uncorrelated with byte n-grams from other regions, and thus result in an improved model. In future work we hope to test and better understand the correlations between different types of feature information. One hypothesis as to how this information may be leaked, is that certain imports are strongly correlated with certain code patterns that get reused. This may not be broadly informative about higher level information such as malware author or source language, but are correlated enough with the imports to allow use as a proxy for the import itself.

Ultimately, it seems that byte n-grams are robust in their ability to learn, but weak in what types of information they are able to learn. That is to say, our results seem to confirm that byte n-grams are beholden to using certain types of low entropy information that can already be more easily extracted from the imports and PE-header sections of a binary. But regardless of where byte n-grams are applied, if such information exists or is leaked from other features, it appears byte n-grams will be able to find, extract, and use that information (though with potentially reduced effectiveness).

Somewhat more surprising, and not part of the original goal of this work, was what was learned about the effectiveness of assembly n-grams. It appears that assembly-grams, at least when obtained from static analysis, are uniformly less effective then byte n-grams. This is due to significant overfitting to the Group B data distribution, with a failure to generalize well to the other test sets. We have obtained significantly improved results by incorporating opcode disambiguation, a strategy which we are not aware of any prior works using for malware detection. Even with our improved disambiguated instructions, their performance still lags that of byte n-grams on the executable sections of a binary.

This result is counter to our intuition, as we would believe disassembly to be raising the feature representation up to a higher level, and thus making the job of the learning algorithm easier. Yet our disambiguation results clearly indicate that this may be inadvertently hiding important discriminative information. It is also possible that the higher level representation afforded by assembly grams is in some way enabling greater overfitting to the original data. If so, this could indicate a bias in the Group B data that is highly specific.

As discussed in subsection 6.1, byte n-grams also have the unique property in that they do not care about instruction alignment. It is thus common to have a byte n-gram that starts in one instructions, and ends in another. It is possible that this accounts for the remainder of the performance gap between byte and assembly n-grams, and we hope to explore this in future work. Given that a byte 6-gram can be representing less information than an assembly 1-gram, we suspect that this scenario is a significant component of byte n-gram's learning ability when forced to learn from only the code section of a binary.

Our results are also impacted by the existence of files that could not be disassembled, has happened before [5]. There may also be files with varying portions of erroneous disassembly, as disassembly of malware is not trivial. The difficulties and potential obfuscations that can prevent accurate disassembly is recognized as a challenging problem [44]–[48]. Others have noted there are different methods to performing disassembly that may produce differing results [19], with no one method being necessarily "better" than another. It seems clear from our work that a more thorough investigation of assembly n-grams is warranted, including use of different disassemblers, using disassembly obtained from dynamic analysis (which was not studied in this work), figuring out the best method for handling failure cases, and collective impacts these situations have on malware detection.

It would also be good to further evaluate assembly n-grams in the context of malware family classification. While some have used assembly-grams successfully for this task before [15], [49], a more thorough investigation is warranted. In particular, we hypothesize that assembly-grams may be more effective for family classification then for malware detection. We come to this theory by noting that the Group B test accuracies, which are from the same training distribution, provide reasonable performance. If assembly features processes greater specificity, this may be advantageous in family identification, where the classification labels are intrinsically more specific than the broader benign vs malicious task we have evaluated.

8. Conclusions

Inspired by recent work that questioned the effectiveness of byte n-grams, we have delved further into understanding what types of information they can learn from Microsoft executable binaries. In doing so we make two unexpected conclusions. First, that byte n-grams can learn from higher entropy regions, such as the code section, of a binary — though it may not be learning much information beyond than what is found in the PE-Header and Import sections. Second, that assembly n-grams do not appear to generalize to new data, performing worse than byte n-grams learned from the code regions of a binary. Further, that the standard approach to creating assembly-grams is throwing away useful discriminative information. Combined, these results indicate that byte-grams have more utility than given credit for, and assembly-grams somewhat less.

References

[1] S. J. Stolfo, K. Wang, and W.-J. Li, "Towards Stealthy Malware Detection," in *Malware Detection*, 2007, pp. 231–249.

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

[3] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-gram-based detection of new malicious code," in *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, vol. 2, 2004, pp. 41–42.

[4] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-Sequence-Based Malware Detection," in *Engineering Secure Software and Systems: Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings*, 2010, pp. 35–43.

[5] R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, and Y. Elovici, "Unknown Malcode Detection Using OPCODE Representation," in *Proceedings of the 1st European Conference on Intelligence and Security Informatics*, 2008, pp. 204–215.

[6] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia, "Exploiting similarity between variants to defeat malware," in *Proc. BlackHat DC Conf*, 2007.

[7] R. Perdisci, A. Lanzi, and W. Lee, "McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables," in *2008 Annual Computer Security Applications Conference (ACSAC)*, 2008, pp. 301–310.

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

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

[10] M. Schultz, E. Eskin, F. Zadok, and S. Stolfo, "Data Mining Methods for Detection of New Malicious Executables," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, 2001, pp. 38–49.

[11] S. Jain and Y. K. Meena, "Computer Networks and Intelligent Computing: 5th International Conference on Information Processing, ICIP 2011, Bangalore, India, August 5-7, 2011. Proceedings," in *Computer Networks and Intelligent Computing*, 2011, ch. Byte Level, pp. 51–59.

[12] O. Henchiri and N. Japkowicz, "A Feature Selection and Evaluation Scheme for Computer Virus Detection," in *Proceedings of the Sixth International Conference on Data Mining*, 2006, pp. 891–895.

[13] Y. Elovici, A. Shabtai, R. Moskovitch, G. Tahan, and C. Glezer, "Applying Machine Learning Techniques for Detection of Malicious Code in Network Traffic," in *Proceedings of the 30th Annual German Conference on Advances in Artificial Intelligence*, 2007, pp. 44–50.

[14] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on OpCode patterns," *Security Informatics*, vol. 1, no. 1, pp. 1–22, 2012.

[15] N. Runwal, R. M. Low, and M. Stamp, "Opcode Graph Similarity and Metamorphic Detection," *J. Comput. Virol.*, vol. 8, no. 1-2, pp. 37–52, 2012.

[16] A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, pp. 1–12, 2015.

[17] M. M. Masud, L. Khan, and B. Thuraisingham, "A scalable multi-level feature extraction technique to detect malicious executables," *Information Systems Frontiers*, vol. 10, no. 1, pp. 33–45, 2008.

[18] E. Menahem, A. Shabtai, L. Rokach, and Y. Elovici, "Improving Malware Detection by Applying Multi-inducer Ensemble," *Comput. Stat. Data Anal.*, vol. 53, no. 4, pp. 1483–1494, 2009.

[19] G. Yan, N. Brown, and D. Kong, "Exploring Discriminatory Features for Automated Malware Classification," in *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013, pp. 41–61.

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

[21] W. Mazurczyk and L. Caviglione, "Information Hiding as a Challenge for Malware Detection," *IEEE Security & Privacy*, vol. 13, no. 2, pp. 89–93, 2015.

[22] G. Dahl, J. Stokes, L. Deng, and D. Yu, "Large-Scale Malware Classification Using Random Projections and Neural Networks," in *Proceedings IEEE Conference on Acoustics, Speech, and Signal Processing*, 2013.

[23] N. Karampatziakis, J. W. Stokes, A. Thomas, and M. Marinescu, "Using File Relationships in Malware Classification," in *Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.

[24] R. Rosenthal, "The file drawer problem and tolerance for null results.," *Psychological Bulletin*, vol. 86, no. 3, pp. 638–641, 1979.

[25] D. L. Sackett, "Bias in analytic research," *Journal of Chronic Diseases*, vol. 32, no. 1-2, pp. 51–63, 1979.

[26] K. Hahn, "Robust static analysis of portable executable malware," Master's thesis, HTWK Leipzig, 2014, p. 134.

[27] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "MutantX-S: Scalable Malware Clustering Based on Static Features," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 187–198.

[28] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 845–860.

[29] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based Malware Detection," in *Proceedings of the Second International Conference on Engineering Secure Software and Systems*, 2010, pp. 35–43.

[30] D. Bilar, "Opcodes As Predictor for Malware," *Int. J. Electron. Secur. Digit. Forensic*, vol. 1, no. 2, pp. 156–168, 2007.

[31] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala, "Malware Detection Using Assembly and API Call Sequences," *J. Comput. Virol.*, vol. 7, no. 2, pp. 107–119, 2011.

[32] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[33] ——, "Arcing Classifiers," *The Annals of Statistics*, vol. 26, no. 3, pp. 801–824, 1998.

[34] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society, Series B*, vol. 67, no. 2, pp. 301–320, 2005.

[35] A. Y. Ng, "Feature selection, L1 vs. L2 regularization, and rotational invariance," *Twenty-first international conference on Machine learning - ICML '04*, p. 78, 2004.

[36] D. H. Wolpert, "Stacked generalization," *Neural networks*, vol. 5, pp. 241–259, 1992.

[37] T. Singh, F. Di Troia, V. A. Corrado, T. H. Austin, and M. Stamp, "Support vector machines and malware detection," *Journal of Computer Virology and Hacking Techniques*, pp. 1–10, 2015.

[38] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[39] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan, "A Dual Coordinate Descent Method for Large-scale Linear SVM," in *Proceedings of the 25th international conference on Machine learning - ICML '08*, 2008, pp. 408–415.

[40] E. Raff, "JSAT: Java Statistical Analysis Tool, a Library for Machine Learning," *Journal of Machine Learning Research*, vol. 18, no. 23, pp. 1–5, 2017.

[41] K. H. Brodersen, C. S. Ong, K. E. Stephan, and J. M. Buhmann, "The Balanced Accuracy and Its Posterior Distribution," in *Proceedings of the 2010 20th International Conference on Pattern Recognition*, 2010, pp. 3121–3124.

[42] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[43] C. Cortes and M. Mohri, "AUC Optimization vs. Error Rate Minimization," in *Advances in Neural Information Processing Systems 16*, 2004, pp. 313–320.

[44] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communication security - CCS '03*, 2003, p. 290.

[45] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, 2002.

[46] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," The University of Auckland, Tech. Rep., 1997.

[47] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, 2004, p. 18.

[48] N. Karampatziakis, "Static Analysis of Binary Executables Using Structural SVMs," in *Proceedings of the 23rd International Conference on Neural Information Processing Systems*, 2010, pp. 1063–1071.

[49] Y. Ye, T. Li, Y. Chen, and Q. Jiang, "Automatic Malware Categorization Using Cluster Ensemble," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010, pp. 95–104.