Malware Classification Using Static Analysis Based Features

Mehadi Hassen
School of Computing
Florida Institute of Technology
Melbourne, Florida, USA - 32901
Email: mhassen2005@my.fit.edu

Marco M. Carvalho
School of Computing
Florida Institute of Technology
Melbourne, Florida, USA - 32901
Email: mcarvalho@fit.edu

Philip K. Chan
School of Computing
Florida Institute of Technology
Melbourne, Florida, USA - 32901
Email: pkc@fit.edu

***Abstract*—Anti-virus vendors receive hundreds of thousands of malware to be analysed each day. Some are new malware while others are variations or evolutions of existing malware. Because analyzing each malware sample by hand is impossible, automated techniques to analyse and categorize incoming samples are needed. In this work, we explore various machine learning features extracted from malware samples through static analysis for classification of malware binaries into already known malware families. We present a new feature based on control statement shingling that has a comparable accuracy to ordinary opcode n-gram based features while requiring smaller dimensions. This, in turn, results in a shorter training time.**

I. Introduction

According to AV-Test [1] over 300,000,000 malware were registered in 2014 alone, more than half of which were variations of already existing malware. The sheer volume of the problem makes it impossible for a human expert to analyze each malware. Hence, automated classification of malware samples is needed.

In recent years various research projects [8], [10], [11] have focused on developing classification or clustering techniques to automatically categorize malware into malware families. However, there are many challenges, such as scalability and resilience to code obfuscation techniques, faced by these systems.

In our work we explore the use of different features, extracted using static analysis, for classifying malware into different families. We also propose a new feature based on control statement shingling which has comparable classification accuracy with ordinary opcode n-gram based features. These new features also result in shorter training time.

Before presenting our work, we review past researches in this area in Section II. Background about the machine learning algorithm used in this work is presented in Section III. In Sections IV and V we discuss our solution, and we present our experimental results in Section VI.

II. Related Work

Malware Classification can refer either to classification of binaries as malicious or benign, or classification of malware samples into different known malware families. Our work deals with the later one. However, for the sake of completeness we will look at past research in both.

In [15], the authors evaluate the effectiveness of different data mining techniques in classifying new unseen binaries as malicious or benign. They extract features based on DLL imports, printable strings and program byte sequence. Then they use Naive Bayes and RIPPER, a rule learning algorithm, to learn models that can be used for classification of new samples. Other works, such as [12], also use program printable string feature in addition to function length features to training machine learning models.

The problem with using program byte sequence feature, as done in [15], is that it is easy to obfuscate program byte sequence using various binary packing techniques such as UPX [4]. One way to address this is to examine the opcode or instruction sequence after disassembling the binary program. X. Hu et.al. [11] use opcode sequence, n-gram, features for clustering malware samples.

To get a better understanding of the semantics of a program, other works such as [13], [18], [10], [9] investigate function call graphs. A call graph provides an abstract representation of a program in the form a directed graph. In this graph, the vertices correspond to functions, and the edges correspond to one or more calls made from one function to another [14].

Hu et. al. [10] implement a database management system, SMIT, to support malware lookups. In SMIT, a malware sample is represented using a call graph. SMIT builds a two-level indexing scheme. The first level uses a B+ tree where the keys are the total number of instructions, total number of control instructions, total number of functions, and the median number of instructions per function. The second level index is an Optimistic Vantage Point Tree(VPT), which takes the similarity of the malware call graphs into account when placing malware into storage buckets. The similarity measure used here is an approximate graph edit distance on the malware sample call graphs.

In SMIT, each function, vertex of the call graph, is represented in terms of the function name, assembly instruction mnemonics sequence, and a CRC of the instruction sequence to speed up exact matching between functions. Local, statically linked and dynamically linked functions are considered. When matching two vertices from two call graphs, a match based on function name is used if the functions are statically linked or dynamically linked. In case of local function, CRC is first used for exact matching; if that fails, then sequence edit distance is used as a measure of the similarity between the two functions'

978-1-5386-2726-6/17/$31.00 ©2017 IEEE

instruction sequences.

Similar to SMIT, [13] also uses graph edit distance as a measure of similarity between two malware call graphs. Vertex insertion/deletion, unpreserved edge and vertex relabelling cost are considered when calculating approximate edit distance. Unlike SMIT, where vertices corresponding to local functions are matched based on their instruction sequence when considering relabelling cost, they consider matching of external functions only when calculating vertex relabeling cost. They then apply Kmeans and DBSCAN clustering algorithms to cluster similar malware samples together.

An alternative call graph similarity metric, that is based on the number of matching edges, is proposed in [18] for discovering similar malware variants. Vertices of a call graph, which represent functions, are first matched. Similar to the previous two works, vertices representing external functions are matched based on name; however, local functions are matched based on the external functions they call, the cosine similarity of their instruction frequency, or neighboring matching functions. Once the vertices are matched then matching edges are identified and a similarity metric of two call graphs is computed.

Call graph based techniques incur performance overhead due to graph comparison operations. For instance, k nearest neighbour query in SMIT, with k equal to 5, takes more than 100sec. Considering the large number of malware that are released each day, these techniques face difficulty scaling. There are been research efforts to address this. Hassen et al.[9], for instance, try to address this by proposing a technique based on Locality Sensitive Hashing to create vector representations of function call graphs.

III. BACKGROUND

Random Forest is an ensemble learning technique which uses decision trees as the base classifiers. In general, ensemble learning techniques improve prediction accuracy by combining predictions from multiple classifiers. The individual classifiers can be built by [17]:

- Manipulating the training set. For example, bagging, boosting, random forest.
- Manipulating the input feature. For example, random forest.
- Manipulating class labels.
- Manipulating the learning algorithm. For example, varying the initial weights in neural networks.

Random Forest algorithm manipulates both the training set and input features. The general algorithm for random forest is shown in Algorithm 1. The algorithm builds T decision trees. For each tree, training samples are randomly selected, based on some distribution, from the original training dataset. The new training set D_t is equal in size to the original D. An unpruned decision tree is then trained on D_t. When building the individual trees, instead of considering all available features for splitting at each tree node, only F randomly selected features are considered. Finally, during classification, a majority vote is taken among all the trees.

There are various advantages to using Random Forests for the dataset and features explored here.

1) Random Forest tend to perform better when the set of feature to select from is large. This allows the construction of de-correlated individual trees, which in turn, improves the prediction performance of random forests. In our case for instance, opcode 4-gram feature after hash trick with 14-bit hash has a feature vector size of $2^{14} = 16384$.
2) Speed of training is fast because at any given tree node, random forest only considers a randomly selected subset of features that are much smaller in number than our entire feature space. Hence, reducing the training time.
3) Prediction accuracy was better than or comparable to other techniques we looked at for Microsoft Malware Classification Challenge Dataset, such as logistic regression, backpropagation artificial neural networks, and decision tree.

Algorithm 1: General Random Forest Algorithm

Input : D: Training data,
T: Number of decision trees in the random forest,
F: Number of randomly selected features considered at each tree node.
Output: Random Forest with T decision trees base classifiers.

1 **for** $t = 1$ *to* T **do**
2 - Create bootstrap sample D_t data from original training data D by randomly selecting elements from D where $|D_t|$ = $|D|$.
3 - Train an unpruned decision tree on D_t. At each tree node consider only F randomly selected features for splitting.

IV. SYSTEM OVERVIEW

Our malware classification pipeline consists of three components:

- Feature extraction
- Learning Classification Model
- Classification of Unseen Binaries

Feature extraction, discussed in detail in Section V, is the first component in the pipeline. Here, static analysis of a disassembled malicious binary is performed to extract different features that will be used by the machine learning algorithm to learn models to be used for classification of malicious binaries.

In our experiments, Weka's implementation of Random Forest is used both to learn the classification models and classifying samples. Weka [5] is an open source implementation of a collection of machine learning algorithms. The algorithms can be accessed from the command line tool, GUI interface, or from a Java code.

When classifying new samples into the known malware families, we first extract features. Then, we use Weka by giving

the trained model and feature vectors of the new samples as input.

V. Feature Extraction

The features discussed here are all extracted using static analysis of the disassembled malicious binaries. Static analysis is not the only way to perform analysis of malicious binaries. In dynamic analysis, a malicious binary is run in a sandbox environment while monitoring different aspects of its execution, such as system call, file, network, and registry activities. Despite all the advantages of dynamic analysis, its main shortcoming is its performance overhead. When considering large datasets with thousands of binaries, dynamic analysis does not scale well. Since scalability is one of the requirements of the current work, static analysis is used to extract features from disassembled files of malicious binaries.

To extract features discussed in the next sections, malware binaries need to be unpacked, if packing tools were used to obfuscate the binaries, and then disassembled using unpacking tools.

In the next subsections, we will discuss features that have been used in the past, such as Instruction frequency, opcode n-grams, DLL features, as well as a new control statement shingling based feature proposed in this work.

A. Assembly Instruction Mnemonics Frequency

The first feature considered is instruction mnemonics frequency. Each malware sample is represented as a vector, where the elements of the vector represent frequency of occurrence of an assembly instruction in that specific malware. The frequency values are then normalized by the total number of instructions in that malware.

B. Instruction Opcode N-Grams

The second set of features extracted from disassembled files of malicious binaries consider instruction opcode n-gram frequencies. We decided to use opcode n-grams instead of instruction mnemonic n-grams because opcodes are more specific, hence, providing more discriminating features. For example, there are several opcode values that represent an instruction mnemonic mov based the operand's location and type.

When extracting opcode n-grams, first a malware file is represented as a sequence of instruction opcodes. Then, n-grams of the instruction opcode sequence are created, and their frequencies are counted and normalized by the total number of opcode n-grams in the malware binary.

Ideally, we would want to represent the occurrence frequency of each possible opcode n-gram as a value in the feature vector. However, there are two challenges associated with that:

- The number of unique n-grams grows exponentially in the number of distinct instructions and the length of n-grams. This impacts both on the learning speed and prediction performance of the machine learning model.
- Sparseness of observed n-grams. In practice, not all of the possible opcode n-grams are observed in a given malware. This results in many opcode n-gram frequencies having zero values.

One way, to address the these problems is to use feature reduction techniques to reduce the feature space into lower dimensions. One such technique is using hashing. Previous works, both in the malware classification domain [11] and other domains [7], have used feature hashing for dimensionality reduction and have shown that it does not result in loss of classification performance, given a sufficiently long hashing bit. In fact, dimensionality reduction tends to reduce overfitting [16].

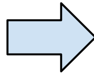
The hashing used here works as follows: Opcode sequences are extracted from a disassembled malware binary, and a sliding window, of size n, is moved over the sequence to get the n-grams. In the example shown in Figure 1, a sliding window of size 2 to get 2-grams, such as 2B-8B, 8B-B8, B8-2B, etc., are extracted. Then a non-crypto hashing function, which uniformly distributes the keys, is used to hash the opcode n-grams into buckets. The feature vector has one feature for each bucket, where each feature value corresponds to the observed frequency of opcode n-grams hashed to that bucket. In our example, for instance, the 2-gram 2B-8B is observed twice so the frequency vector cell that corresponds to the bucket it hashes to has a count of two. This frequency vector is further normalized by scaling its values to have the same range, say $[-1, 1]$ or $[0, 1]$. In the experimental results section the effect of different number of buckets, which is the result of the bit length of hashing, is discussed in more detail.

C. Proposed Opcode N-Grams with Control Statement Shingling

Naturally, code is structured into functions, control blocks, and so on. However, the opcode n-gram feature discussed in the previous subsection does not take the structure of code into account. In an attempt to address this, we borrow the idea of word shingling from text document classification, where stop-words are considered as delimiters in the document ,and n-grams of characters, words, or phrases are constructed starting at the delimiter. In the case of disassembled malware code, we consider the opcode of control statements, such as JMP, LOOP, CALL, as stop-words. These control statements delimit the different control blocks in the opcode sequence.

Properly representing control blocks in the extracted feature presents a new challenge, such as how many n-grams to consider to represent each block and whether or not to include the stop-words(control statements) when constructing n-grams. Both of these impact the classification accuracy and feature dimensionality, which in turn affects model training speed. Considering stop-words as part of the n-gram, for instance, results in fewer numbers of unique n-grams, and hence smaller numbers of features. Similarly the smaller number of n-grams considered in each control block also results in fewer unique n-grams.

Giving dimensionality reduction more consideration, we decided to consider only one n-gram, including the stop-words, from each control block. We start by extracting opcode sequences from the disassembled binary. The shingles are



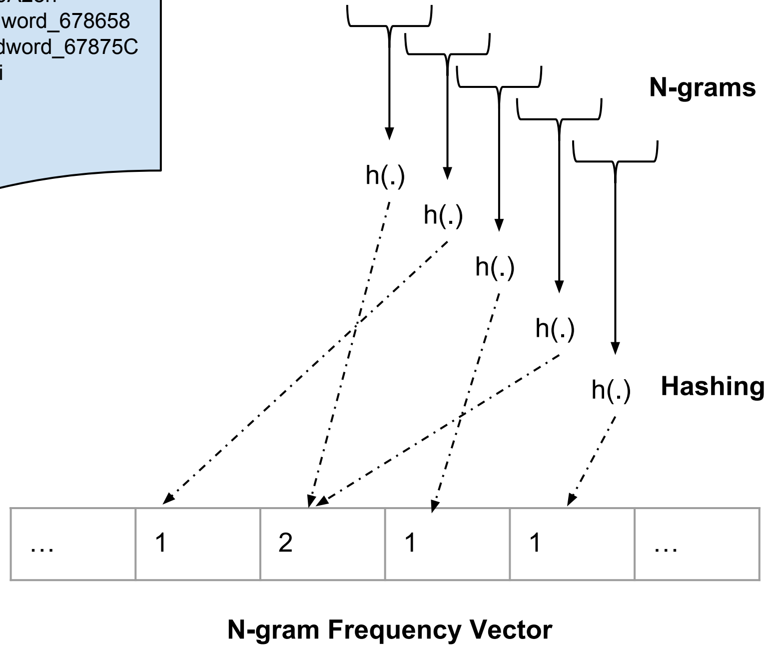


Fig. 1. Extracting opcode n-grams and hashing to reduce dimensionality

then constructed as opcode n-grams starting with a stop-word. For example, in Figure 2 JNZ is a stop-word and n-gram is extracted including JNZ. We then use feature hashing to reduce dimensionality of our feature vector. The feature vector after hashing presents the frequency of n-gram shingles in the corresponding bucket.

Besides taking the structure of the code into account, these features have the added advantage of resulting in a fewer number of unique opcode n-grams. This translates to a smaller number of hashing collisions and hence, requires fewer hash bits, resulting in a lower dimensional feature vector, as shown in Section VI-B.

D. Import Address Table(DLL) Feature

Features discussed so far depend on a successful disassembling of the malware binaries. However, this is not always the case, as some advanced malware packers out there make it very difficult to unpack and disassemble the malware binaries. In such cases, we need to look for other features that can be obtained without the need for disassembling. On such feature is the import table information. The import address table is one of the sections of a PE file. It contains information about external libraries and the function in those libraries that are imported by a program [3].

As a machine learning feature, first all the imported libraries(DLLs) in all of the malware samples in the training set are listed. Once this is done, then a binary vector is constructed for them. A cell in this vector is 1 if that library is imported by the malware sample under consideration; it is 0 otherwise. Each malware sample will be represented by this binary vector.

TABLE I. DATASET CLASS DISTRIBUTION

Malware Family Name	Number of Samples
Ramnit	1541
Lollipop	2478
Kelihos ver3	2942
Vundo	474
Simda	42
Tracur	751
Kelihos˙ver1	398
Obfuscator.ACY	1228
Gatak	1013

VI. EXPERIMENTS AND RESULTS

A. Dataset

The dataset used in this paper was obtained from the the Microsoft Malware Classification Challenge [6]. It has 10,867 labelled malware samples, which belong to 9 malware families. Table I shows the class distribution for the nine malware families. For each malware sample, the disassembled file, generated using IDA pro [2], and its binary file, without a PE header, are given.

B. Hash bit length when using feature hashing with n-gram features

The number of bits used to represent the hash values determines the total number of buckets to which the keys get

Fig. 2. Extracting opcode n-grams and hashing to reduce dimensionality

hashed. The hash bits greatly affect the performance of the machine learning algorithm. If too small, there will be a lot of collisions and many n-gram features will be hashed into the same bucket, hindering descriptiveness of the reduced feature vector. Large values, on the other hand, fail to reduce the dimension of our original opcode n-gram feature which results in the machine learning algorithm taking longer training time, and being susceptible to overfitting. Therefore, it is important to determine the appropriate hash bit length.

To determine the hash bit length that is appropriate for both the opcode n-gram and opcode n-gram with control statement shingling, we conducted experiments by varying both the length of the n-gram and the hash bit length. For these experiments we used Random Forest with 100 base classifier trees (i.e. the T in Algorithm 1). These experiments were conducted on a smaller subset of the original data, with 946 samples and 10 fold cross validation. The results of these experiments are shown in Figure 3.

For a fixed value of n in n-gram, we expect the prediction accuracy to increase with an increase in the number of bits used to represent the hash value. This is because the larger number of hash bits results in smaller hash collisions. On the other hand, for a fixed hash bit length, we expect the prediction accuracy to decrease as the length of the n-gram increases. This is because the number of possible n-grams increase as n increases, and this will result in higher number of collisions.

The results in Figure 3 agree, for the most parts, with our expectation. For the opcode n-gram features, Figure 3a, 3-gram and 4-gram accuracy increases as the number of hash bits increase. In the case of 2-grams, however, accuracy for the 14-bit hash is less than accuracy at the 12-bit. For 2-gram opcode sequence there are I^2 distinct 2-gram opcode sequences, where I is the number of distinct x86 opcodes. And a 14-bit hash results in a feature vector of size 2^{14}=16384, resulting in $I^2/16384$ collision on average, if the hashing function uniformly distributes the keys across the 2^{14} buckets. However, the original 2-gram features are very sparse, so even though we expected $I^2/16384$ collisions per bucket, most of the 2-gram frequencies have zero values, thus resulting in the hash buckets also having zero values. This is exactly what we found out when we looked at the feature vector for 14-bit hashed 2-grams. When this sparse feature vector is given as input to our algorithm, Random Forest, it is possible that many of the F features that are considered by the algorithm at each tree node are zero and do not help in distinguishing the malware class. Hence, we see that in the case of 2-gram with 14-bit hashing, the prediction accuracy does not improve. One of the results that we haven't been able to explain has to do with the slight decrease in accuracy in case 4-gram when hash bits are increased from 8-bit to 10-bit in Figure 3a. In spite of this decrease, the over all trend of increasing in accuracy with increase in hash-bits is still evident in 4-grams for 12-bit and 14-bit hash.

Opcode n-gram with control statement shingling feature, Figure 3b, also exhibits similar behavior. In this case, the optimal hash length for 2-gram is at 10-bit. It is mainly due to the fact that our proposed features resulted in a smaller number of distinct n-grams. This, in turn, results in a fewer number of hash collisions, achieving good classification accuracy even at

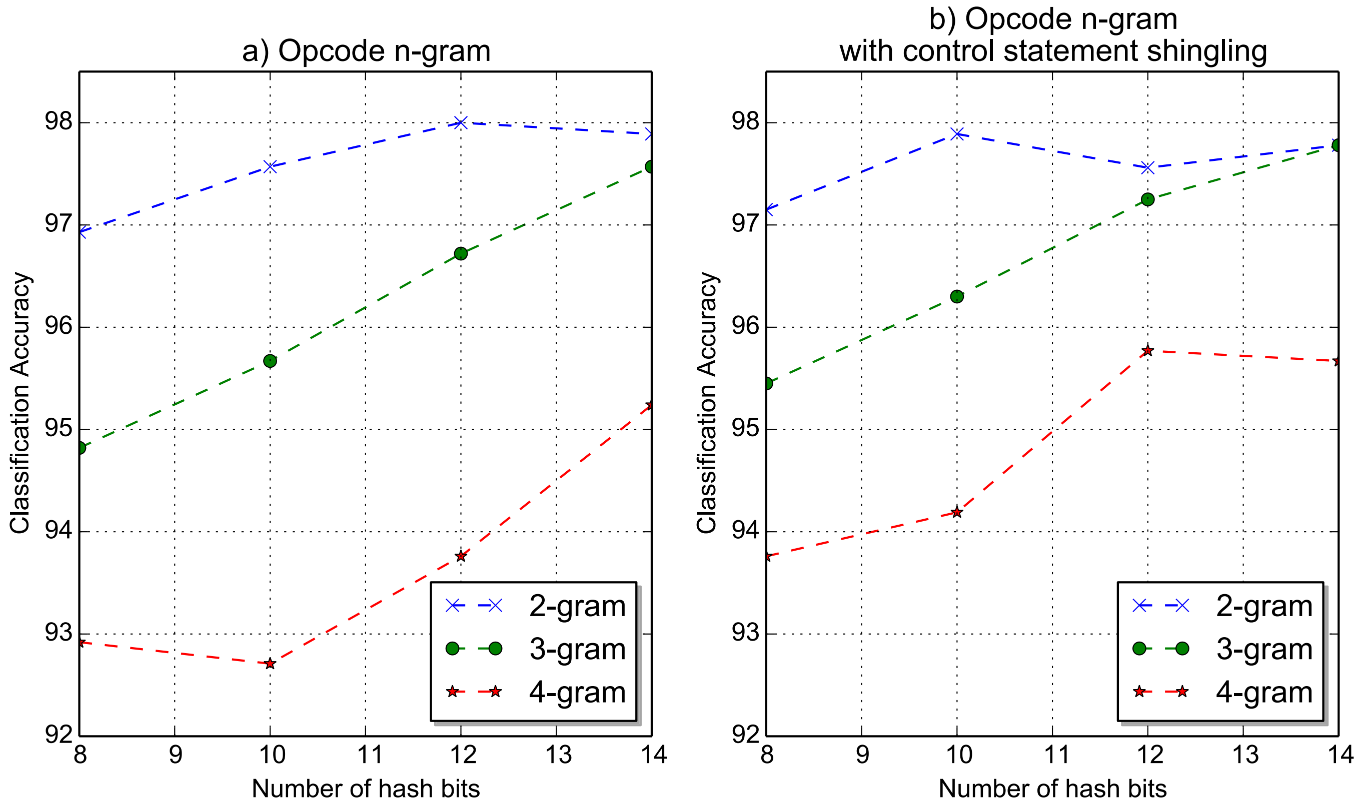


Fig. 3. Effect of hash bit length on prediction accuracy using Random Forest

a smaller hash bit length.

We also conducted experiments to justify our decision to include stop-words (control statements), such as JMP,CALL, etc., when representing each control block in terms of n-grams. The classification accuracy including the stop words using 10-bit feature hashing is 97.89%, whereas, classification accuracy when excluding stop-words using 10-bit feature hashing is lower at 97.04%. The optimal hash bit length when excluding stop-words occurs at 12-bits due to the larger number of unique n-grams. Therefore, not only does including stop-words in the n-grams improve accuracy, it also reduces the number of unique n-grams which, in turn, reduces the feature dimensionality as well as improving model training speed.

C. *Evaluating Features*

We now compare the prediction accuracy of the features discussed in Section V. In addition to the individual features, we also look at some of their combinations. Table II shows the prediction accuracy on the training set using 10 fold cross validation.

The last column in the Table shows the number of data instances considered for each feature. One of the problems we faced when applying n-gram based features to this data set was that not all of the malware samples were properly parsable by our feature extractor. Hence, we see that when using only instruction mnemonics or opcode based features, we are only able to classify 10,260 instances. To take advantage of the high prediction accuracy of n-gram based features and all

TABLE II. PREDICTION ACCURACY USING RANDOM FOREST

Features	Accuracy	Instances
Asm instruction frequency	97.71	10260
Opcode 2-gram	99.21	10260
Opcode 2-gram using control stmt shingling	99.11	10260
DLL boolean feature	83.41	10867
DLL with Opcode 2-gram	98.25	10867
DLL with Opcode 2-gram using control stmt shingling	97.98	10867

inclusiveness of DLL boolean features, we combined the two features. Results are shown in the last two rows with prediction accuracy of 98.25% and 97.98% for opcode 2-gram features with 12-bit hashing and opcode 2-gram feature with 10-bit hashing and control statement shingling, respectively.

The new control statement shingling feature proposed in this paper, opcode n-gram using control statement shingling, performs very well at 99.11% almost equivalent to normal opcode n-gram feature which has accuracy 99.21%. Since this new feature encounters a smaller number of unique n-grams, it has the added advantage that it needs a fewer number of hash bits to represent, hence having a smaller space requirement. In the results shown here, opcode n-gram using control statement shingling is represented using 10-bit hash, compared to the normal opcode n-gram feature which requires 12-bits, hence resulting in almost 4 times as many features.

D. *Training Time*

The smaller number of features that result from our new feature also translates to a faster training time. As shown in

TABLE III. TRAINING TIME

Feature	Training Time (sec)	
	Random Forest	Logistic Regression
12-bit opcode n-gram	1.52	2169.72
10-bit control stmt shingling	1.28	67.03

Table III, this is less evident with Random Forest because it works on a random subset of features when evaluating the the features for each tree node. The average training time for Random Forest using the control statement shingling features with 10-bit hash is 1.28 seconds, whereas using n-gram frequency features with 12-bit hash it was 1.52 seconds.

The training time difference becomes very significant when using machine learning algorithms that consider all features for model training. For example, using Logistic Regression, the average training time with our new feature which only needs 10-bit hash is 67.03 seconds, whereas when using normal n-gram features with 12-bit hash is 2169.72 seconds.

These experiments were carried out on a smaller subset of the original dataset consisting of 946 samples. We used Weka's implementation Random Forest and Logistic Regression ran on a machine with 2.6GHz 8-core cpu and 100GB main memory.

VII. LIMITATIONS

The machine learning features used in this paper rely on the correct disassembly of the malware binary. The use of more advanced binary packers by malware authors can make disassembling difficult. One way to handle this is to use dynamic analysis to run the packed program and dump process memory image once the malware has unpacked itself. This is then given to the disassembler.

The proposed classification technique is for identifying the malware family of a given malware. This requires the malware to belong to, or to be a variation of, one of the already known malware families. Identifying new malware families is beyond the scope of of this work. Clustering techniques can be used for this purpose.

VIII. CONCLUSION

In this work we investigated the problem of classifying malware samples into known malware families. This is an important problem because more than half of the malware released each year are variations of existing malware. By classifying malware samples into families, representative samples of a given family can be analysed by human experts and defensive measures can be taken.

We evaluated various static analysis based features for malware classification. We also proposed a new feature which performs n-gram shingling with control statements as stop-words. We showed that the new feature performs comparably with opcode n-gram feature while requiring smaller feature vector and shorter training time.

REFERENCES

[1] Av-test malware statistics. http://www.av-test.org/en/statistics/malware/.

[2] Ida pro. https://www.hex-rays.com/products/ida.

[3] An in-depth look into the win32 portable executable file format. https://msdn.microsoft.com/en-us/magazine/cc301808.aspx.

[4] Ultimate packer for executables(upx). http://upx.sourceforge.net.

[5] Weka 3. http://www.cs.waikato.ac.nz/ml/weka/.

[6] Microsoft malware classification challenge (big 2015). https://www.kaggle.com/c/malware-classification, 2015. [Online; accessed 27-April-2015].

[7] J. Attenberg, K. Weinberger, A. Dasgupta, A. Smola, and M. Zinkevich. Collaborative email-spam filtering with the hashing trick. CEAS, 2009.

[8] D. H. Chau, C. Nachenberg, J. Wilhelm, A. Wright, and C. Faloutsos. Polonium: Tera-scale graph mining and inference for malware detection. In *SIAM International Conference on Data Mining*, volume 2, 2011.

[9] M. Hassen and P. K. Chan. Scalable function call graph-based malware classification. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 239–248. ACM, 2017.

[10] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM, 2009.

[11] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin. Mutantx-s: Scalable malware clustering based on static features. In *USENIX Annual Technical Conference*, pages 187–198, 2013.

[12] R. Islam, R. Tian, L. Batten, and S. Versteeg. Classification of malware based on string and function feature selection. In *Cybercrime and Trustworthy Computing, Workshop*, page 917. IEEE, 2010.

[13] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[14] B. G. Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, (3):216–226, 1979.

[15] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001.

[16] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[17] P.-N. Tan, M. Steinbach, V. Kumar, et al. *Introduction to data mining*, volume 1. Pearson Addison Wesley Boston, 2006.

[18] M. Xu, L. Wu, S. Qi, J. Xu, H. Zhang, Y. Ren, and N. Zheng. A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques*, 9(1):35–47, 2013.