Drew *et al. EURASIP Journal on Information Security* (2017) 2017:2
DOI 10.1186/s13635-017-0055-6

EURASIP Journal on Information Security

RESEARCH Open Access

Polymorphic malware detection using sequence classification methods and ensembles

BioSTAR 2016 Recommended Submission - EURASIP Journal on Information Security

Jake Drew[1*], Michael Hahsler[2] and Tyler Moore[3]

Abstract

Identifying malicious software executables is made difficult by the constant adaptations introduced by miscreants in order to evade detection by antivirus software. Such changes are akin to mutations in biological sequences. Recently, high-throughput methods for gene sequence classification have been developed by the bioinformatics and computational biology communities. In this paper, we apply methods designed for gene sequencing to detect malware in a manner robust to attacker adaptations. Whereas most gene classification tools are optimized for and restricted to an alphabet of four letters (nucleic acids), we have selected the *Strand* gene sequence classifier for malware classification. Strand's design can easily accommodate unstructured data with any alphabet, including source code or compiled machine code. To demonstrate that gene sequence classification tools are suitable for classifying malware, we apply Strand to approximately 500 GB of malware data provided by the Kaggle Microsoft Malware Classification Challenge (BIG 2015) used for predicting nine classes of polymorphic malware. Experiments show that, with minimal adaptation, the method achieves accuracy levels well above 95% requiring only a fraction of the training times used by the winning team's method.

Keywords: Sequence classification, Minhashing, Polymorphic malware, Strand

1 Introduction

The analogy between information security and biology has long been appreciated, since Cohen coined the term "computer virus" [1]. Modern malware frequently takes the form of a software program that is downloaded and executed by an unsuspecting Internet user. "Infection" can be achieved through compromising many thousands of websites en masse [2], social engineering, or by exploiting vulnerabilities on end-user systems. Regardless of how the infection occurs, cybercriminals have also undertaken considerable efforts to evade detection by antivirus software [3, 4]. Current signature based detection methods are highly sensitive to minor changes within the structure of a malware program. In many cases, a small change within the malware program alters the program's signature sufficiently enough to thwart antivirus detection.

Developers of such polymorphic malware attempt to avoid the detection of their malicious software by constantly changing the program's appearance while keeping the functionality the same. This can be achieved by manipulating the code using multiple forms of obfuscation. Techniques include encryption of malicious payloads, obfuscating variable names using character code shifts, equivalent code replacements, register reassignments, and removal of white space or code minification [5–7]. Individual instances of polymorphic malware can maintain the same general functionality while displaying many unique source code characteristics. For example, the computer worm Agobot or Gaobot was first identified around 2002 [8]. Over 580 variations of this malware were subsequently identified [9]. Today, each malware category can spawn many thousands of mutations, adding up to as much as one million new "signatures" per day [10]. In

*Correspondence: jakemdrew@gmail.com
[1]Darwin Deason Institute for Cyber Security, Southern Methodist University, Dallas, TX, USA
Full list of author information is available at the end of the article

some cases, variations between malware instances occur simply to avoid detection. In others, new functionality emerges over time. Such changes require a robust form of malware classification which is less influenced by generational variation.

Gradual changes in polymorphic malware can be seen as mutations to the code. Thus, these changes are similar to mutation of biological sequences which occur over successive generations.

Recently, significant advances have been made in gene sequence classification in terms of classification accuracy and processing speed. Originally, classification was based on expensive sequence alignment tools like BLAST [11] for comparing sample sequences to other sequences from known taxonomies. Many newer sequence classification tools claim to be faster and/or more accurate. Examples are BLAT [12], the Ribosomal Database Project (RDP) naive Bayes classifier [13], UBLAST/USEARCH [14], Strand [15], Kraken [16], and CLARK [17].

Given the similarities between mutations in malware and in gene sequences, it stands to reason that the tools developed for gene sequence classification hold the potential to be applied to polymorphic malware detection. Consequently, in this paper we set out to apply one such classifier, called Strand (The Super Threaded Reference-Free Alignment-Free Nsequence Decoder) [15], to performing classification of polymorphic malware data. We selected Strand because, unlike the aforementioned gene sequence classifiers, it can process sequences of arbitrary alphabets. While BLAST has been adapted by researchers to process non-biological sequences [18], Strand can be used on general sequences "out of the box" and performs more efficiently than BLAST. We then use Strand to classify the malware dataset used in the Kaggle Microsoft Malware Classification Challenge (BIG 2015) [19]. We show how the application achieves comparable accuracy to the winning team's sophisticated malware classification techniques using only a fraction of the time required to generate the Strand training model. This paper is an expanded version of [20] and includes new feature extraction and ensemble techniques for the Interactive Disassembler Tool (IDA) files provided by Microsoft via Kaggle. We explain how 32-Bit vs. 64-Bit hashing functions influence minhash signature classification accuracy and present new results which are approximately seven times faster than our original results presented in [20].

2 Background

We now give a brief overview of word-based gene sequence classification, which is typically done using word matching. Words are extracted from individual gene sequences and used for similarity estimations between two or more gene sequences [21]. Gene sequence words are sub-sequences of a given length. In addition to words they are often also referred to as k-mers or n-grams, where k and n represent the word length. The general concept of k-mers or words was originally defined as n-grams during 1948 in an information theoretic context [22] as a subsequence of n consecutive symbols. We will use the terms words or k-mers in this paper to refer to n-grams created from a gene sequence or other forms of unstructured input data. Over the past 20 years, numerous methods utilizing words for gene sequence comparison and classification have been presented [21].

Methods like BLAST [11] were developed for searching large sequence databases. Such methods search for seed words first and then expand matches. These so called alignment-free methods [21] are based on gene sequence word counts and have become increasingly popular since the computationally expensive sequence alignment method is avoided. In this paper, we refer to the individual characters (A, C, G, T) within a particular gene sequence as *bases*. The most common method for word extraction uses a sliding window of a fixed size. Once the word length k is defined, the sliding window moves from left to right across the gene sequence data producing each word by capturing k consecutive bases from the sequence.

The RDP classifier [13] uses only eight characters within each gene sequence word during both training and classification processing. This makes the total possible number of unique words (i.e., features for the classifier) only $4^8 = 65,536$ words. Unfortunately, such a small feature space makes distinguishing between many sequence classes challenging.

Rapid abundance estimation and sequence classification tools [16, 17] use longer words and derive a large speed advantage by utilizing, instead of word counts, a simple match between the words extracted from sequence data to identify the similarity between two sequences. However, this approach comes at the cost of storing a very large number of sequence words to make accurate classifications. For example, the extraction of $k = 30$ base words results in $4^{30} \approx 10^{18}$ unique word possibilities within the training data feature space when an alphabet of four symbols (A, C, G, T) is considered.

The issues with the need to store a large number of words becomes even more problematic when the size of the alphabet increases. This is clearly the case when we consider compiled code or source code. Strand addresses this problem by utilizing a form of lossy compression called Minhashing [23] which still supports sequence comparison, but with a much reduced memory footprint.

3 Strand

Next, we give a very short overview of the Strand classification process (see [15] for more details).

Strand uses the map reduction aggregation process shown in Fig. 1 to rapidly prepare and process input data in parallel during training or classification. Map reduction aggregation executes using shared memory during all stages within each Strand worker process. When multiple worker processes are used in a cluster, a single master process combines the outputs from each of the self-contained workers as they complete.

During stage 1 of map reduction aggregation, multiple threads extract words and associated classes from the gene sequence data in parallel. Simultaneously, a stage 2 combiner process minhashes each extracted word eventually creating a minhash signature for each input sequence provided. Finally, the unique minhash keys within each minhash signature are summarized by class during the reduce stage. During training, the reduce step adds minhash values into the training data structure, and during classification, minhash values are looked up within the training data structure and minhash intersections for each class are tabulated to determine one or more class similarity estimates.

3.1 Traditional MapReduce vs. map reduction aggregation

We now compare map reduction aggregation to more traditional MapReduce style processing for the benefit of understanding its advantages. Map reduction aggregation includes a preliminary map stage, any number of required intermediate map or combiner stages, and a reduce stage. In traditional MapReduce, a combiner stage is simply an intermediate or semi-reducer that further processes data prior to the final reduce stage [24]. In Strand, all stages required for map reduction aggregation processing are self-contained within a training or classification worker process which allows each processing stage access to the same shared memory at all times during machine learning. This is highly advantageous when compared to other traditional forms of MapReduce. The traditional MapReduce execution overview is illustrated in Fig. 2.

The following steps comprise the typical MapReduce model [25]:

1. Input data is split into multiple pieces which are managed by a master process.
2. Next, worker processes await either map or reduce tasks provided by the master.
3. Specific operations for both the map and reduce procedures are specified by the user.
4. The master monitors each map task's successful completion and notifies reduce workers of the map file output locations.
5. Intermediate files on local disks are required between each of the map, combiner, and reduce stages executed for traditional MapReduce.
6. When the reduce stage reads in mapped files from disk, the data is also sorted since a large number of keys may map to a single reduce task.
7. The reduce function processes each sorted map item according to the user specified reduce operations writing results to a separate final result file for each reduce task executed.
8. Finally, the master returns control to the calling program once all reduce steps have successfully completed.

Strand's map reduction aggregation methods take advantage of the parallelism constructs afforded by the MapReduce model while avoiding much of the overhead associated with intermediate file disk I/O, sorting, and inter-process communication between the master and worker processes located on different commodity hardware machines.

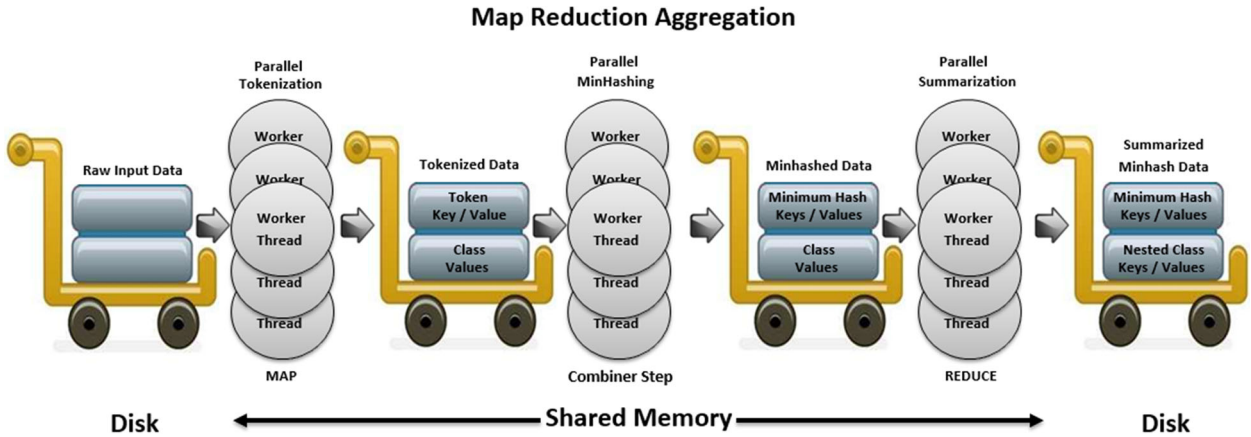

Fig. 1 Strand map reduction aggregation processing for a single training or classification worker process

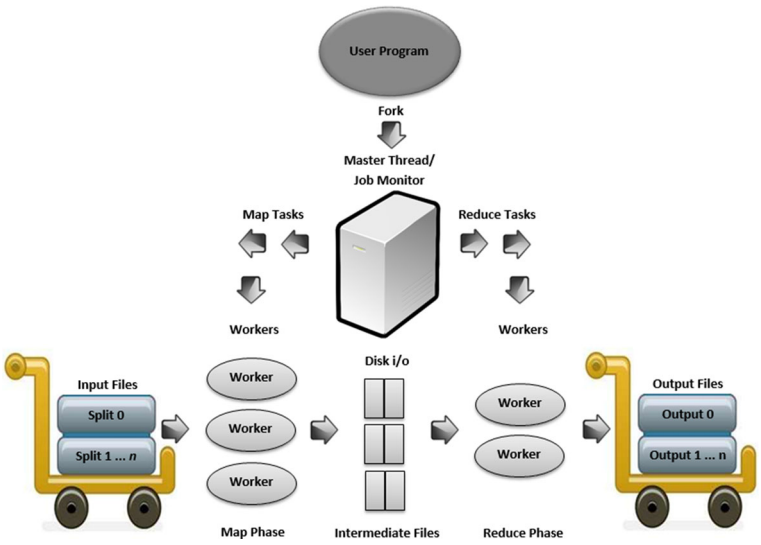

Fig. 2 Traditional mapreduce execution overview. Courtesy of [25]

The map reduction aggregation method specifies how targeted input data will be aggregated within the current system during training and classification worker processing. Input data is consistently dissected by mapping and optional combiner processes into individual, independent units of intermediate work typically comprising consistently mapped gene sequence word keys and class values that are conducive to simultaneous parallel reduction processing. The reduce method continually and simultaneously aggregates the mapped word keys and values by eliminating the matching keys and aggregating values consistent with the specified reduce operations for all matching keys which are encountered during reduce processing.

All map, combiner, and reduce stages are self-contained within a single user specified map reduction aggregation method allowing access to shared memory between all processing stages. The user specified map reduction aggregation method operates within any number of training and classification worker processes to scale as required by the user or machine learning task at hand. Strand training worker processes apply map reduction aggregation to gene sequence input data, reducing the resulting minhash signatures and associated classes into the Strand training data structure. In certain Strand embodiments, class frequencies are maintained for each unique minhash key. During classification, minhash values within each minhash signature resulting from map reduction aggregation are looked up within the Strand training data structure to determine an accurate estimation of Jaccard similarity between the query sequence and all known classes. Minhashing is used as a form of lossy compression to reduce the overall size of the training data structure and decrease the processing time required to estimate the similarity between a query sequence and one or more known classes within the system.

3.2 Minhashing during map reduction aggregation

Minhashing [23] is utilized within Strand to drastically reduce the amount of storage required for high-capacity map reduction aggregation and classification function operations. Map reduction aggregation requires multiple pipeline stages when lossy compression via minhashing is deployed.

In Fig. 1, Strand uses a map reduction aggregation pipeline including an additional combiner step to facilitate minhashing. This process also represents a more accurate method for Jaccard approximation than mere random selection of words. Minhashing is a form of lossy data compression used to remove a majority of the gene sequence words produced during stage one mapping by compressing all words into a much smaller minhash signature.

During stage one of the map reduction aggregation method shown in Fig. 1, transitional sequence word outputs are placed into centralized, thread-safe storage areas accessible to minhash operation workers. In stage 2, a pre-determined number of distinct hashing functions are then used to hash each unique key produced during the stage one map operation one time each. As the transitional keys are repeatedly hashed, only one minimum hash value for each of the distinct hash functions are retained across all keys. When the process is completed, only one minimum hash value for each of the distinct hash functions remains in a vector of minhash values which represent the unique characteristics of the

learning or classification input data within a minhash signature.

To further enhance minhashing performance, only a single hash function can be used to generate a minhash signature. This eliminates the overhead of hashing words multiple times to support the family of multiple hashing functions traditionally used to create a minhash signature. In this scenario, all words are hashed by a single hashing function and n minimum hash values are selected to make up the minhash signature. These minimum values represent a random permutation of all words contained within the target sequence.

The minhash signature is further reduced by storing each minhash value in a partitioned collection of nested categorical key-value pairs. The training data structure illustrated in Fig. 3 is designed in this manner. The training data structure's nested key-value pairs are partitioned or sharded by each distinct hash function used. For example, when the minhashing process uses 100 distinct hash functions to create minhash signatures, the training data structure is divided into 100 partitions. All unique minhash keys created by hash function 0 are stored within partition 0 of the training data structure. Likewise, all unique minhash keys created by hash function 99 are stored in partition 99. However, when only a single hash function is used, no partitions are required.

The partitioned training data structure shown in Fig. 3 includes minimum hash values which act as the *key* in the nested *categorical* key-value pair collection. Each minhash key contains as it value a collection of the classes which are associated with that key in the system. This collection of classes represents the nested categorical key-value pairs collection. Each nested categorical key-value pair contains a known class as its key and an optional frequency, weight, or any other numerical value which represents the importance of the association between a particular class and the minhash value key.

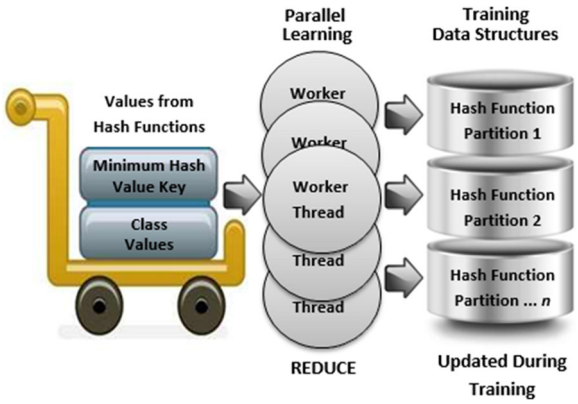

Fig. 3 The strand partitioned training data structure

4 Classification function processing

Using a single training data structure, multiple classification scores can be used. Jaccard Similarity between two sequences represented by a set of words is calculated using the intersection divided by the union between the two sets. No frequency values are required for this similarity measure. For example, the Jaccard similarity between two sequences represented by two sets $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively, is defined as $S_J(\mathcal{S}_1, \mathcal{S}_2)$, where:

$$S_J(\mathcal{S}_1, \mathcal{S}_2) = \frac{|\mathcal{S}_1 \cap \mathcal{S}_2|}{|\mathcal{S}_1 \cup \mathcal{S}_2|}$$

Weighted Jaccard Similarity can be used when the class frequency for unique minhash values are retained in the nested categorical key-value pair collection and taken into consideration [26]. The Weighted Jaccard similarity between two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ is defined as $S_{WJ}(\mathcal{S}_1, \mathcal{S}_2)$, where $\mathcal{S}_{1_i}$ is the set frequency of token $_i$ in a set, and $_i$ iterates over all tokens:

$$S_{WJ}(\mathcal{S}_1, \mathcal{S}_2) = \frac{\sum_i min(\mathcal{S}_{1_i}, \mathcal{S}_{2_i})}{\sum_i max(\mathcal{S}_{1_i}, \mathcal{S}_{2_i})}$$

In Strand, Jaccard similarity between the sets of all words in two sequences is approximated by intersecting two sets of minhash signatures where longer signatures provide more accurate Jaccard similarity or distance approximations [27]. Class frequencies may be used to produce other Jaccard Index variations such as the Weighted Jaccard Similarity [26] shown above. However, large performance gains are achieved in Strand by using binary classification techniques where no nested categorical frequency values or log based calculations are required during classification function operations. In the binary minhash classification approach, minhash signature keys are simply intersected with the minhash keys of known classes to calculate the similarity between a query sequence and a known class.

We create a minhash signature by performing minhashing on all words in a gene sequence. The minhash signature is a collection of integers which represent the unique characteristics of all words created from a particular gene sequence and is typically much smaller than original collection of words itself

$$\mathcal{M} = \text{minhash}(\mathcal{S})$$

Minhashing allows us to efficiently approximate the Jaccard index between two sequences, $\mathcal{S}_1$ and $\mathcal{S}_2$:

$$S_J(\mathcal{S}_1, \mathcal{S}_2) \approx \frac{|minhash(\mathcal{S}_1) \cap minhash(\mathcal{S}_2)|}{k},$$

where the intersection operator is used here to indicate for how many hash functions [28] the minhash values agree

between the two signatures. This can be seen as a multivalue extension of the Simple Matching Coefficient [29] between two vectors where:

$$S_J(\mathcal{S}_1, \mathcal{S}_2) \approx S_{SMC}(minhash(\mathcal{S}_1), minhash(\mathcal{S}_2))$$

Next, we discuss scoring the similarity between a sequence minhash signature and the category minhash signatures used for classification. Category signatures are created by combining the minhash signatures of all sequences for the category in the training set. Note that this means that for many hash functions, we will have several minhash values from different sequences. Since sequences in the same category will be similar, many minhash values will agree resulting in a compact category signature.

After training has completed, classification accuracy may also be increased by making a single pass of the training data and *pruning* or removing minimum hash values which are associated with more than n categories. Our own empirical results show that the value for n may vary based upon a particular set of training data. In this research, pruning minhash values with more than one category association achieved the best results.

However, this means that we do not directly estimate the Jaccard index between a sequence and the categories, but we measure similarity based on the number of collisions between the minhash values in the sequence signature and the category signature.

Definition 1 (**Minhash Category Collision**) *We define the Minhash Category Collision between a sequence S represented by the minhash signature $\mathcal{M}$ and a category signature $\mathcal{C}$ as:*

$$\text{MCC}(\mathcal{M}, \mathcal{C}) = |\mathcal{M} \cap \mathcal{C}|,$$

where the intersection is calculated for each minhash hashing function separately.

We calculate MCC for each category and classify the sequence to the category resulting in the largest category collision count. While many other more sophisticated approaches for scoring sequences are possible, these are left for future research.

5 Applying strand to Malware classification

The Kaggle Microsoft Malware Classification Challenge (Big 2015) [19] simulates the file input data processed by Microsoft's real-time detection anti-malware products which are installed on over 160M computers and inspect over 700M computers each month [19]. The goal of the Microsoft Malware Classification Challenge is to group polymorphic malware at a high level into 9 different classes of malicious programs including: Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator.ACY, and Gatak.

5.1 The training and classification input data

Microsoft provided almost a half terabyte of training and classification input data which included:

1. **Binary Files:** 10,868 training and 10,873 test files containing the raw hexadecimal representation of the file's binary content with the executable headers removed.
2. **Asm Files:** 10,868 training and 10,873 test files containing a metadata manifest including data extracted by the Interactive Disassembler Tool. This information includes things such as function calls, strings, assembly command sequences and more.
3. **Training Labels:** Each training and test file name is a MD5 hash of the actual program. The training labels file contains each MD5 hash and the malware class which it maps to. No labels were provided for the test data input files.

5.2 Challenge evaluation, competitors, and results

Kaggle challenge participants were evaluated using a multi-class logarithmic loss score. Each test file submission made required not only the predicted malware class, but the estimated probabilities for the file belonging to each of the nine classes. Each submission record included the file hash and nine additional comma-delimited fields containing values for the predicted probability that a given file belongs a particular class. The logarithmic loss score is defined as:

$$\log - \text{loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} \log\left(p_{ij}\right)$$

Where N is the number of test set files and M is the number of classes. The variable $y_{ij} = 1$ when file i is a member of class j and zero for all other classes. The predicted probability that observation i belongs to class j is given by the variable p_{ij}. The submitted probabilities are truncated for the interval $[10^{-15}, 1 - 10^{-15}]$ by $p_{ij} = \max\left(\min\left(p_{ij}, 1 - 10^{-15}\right), 10^{-15}\right)$ prior to scoring in order to avoid extremes in the log function [30].

There were 377 international teams competing in the contest with US$16,000 in available prize money. The winning team achieved a logarithmic loss ratio score of 0.002833228 where a lower value represents a better score. The winning team reported that their model produced an accuracy level greater than 99% during 10-fold cross-validation [31]. Their process was highly specialized and

tailored specifically to the task and available data for detecting the nine classes of malware presented in the challenge. Alternatively, we present here the results of a more general and performance oriented approach which should work well on many forms of input data where generational polymorphism occurs.

The winning team's final submission used a highly complex ensemble of models [32] using a combination of features including: byte 4-gram instruction counts, function names and derived assembly features, assembly op-code n-grams, disassembled code segment counts, and image based features using the binary file data. Generating these features required 500 GB of disk space for the original training data and an additional 200 GB for engineered features. While the final features used for the model required only 4 GB, both feature creation and generating the top performing model takes around 48 h. Furthermore, it takes an additional 24 h to generate the best model ensemble which produced the winning score [32]. In short, the winning submission takes 72 h to produce using a Google Compute Engine with 16 CPUs, 104 GB RAM, and 1 TB of disk space. The winning model requires about 700 GB of disk space including 500 GB for the original data and an additional 200 GB of disk space for the meta data generated.

6 Applying strand to Microsoft Malware classification challenge

While Strand was originally designed to process FASTA formatted gene sequence files, only minor changes were required to accommodate for reading and processing the malware bytes files as input. This is possible since unlike many other sequence classifiers and k-mer counters [16, 17, 33], Strand uses no special encoding of sequence data and supports any Unicode character within the sequences.

During gene sequence classification, the short reads of sequence data commonly generated by modern sequencers can be in either forward or reverse-complement order. As a result of this limitation, classification searches on sequence data must be made using each input sequence's forward and reverse-complement effectively doubling the number of classification searches required. This particular feature of Strand is gene sequence specific and was irrelevant for malware classification. After turning off the reverse-complement search and modifying the sequence file parsing routine, Strand was able to train and classify against malware data with no other changes.

6.1 Developing binary file features for strand

We now discuss the feature engineering required to prepare bytes files for training and classification using Strand. All of the malware feature engineering required to convert bytes file program data into Strand sequences fits into just a few lines of code. While conversion of the raw hex data to a sequence alphabet (A,C,T,G) is possible, it is not required to produce satisfactory classification results. The disassembled code files (Asm files) were not used to produce the score and accuracy results presented later in Tables 1 and 2 and are covered in detail in the next section.

Figure 4 illustrates the typical content encountered within the bytes hex data files provided by Microsoft. The first eight characters of each line contain a line number, and the last line shows how some hex content is unavailable and displayed as "??". Both the line numbers and "??" symbols are removed during Strand processing.

When reading each bytes file, Strand uses the code shown in Fig. 5 to convert the bytes malware files into a Strand sequence. During processing each carriage return, space, and "?" character are removed. This produces a single string or Strand sequence containing all hex content read from the malware file. Once the malware hex data is cleaned, sequence words of length k or k-mers are generated by Strand as previously described.

6.2 Developing Asm file features for strand

Each Asm file provided for training or classification contains a metadata manifest which includes details extracted from each malware program's binary content such as function calls, assembly commands, strings, and other relevant executable data elements. This data was provided by Microsoft to Kaggle and generated by the Interactive Disassembler tool. We focused on extracting only the assembly language commands from each pure code segment contained within each Asm file.

Figure 6 shows a sample of assembly commands within an Asm file's pure code segment. Using known valid assembly language commands [34] and additional tokens extracted from the Asm training data, a list of 1251 unique Asm file commands and tokens were collected. Next, the commands were assigned a unique index number which was converted from a base 10 to base 4 value. Finally, each base 4 number was encoded to a 5 character gene sequence value where $0 = \mathrm{A}, 1 = \mathrm{C}, 2 = \mathrm{G}$, and $3 = \mathrm{T}$. For instance, the assembly command "adc" was assigned the base 10 index 4, which converts to the base 4 value 00010, and is encoded to the gene sequence value "AAACA".

Using this approach each Asm file's content is converted to a contiguous string of gene sequence characters. Furthermore, this encoding gives each assembly command or Asm file token a common length. Unique assembly commands and tokens collected ranged from 2 to 15 characters in length. Processing this data in

Table 1 Ten-fold cross-validation results when using strand with 32-bit hash codes and bytes file hex sequences to predict 10 folds from the Microsoft malware training data

10-Fold cross-validation results Binary 32-Bit						
Fold	Classified	Correct	Sensitivity	Precision	Training	Prediction
1	1087	979	90.06%	90.06%	6:46:31	0:30:38
2	1087	998	91.81%	91.81%	6:25:38	0:30:26
3	1087	1011	93.01%	93.01%	6:35:01	0:31:50
4	1087	995	91.54%	91.54%	6:34:53	0:33:53
5	1087	1004	92.36%	92.36%	6:21:12	0:28:12
6	1087	1003	92.27%	92.27%	6:27:41	0:28:13
7	1087	1006	92.55%	92.55%	6:49:04	0:33:34
8	1087	993	91.35%	91.35%	6:32:48	0:31:45
9	1086	1001	92.17%	92.17%	6:26:36	0:33:26
10	1086	995	91.62%	91.62%	6:51:28	0:29:05

a fashion similar to the bytes files as raw text would result in many of the unique token values being broken across each gene sequence word created during Strand training and classification processing. For example, creating gene sequence words within Strand of length 50 would ensure that each gene sequence word contained a block of five assembly commands or Asm file tokens. After testing word sizes of 45, 50 ,55, and 60, empirical results showed that a block of five commands optimized both sensitivity and precision during classification processing.

Extracting only the assembly command sequence data from each Asm file also greatly reduced the total size and volume of data being processed by Strand during training and classification resulting in an almost 1/7 decrease in processing time when compared to models produced by using the bytes files as input. Finally, Asm command sequences also produce a more accurate classification result as well which is shown in Section 7.3.

7 Malware classification results using strand

While we did not produce a winning logarithmic loss score for the Kaggle Microsoft Malware Classification Challenge (BIG 2015) [19], we were able to achieve a top score of 0.047999572 when using a 64-bit minhashing configuration with Strand. We used several techniques including model ensembling, pruning, and prediction adjustments based on the confidence scores produced by Strand to achieve our best score. These results are discussed in detail in the following sections.

The individual Asm sequence model is our most impressive result achieving greater than 98.59% accuracy during ten-fold cross validation and a substantial performance gain when compared to the winning team's 72 h

Table 2 Ten-fold cross-validation results when using Strand with 64-bit hash codes and bytes file hex sequences to predict 10 folds from the Microsoft malware training data

10-fold cross-validation Results Binary 64-Bit						
Fold	Classified	Correct	Sensitivity	Precision	Training	Prediction
1	1087	1053	96.87%	96.87%	6:42:54	0:33:22
2	1087	1054	96.96%	96.96%	5:53:21	0:31:36
3	1087	1069	98.34%	98.34%	6:50:26	0:34:12
4	1087	1052	96.78%	96.78%	6:32:24	0:35:00
5	1087	1065	97.98%	97.98%	6:50:25	0:32:50
6	1087	1061	97.61%	97.61%	6:36:49	0:35:21
7	1087	1063	97.79%	97.79%	6:50:02	0:34:48
8	1087	1053	96.87%	96.87%	6:33:02	0:33:30
9	1086	1059	97.51%	97.51%	6:28:38	0:30:58
10	1086	1058	97.42%	97.42%	6:35:53	0:27:17

```
00401000 56 8D 44 24 08 50 8B F1 E8 1C 1B 00 00 C7 06 08
00401010 BB 42 00 8B C6 5E C2 04 00 CC CC CC CC CC CC CC
00401020 C7 01 08 BB 42 00 E9 26 1C 00 00 CC CC CC CC CC
00401030 56 8B F1 C7 06 08 BB 42 00 E8 13 1C 00 00 F6 44
00401040 24 08 01 74 09 56 E8 6C 1E 00 00 83 C4 04 8B C6
00401050 5E C2 04 00 CC CC CC CC CC CC CC CC CC CC CC CC
00401060 8B 44 24 08 8A 08 8B 54 24 04 88 0A C3 CC CC CC
00401070 8B 44 24 04 8D 50 01 8A 08 40 84 C9 75 F9 2B C2
00401080 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401090 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010A0 08 50 51 52 56 E8 18 1E 00 00 83 C4 10 8B C6 5E
004010B0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
0042A800 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
```

Fig. 4 Malware bytes file hex data content

training time. In fact, we are able to generate the Asm training models for 10-fold cross validation in under 45 min while processing 224 GB of training data and 189 GB of test data as shown in Table 3 below.

7.1 Influence of hash code size and pruning on classification accuracy

The individual size of each hash code value making up Strand's minhash signature is critical for producing accurate classification results. While hashing words into 32 or 64-bit integers can reduce the memory footprint by very large amounts, the selected word length can also influence collisions, drastically impacting classification accuracy.

Table 4 illustrates how the available unique hash values per unique gene sequence word and potential collisions produced are influenced by both word length and the selection of an appropriate hash code size. For example, over 1 billion potential collisions per word are observed when hashing a 31 base gene sequence word into only 32-bits while a 64-bit hash provides ample room for each unique 31 base word value. Finally, the remaining sections of Fig. 4 show that a 32-bit hashing function provides enough unique values to map single words up to 16 bases in length, while a 64-bit hashing function supports unique values for single words up to 32 bases in length. Collisions may still occur when storing a 31 base word as a 64-bit hash. However, they are drastically reduced when compared to storing a 31 base word as a 32-bit hash.

Finally, minimum hash values which are associated with multiple classification categories my add noise to a given model since multiple categories may receive votes when a minhash signature contains such a multi-category value. The Strand classifier includes a function which may be executed after a training model has been created to remove any minhash values within the training data that are associated with multiple categories. Empirical results show a small lift in classification accuracy on both the bytes and Asm models when all minhash values associated with more than one category are removed.

7.2 Binary file classification results

Table 1 shows ten-fold cross-validation results for models using only bytes file hex data. Strand averaged 91.88% accuracy across the ten folds predicted using only 32-bit hashing functions. Table 2 shows 10-fold cross-validation results for the version of Strand using 64-bit hash codes. Strand averaged 97.41% accuracy across the 10 folds. When using 64-bit hashing functions, we were able to drastically reduce the logarithmic loss score produced from 0.452784 to 0.222864. While memory consumption increased slightly, there was no large degradation in training or classification performance.

The training times in Tables 1 and 2 represent the time required to train on 90% of the 10,868 Malware Classification Challenge training data records (9782 training records). The classification times in both tables reflect the time required to classify the number of records reflected in the "Classified" column which represent 10% of the training data for each fold. The 32-bit and 64-bit versions of Strand required 5.482 and 5.483 total hours, respectively, to classify all of the 10,868 training records during 10-fold cross validation.

```
StringBuilder currSeqText = new StringBuilder();

foreach (var line in File.ReadLines(FnaFileMap.FilePath))
{
    //Remove the carriage returns, line number
    // , spaces, and ??
    //values from each line of hex in the .bytes file.
    currSeqText.Append(line.Substring(9)
      .Replace(" ", string.Empty)
      .Replace("?", string.Empty)
    );
}
```

Fig. 5 Strand C# code used to process malware .bytes hex data files

```
; Segment type: Pure code
; Segment permissions: Read/Execute
_text           segment para public 'CODE' use32
        assume cs:_text
        ;org 401000h
        assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
        push    esi
            lea     eax, [esp+8]
        push    eax
            mov     esi, ecx
                call    ??0exception@std@@QAE@ABQBD@Z
                mov     dword ptr [esi], offset off_42BB08
            mov     eax, esi
        pop     esi
            retn    4
```

Fig. 6 Sample of assembly commands in an Asm file pure code segment

The 64-bit model takes up approximately 5 GB in memory and 436 MB compressed on disk while the 32-bit version takes up approximately 3 GB in memory and 255 MB compressed on disk. Due to the small size of the model, multiple copies can be loaded into memory for multiple worker processes to take advantage of process level parallelism when classifying large volumes of data. For example, 15 classification workers were used to process the test files provided by Microsoft.

7.3 Asm file classification results

When using the sequence data created from Asm file assembly commands and tokens, we were able to increase accuracy during ten-fold cross-validation while drastically reducing training and classification times to under 45 and 5 mins respectively per fold. The primary reason for this large gain in performance is a substantial reduction in input data required for processing when compared to the bytes file models. For instance, 10,873 test data files generated for Asm sequences requires 720 MB size on disk while the same number of corresponding bytes files required 47.3 GB. Likewise, 10,868 training files generated for Asm sequences required 773 MB on disk while the same number of corresponding bytes files also require 47.3 GB. The 64-bit Asm model achieved a log loss score of 0.062721944 when used to predict the Kaggle test data.

Table 3 shows ten-fold cross-validation results for models using Asm sequence data as input. We were able to achieve a higher average sensitivity of 98.59% across each of the ten folds when compared to the bytes files. Furthermore, both training and classification times are approximately eight times faster on all folds. The training times in Table 3 represent the time required to train on 90% of the 10,868 Malware Classification Challenge training data records (9782 training records). The classification times reflect the time required to classify the number of records reflected in the "Classified" column which represent 10% of the training data for each fold.

Table 3 Ten-fold cross-validation results when using Strand with 64-bit hash codes and Asm sequences to predict 10 folds from the Microsoft malware training data

10-fold cross-validation results ASM 64-Bit						
Fold	Classified	Correct	Sensitivity	Precision	Training	Prediction
1	1087	1071	98.53%	99.17%	0:40:21	0:04:19
2	1087	1071	98.53%	98.98%	0:39:11	0:05:31
3	1087	1073	98.71%	99.17%	0:39:45	0:05:19
4	1087	1061	97.61%	98.61%	0:40:44	0:05:24
5	1087	1072	98.62%	99.17%	0:37:12	0:04:49
6	1087	1074	98.80%	99.26%	0:40:21	0:05:12
7	1087	1076	98.99%	99.35%	0:41:51	0:04:33
8	1087	1076	98.99%	99.26%	0:31:06	0:04:36
9	1086	1071	98.62%	98.89%	0:38:43	0:05:08
10	1086	1070	98.53%	98.89%	0:33:34	0:04:11

Table 4 Average collisions per gene sequence word varying base word size for 32 and 64-bit hash codes

Hash size	Sequence	Avg. seq. words per hash value
32 bits	31 base word	1 073 741 824
64 bits	31 base word	0.25
32 bits	16 base word	1
64 bits	32 base word	1

Unlike the previous bytes file models, Strand's high level of accuracy leaves it unable to make a handful of predictions in some cases (43 out of 10878) which makes distinguishing between Precision and Sensitivity relevant. In the bytes model case, both values are the same for each fold. While Strand's average sensitivity was 98.59% across each of the ten folds, precision is calculated by excluding the 43 cases where Strand was unable to make a prediction. Strand's precision for the Asm model during ten-fold cross-validation is 99.10% when considering only the records which Strand is able to predict.

7.4 Ensemble classification results

We tried multiple approaches for creating ensembles using both the bytes and Asm models from Strand. While adding together minhash category scores for both models did produce a very small lift in accuracy during cross-validation, we were unable to lower the Kaggle log loss score using this approach.

There were 61 out of 10,877 Kaggle test records for which the Asm model was unable to make a prediction. The final bytes model had no such records. We combined predictions from both models by defaulting to the bytes model prediction only when no Asm file prediction was available. This approach produced a Kaggle log loss score of 0.081511944. A similar and possibly more accurate version of this ensemble could be made by generating a second Asm model using a shorter token length of 50 vs. 55 to pick up additional missing predictions. However, we leave this to future research.

Strand produces minhash collision scores for each known category within the training data repository. Results from multiple training models may be utilized to perform more accurate predictions in some instances. Since minhash collisions approximate Jaccard similarity, winning category *MCC* scores also reflect confidence in a particular prediction. For example, a 2400 value minhash signature with 2400 collisions for a given category indicates a Jaccard similarity of approximately 1 while only 15 collisions may indicate a Jaccard similarity of only 0.00625%. In fact, 15 out of the 16 incorrect predictions for Fold 1 in Table 3 had less than 221 out of 2400 minhash collisions. This represents a Jaccard similarity of approximately 0.092%, while all had less than 517 or 0.095% Jaccard similarity approximation.

Our best log loss score was achieved by taking the Asm and bytes ensemble and setting the values for all categories to $1/M$ (total number of categories) for predictions with very low confidence. In this case, the values for each category were 1/9 or equal probability when predictions had less than 15 minhash collisions. Empirical results showed that 15 was the best threshold out the values 5, 10, 15, 20, and 25. Using this approach changed 23 out of 10,877 predictions to equal probability producing a final log loss score of 0.047999572.

8 Conclusions

In this paper we have demonstrated how modern gene sequence classification tools can be applied to large-scale malware detection. In this first study, we have shown how the gene sequence classifier Strand can be used to predict multiple classes of polymorphic malware using data provided by the Kaggle Microsoft Malware Classification Challenge (Big 2015). While the approach, using only minimal adaptation, did not best the accuracy scores achieved by the highly tailored approach that won the competition, we did achieve classification accuracy levels exceeding 98% while making predictions over seven times faster than the training times required by the winning team.

From the success of this demonstration, we conclude that gene sequence classifiers in general, and Strand in particular, hold great promise in their application to security datasets. In addition to polymorphic malware, we anticipate that these classifiers can be used anywhere data sequences are used, such as in network traces of attacks or the identification of ransomware.

Authors' contributions

JD implemented the code required for the Strand application and performed all experiments on the BIG 2015 Polymorphic Malware Dataset. JD wrote the first draft version of this paper. MH made substantial contributions to the architecture of strand and the design of experiments carried out within the paper. MH gave final approval of the version to be published and made many revisions to the final publications content. TM contributed to the cyber security component of the paper making major revisions and drafting large portions of publication sections related to cyber security. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Author details

[1]Darwin Deason Institute for Cyber Security, Southern Methodist University, Dallas, TX, USA. [2]Department of Engineering Management, Information, and Systems, Southern Methodist University, Dallas, TX, USA. [3]Tandy School of Computer Science, The University of Tulsa, Tulsa, OK, USA.

Received: 6 October 2016 Accepted: 12 January 2017
Published online: 23 January 2017

References

1. F Cohen, Computer viruses. Comput. Secur. **6**(1), 22–35 (1987). doi:10.1016/0167-4048(87)90122-2
2. NPP Mavrommatis, MARF Monrose, in *USENIX Security Symposium*. All your iframes point to us (USENIX Association, Berkeley, 2008), pp. 1–16
3. McAfee: For Consumers (2014). https://www.mcafee.com/consumer/en-us/store/m0/index.html. Accessed 06 Jan 2016

4. Norton Norton Anti (2014). http://us.norton.com. Accessed 06 Jan 2016
5. M Christodorescu, S Jha, S Seshia, D Song, RE Bryant, *et al*, in *Security and Privacy, 2005 IEEE Symposium On*. Semantics-aware malware detection (IEEE, Los Alamitos, 2005), pp. 32–46
6. P Ször, P Ferrie, in *Virus Bulletin Conference*. Hunting for metamorphic, (2001)
7. JM Drew, Mass Compromise of IIS Shared Web Hosting for Blackhat SEO: A Case Study (2014). http://blog.jakemdrew.com/2015/03/10/mass-compromise-of-iis-shared-web-hosting-for-blackhat-seo-a-case-study/. Accessed 06 Jan 2016
8. Wikipedia: Agobot (2014). https://en.wikipedia.org/wiki/Agobot. Accessed 06 Jan 2016
9. M Bailey, J Oberheide, J Andersen, ZM Mao, F Jahanian, J Nazario, in *Recent Advances in Intrusion Detection*. Automated classification and analysis of internet malware (Springer, Heidelberg, 2007), pp. 178–197
10. V Total, File Statistics During Last 7 Days. https://www.virustotal.com/en/statistics/. Accessed 15 Jan 2015
11. SF Altschul, W Gish, W Miller, EW Myers, DJ Lipman, Basic local alignment search tool. J. Mol. Biol. **215**(3), 403–410 (1990)
12. WJ Kent, Blat-the blast-like alignment tool. Genome Res. **12**(4), 656–664 (2002)
13. Q Wang, GM Garrity, JM Tiedje, JR Cole, Naive bayesian classifier for rapid assignment of RNA sequences into the new bacterial taxonomy. Appl. Environ. Microbiol. **73**(16), 5261–5267 (2007)
14. RC Edgar, Search and clustering orders of magnitude faster than blast. Bioinformatics. **26**(19), 2460–2461 (2010)
15. J Drew, M Hahsler, in *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*. Strand: fast sequence comparison using mapreduce and locality sensitive hashing (ACM, New York, 2014), pp. 506–513
16. DE Wood, SL Salzberg, Kraken: ultrafast metagenomic sequence classification using exact alignments. Genome Biol. **15**(3), 46 (2014)
17. R Ounit, S Wanamaker, TJ Close, S Lonardi, Clark: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers. BMC Genomics. **16**(1), 236 (2015)
18. E Peterson, D Curtis, A Phillips, J Teuton, C Oehmen, in *Intelligence and Security Informatics (ISI), 2013 IEEE International Conference On*. A generalized bio-inspired method for discovering sequence-based signatures, (2013), pp. 330–332. doi:10.1109/ISI.2013.6578853
19. Kaggle: Microsoft Malware Classification Challenge (BIG 2015) (2015). https://www.kaggle.com/c/malware-classification. Accessed 04 Nov 2015
20. J Drew, M Hahsler, T Moore, in *International Workshop on Bio-inspired Security, Trust, Assurance and Resilience (BioSTAR 2016)*. Polymorphic malware detection using sequence classification methods (IEEE, Los Alamitos, 2016)
21. S Vinga, J Almeida, Alignment-free sequence comparison—review. Bioinformatics. **19**(4), 513–523 (2003)
22. CE Shannon, A mathematical theory of communication. ACM SIGMOBILE Mobile Comput. Commun. Rev. **5**(1), 3–55 (2001)
23. A Gionis, P Indyk, R Motwani, *et al*, in *VLDB*. Similarity search in high dimensions via hashing, vol. 99, (1999), pp. 518–529
24. hadooptutorial.info: Combiner in MapReduce (2014). http://hadooptutorial.info/combiner-in-mapreduce/. Accessed 02 Apr 2015
25. J Dean, S Ghemawat, Mapreduce: simplified data processing on large clusters. Commun. ACM. **51**(1), 107–113 (2008)
26. S Ioffe, in *Data Mining (ICDM), 2010 IEEE 10th International Conference On*. Improved consistent sampling, weighted minhash and l1 sketching (IEEE, Los Alamitos, 2010), pp. 246–255
27. A Rajaraman, JD Ullman, *Mining of Massive Datasets*. (Cambridge University Press, Cambridge, 2012)
28. J Leskovec, A Rajaraman, JD Ullman, *Mining of Massive Datasets*. (Cambridge University Press, Cambridge, 2014)
29. Wikipedia: Simple Matching Coefficient. https://en.wikipedia.org/wiki/Simple_matching_coefficient. Accessed 14 Aug 2015
30. Kaggle: Evaluation (2016). https://www.kaggle.com/c/malware-classification/details/evaluation Accessed 14 Jan 2016
31. Kaggle: Microsoft Malware Winners' Interview: 1st place, "NO to overfitting" (2015). http://blog.kaggle.com/2015/05/26/microsoft-malware-winners-interview-1st-place-no-to-overfitting Accessed: 02 Nov 2015
32. L Wang, Microsoft Malware Classification Challenge (BIG 2015) First Place Team: Say No To Overfitting (2015). https://github.com/xiaozhouwang/kaggle_Microsoft_Malware/blob/master/Saynotooverfitting.pdf Accessed: 02 Nov 2015
33. G Marcais, C Kingsford, A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics. **27**(6), 764–770 (2011). doi:10.1093/bioinformatics/btr011. http://bioinformatics.oxfordjournals.org/content/27/6/764.full.pdf+html
34. F Cloutier, x86 Instruction Set Reference. http://www.felixcloutier.com/x86/. Accessed 18 Jul 2015