KiloGrams: Very Large N-Grams for Malware Classification

Edward Raff
Laboratory for Physical Sciences
edraff@lps.umd.edu
Booz Allen Hamilton
raff_edward@bah.com

William Fleming
Laboratory for Physical Sciences
william.fleming@lps.umd.edu
U.S. Navy
william.r.fleming1@navy.mil

Richard Zak
Laboratory for Physical Sciences
rzak@lps.umd.edu
Booz Allen Hamilton
zak_richard@bah.com

Hyrum Anderson
Bill Finlayson
Endgame
hyrum@endgame.com
bfinlayson@endgame.com

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

Mark McLean
Laboratory for Physical Sciences
mrmclea@lps.umd.edu

ABSTRACT

N-grams have been a common tool for information retrieval and machine learning applications for decades. In nearly all previous works, only a few values of n are tested, with $n > 6$ being exceedingly rare. Larger values of n are not tested due to computational burden or the fear of overfitting. In this work, we present a method to find the top-k most frequent n-grams that is 60× faster for small n, and can tackle large $n \geq 1024$. Despite the unprecedented size of n considered, we show how these features still have predictive ability for malware classification tasks. More important, large n-grams provide benefits in producing features that are interpretable by malware analysis, and can be used to create general purpose signatures compatible with industry standard tools like Yara. Furthermore, the counts of common n-grams in a file may be added as features to publicly available human-engineered features that rival efficacy of professionally-developed features when used to train gradient-boosted decision tree models on the EMBER dataset.

ACM Reference Format:
Edward Raff, William Fleming, Richard Zak, Hyrum Anderson, Bill Finlayson, Charles Nicholas, and Mark McLean. 2019. KiloGrams: Very Large N-Grams for Malware Classification. In *Proceedings of KDD 2019 Workshop on Learning and Mining for Cybersecurity (LEMINCS'19) (LEMINCS @ KDD'19)*. ACM, New York, NY, USA, 11 pages.

1 INTRODUCTION

In this work, we are interested in the task of finding the top-k most frequent n-grams in a large corpus. Given a corpus C of documents, and an alphabet A, there are $|A|^n$ possible n-grams, making the use of large $n > 6$ computationally infeasible for many applications. Still, n-grams have been a bread-and-butter tool for natural language processing and other related fields for decades, thanks to their simplicity and usefulness. As such, significant work has gone into engineering systems to work with n-grams [7, 24, 28]. This is also true for malware classification, where we wish to determine whether a file is benign or malicious (malware detection), or to identify the specific family of a known malicious file (malware family classification).

In particular, we are interested in selecting n-grams for large values of n. This is motivated by the use of byte n-grams as features for malware classification. There has long existed an intuitive need for larger values of n in this space due to the nature of content encoded in executable file formats. For example, if we consider byte n-grams for Microsoft Windows Portable Executable (PE) files, one x86 assembly code instruction could be up to 15 bytes long. This would require us to consider at least 16-grams to capture this *one* instruction in context. Early work determined large values like $n = 15$ performed best [2], but this was only possible because of the small corpus size (36.9 MB). Goldberg et al. [14] proposed using 20-grams since the average malware detection signature used in 1998 was 20 bytes in length. The seminal BitShred clustering work proposed 16-byte grams, but needed a cluster of 64 machines to scale past 60,000 files, and the use of feature hashing[44] meant they did not have the original features [16]. As the size of malware corpora has grown, the exponential cost in increasing the value of n has forced researchers to consider small values of n and other alternatives. Recent works that have looked at corpora with at least 400,000 files have been constrained to 6-grams or less [34]. Considering that the Anti-Virus (AV) industry is making use of datasets that range in size from ten million [21] to hundreds of millions [41] of files, the methods that exist today simply can't scale to the magnitude of industry corpora, and old results using hundreds of files are not sufficient to base decisions on.

In this work, we introduce the KiloGram technique for efficiently finding the top-k most frequent n-grams for large values of k and n with high probability under the assumption of a power-law distribution to the n-grams. If L is the total number of observed n-grams, or bytes, in the corpus, our algorithm will take only $O(L)$ time and $O(B + k \cdot n)$ memory. The parameter B is a budget factor to control the accuracy of the method, and since $n \ll k \ll B \ll L$, this memory cost is minimal. For our tests, for example, this B corresponds to using ≈9 GB of RAM to extract frequent n-grams from 5 TB of data. For $n \in [2, 8]$, our approach is 60 times faster than previous works, and runtime does not increase with n, allowing us to test $n = 1024$ and beyond. This allows us to answer questions about

LEMINCS @ KDD'19, August 5th, 2019, Anchorage, Alaska, United States

the behavior of byte based n-grams in a more conclusive way than prior work [46].

In § 2, we review related works that aim to increase the value of n for malware classification. The proposed KiloGram algorithm will be presented in § 3. We perform the first ever investigation of large $n \in [8, 1024]$ for malware classification in § 4. We found that $n = 8$ performs well and generalizes well over three years of concept drift in malware. Ourr results show that the common assumption that n should be larger in the malware analysis space do not hold for larger modern corpora. Surprisingly, $n = 1024$ also results in nontrivial malware classification accuracy. We demonstrate in § 5 that large n-grams are interpretable to malware analysts, can help them automate laborious and error prone parts of their job, and can be combined with lower-effort domain knowledge to rival (as measured using the EMBER dataset) proprietary industry feature extractors built from decades of expertise. Finally we will conclude in § 6.

2 RELATED WORK

N-grams have been used as features for malware analysis since the first work in automating malware detection in 1995 [19], and have consistently been used for malware classification systems ever since [20, 34, 35, 42]. Except when using small datasets (hundreds of MB or less), values of $n > 8$ are never tested in published literature due to their computational burden. However an intrinsic concern is that n-grams need to be larger. For example Ibrahim et al. [15] noted that a byte 6-gram was too short to fully capture an observed x86 instruction 2.4% of the time.

In attempts to increase the n-gram length, some have developed techniques that attempt to coalesce multiple n-grams to a single canonical base form. One example of this is n-perms [18], where a sorted ordering is applied to every n-gram to map them to a single canonical form (e.g., *ACB*, *BCA*, and *CAB* would all map to *ABC*). This n-perm approach has been used for malware classification with a value of n as large as 10 [43].

While our focus is on processing the raw bytes of a binary, n-grams have been popular for assembly instructions as well. Similar coalescing techniques have been necessary to do any work with assembly due to computational constraints. Prior works have examined replacing all memory addresses, register references, and constants with generic `mem`, `reg`, and `const` placeholders [22], though it is more common to remove all instruction operands entirely [40].

While much work has gone into storing and processing known n-grams efficiently [24], little has been been done to try to extend the value of n itself in a time and memory efficient manner. The only prior work we are aware of was performed by Nagao and Mori [26]. They considered obtaining $n \in [1, 255]$ by cleverly converting the most frequent n-gram calculation into a sorting problem, resulting in $O(L \log L)$ complexity and $O(L)$ space. While exact, these bounds are worse than ours and necessitate slower out-of-core sorting than the proposed method. Furthermore, the method is limited to $n \leq 255$, and only tested up to $n = 10$. Our method relies on certain distributional assumptions to hold with high probability, but allows us significant speed and practicality benefits with $O(L)$ time and $O(B + k \cdot n)$ memory.

3 KILOGRAMMING

Our goal is to find the top-k most frequent n-grams for large values of n. To do this, we build off of two prior works. First is the hash-gram approach [33]. Hash-grams find the top-k most frequent *hashes* of n-grams. They created a large table of size $B = 2^{31} - 19$ to store hashes, and simply ignored collisions. By using a rolling hash function $h(\cdot)$ [12] , they were able to obtain orders-of-magnitude speedup over normal n-gram tabulation, at the cost of losing information about what n-grams are actually being used.

The hash-gram approach works under the common assumption that n-grams follow a Zipfian (power law) distribution [47]. The Zipfian distribution has probability mass function $f(\cdot)$ and cumulative distribution function $F(\cdot)$ given by

$$f(x; p, |A|) = \frac{x^{-p-1}}{H_{|A|}^{(p+1)}} \quad (1) \qquad F(x; p, |A|) = \frac{H_x^{(p+1)}}{H_{|A|}^{(p+1)}} \quad (2)$$

where $H_z^{(p)} = \sum_{i=1}^{z} i^{-p}$ indicates the z'th harmonic number of the p'th order, and $x \in [1, 2, \ldots, |A|]$.

Under the Zipfian-distributed assumption, it was shown that hash-grams discover the correct top-k hashes with high probability [33]. The Zipfian distribution is a surprisingly good fit to human language and many other tasks [29], and as such has been a common and useful model for n-gram based features in natural language processing[9], as well as for n-grams over bytes from binary executables [34].

Naively, one would like to use an approach such as the Space-Saving algorithm [23], which can return the top-k most frequent items from a stream. At a high level, it works as a kind of 'rank' based cache. If an item is in the Space-Saving data structure, its rank is increased as well as an associated count. If an item is not in the cache, the current item with the lowest rank is replaced, its rank increased, and it's error bound reset. Based on the current error bounds, it can estimate top-k most frequent items in a stream, and in some cases guarantee that they are the true top-k. Thanks to clever design, updates to the Space-Saving data structure are $O(1)$. In this scenario, one would treat all possible n-grams as the stream to process, and select the top-k after processing the stream[1]. However this becomes computationally intractable as k increases, and for a Zipfian distribution with $p = 0$, the Space-Saving algorithm requires $B = O\left(k^2 \log(|A|)\right)$ buckets to obtain the true top-k n-grams, resulting in $O\left(nk^2 \log(|A|)\right)$ memory use. When we consider that the size of our alphabet is a function of the n-gram size (i.e., $|A| = 256^n$), we get $B = O\left(nk^2\right)$ and a total memory use of $O\left(n^2k^2\right)$, which is not tenable if we wish to consider larger k or large n, let alone both as we do in this work. Prior works have used $k = 8,000$ as the largest k [8], which is insufficient for feature selection of n-grams where we need to preserve $k \geq 100,000$.

To resolve these issues, we introduce the *KiloGram* algorithm. This algorithm enables n-gram computation with n exceeding 1000

[1]Small scale tests on 80,000 files found that the computational overhead of the Space-Saving structure is also significant. An attempt to find the top $k = 10,000$ and $n = 6$ with $B = 1,000,000$ in this scenario took just as long as computing the exact n-grams in the first place, and failed to return any of the true top-k due to difficulty is knowing the correct budget size since the O notation hides constant factors.

by extending the hash-gram approach with a second pass that selectively leverages the Space-Saving algorithm. Its run-time complexity is $O(L)$, with two iterations over the corpus to process n-grams and place them into a hash-table (first pass) or Space-Saving data structure (second pass): either insertion is $O(1)$ complexity. The other operations in the proposed method are $O(B)$ (e.g., quick-select), and since $B < L$, we arrive at $O(L)$ total complexity. For memory, we require $O(B)$ memory for the large table T, and an additional $O(k \cdot n)$ memory for the storage of exact n-grams in the space-saving data structure, so that memory complexity is $O(B + k \cdot n)$.

Algorithm 1 KiloGramming

Require: Bucket size B, rolling hash function $h(\cdot)$, corpus of C documents, and desired number of frequent hash-grams k, and hashing stride s.

```
1:  T ← new integer array of size B
2:  for all documents x ∈ C do                    ▷ O(L) for L total n-grams
3:      for n-gram g ∈ x do
4:          q' ← h(g) mod B
5:          if q' mod s = 0 then                  ▷ Hashing-Stride check
6:              T[q'] ← T[q'] + 1
7:  T_k ← QuickSelect(T, k)
8:  S ← new Space Saving structure with B_S buckets.
9:  for all documents x ∈ C do                    ▷ Second pass over data
10:     for n-gram g ∈ x do
11:         q' ← h(g) mod B
12:         if q' ∈ T_k then
13:             Insert g into S
14: return top-k entries from S
```

The pseudo-code is given in Algorithm 1. On the first pass through the dataset, we use the hash-gram approach of creating a large table to find the top-k most frequent hashes, which under the assumptions of a Zipfian distribution, will find the true top-k hashes with high probability [33]. The hash-graming corresponds to lines 1–4 and line 6. Line 5 is an addition we will discuss soon in § 3.1.

Once we have the set of the top-k hashes, we create a new Space-Saving data structure to help us keep track of the corresponding top-k n-grams. We will perform a second pass over the data, and use the top-k list of hashes as a white list for the Space-Saving algorithm. In this way the majority of observed n-grams will not be processed because they do not have one of the specified hash values, and the Space-Saving structure allows us to filter out the collisions from the true most-frequent n-grams. We require only $O(k)$ buckets in the Space Saving structure for all practical use cases, which we prove in § 3.2, resulting in $O(k \cdot n)$ memory use for the second step. This dramatically reduces the amount of memory required, and runs orders of magnitude faster than attempting to use the Space-Saving approach on the entire corpus. The second pass over the data requires less time to run than the first pass because fewer memory accesses are being performed (≥99.99% of n-grams are non-frequent [34]), and these memory accesses result in more cache hits (smaller Space-Saving structure compared to large array T). In testing, the second pass can account for as little as 9.76% of the total runtime.

3.1 Hashing-Stride

We introduce the concept of a *hashing-stride* of size s to further enhance the utility of the n-grams found so that they are useful for creating features. The application of the hash-stride is simple. For each n-gram g, we will compute its hash $q = h(g)$. If $q \mod s \neq 0$, the n-gram is discarded. Thus, hash-striding is simply a deterministic downsampling of input n-grams by a factor of s.

Hash-striding is important to reduce redundancy caused from the sliding window effect across long common sequences. In particular, for a ubiquitous sequence of length $\ell > n$, the resulting top k n-grams would be dominated by $\ell - n + 1$ equally frequent and essentially redundant sub-sequences. Including these n-grams in the top k effectively reduces k by a factor of $(\ell - n)$.

A naive alternative to reduce the number of n-grams considered for the top-k is to use a spatial stride z, where one steps by a constant number of z grams through the input sequence. However, if a frequent n-gram does not occur at intervals of exactly z, this approach would fail to identify occurrences of the n-gram, resulting in inaccurate counts or in the worst case, exclusion. By using a hashing-stride of s, we reduce the total expected number of unique n-grams to process by a factor of s. This is because for any particular n-gram g, we will always count its occurrence regardless of its offset within a file. This ensures that counts of n-grams are accurate.

From an implementation perspective, hashing-stride allows one to perform a necessary first approximation to feature selection without having to perform any kind of communication or coordination between files, and without any additional significant computation. This also means we are technically selecting the top-k n-grams from $|A'|/s$ unique n-grams, where A' is the set of observed n-grams from the possible alphabet A (i.e., $|A'| \leq \min(L, |A|)$). We will continue to refer to this as just the "top-k" for brevity. For all experiments, unless stated otherwise, we use a hash-stride of $s = \lceil n/4 \rceil$.

3.2 KiloGrams under the Zipfian Distribution

We now prove that Algorithm 1 preserves the correct top-k n-grams when A follows a Zipfian distribution. In what follows, $L \geq |A'|$ represents the total number (including duplicates) of n-grams in the corpus. In the proof (see § 3.2.1), it is assumed that the first pass of the algorithm has obtained the true top-k hashes of the top-k n-grams, which was previously proven to occur with a high probability [33]. The proof continues by showing that given the true top-k hashes and $p \geq 1$, the expected number of colliding non-frequent n-grams (including duplicates) is upper bounded by

$$6L/\left(B\pi^2\right). \tag{3}$$

Thus we may preserve the true top-k by having a sufficiently large hash-table to disambiguate the frequent and non-frequent collisions.

Since our implementation is in Java we use $B = 2^{31} - 19$, the largest prime array size allowed by Java. This value is also realistic and requires only *8.6 GB* of RAM, well within the capacity of a modern laptop. With this, Algorithm 1 across one *petabyte* of n-grams ($L = 10^{15}$), we would expect at most 283,100 collisions. As such, adding a constant of 300,000 to the size of the Space-Saving structure S in Algorithm 1 should suffice for any application which could practically run on a single computer. We include $3 \cdot k$ as an

alternative hedge against any situation where our empirical data does not follow a power-law type distribution. Thus, in experiments, we use $B_S = \max(k+300000, 3 \cdot k)$ buckets. In all of our experiments, the bound shown in Equation 3 was never violated.

The Space-Saving structure is unnecessary for the proof, but included to ensure our approach will work should the true distribution depart from a Zipfian distribution. The Space-Saving algorithm also allows for a similar bound on the total number of buckets needed. Following the proofs in [23], and noting that $B > n \cdot k$ for all of our experiments, we reach a bound that $B_S = O(k)$ for all possible Zipfian data streams.

3.2.1 Derivation of KiloGram Bound. For a uniform hash function, the expected number of collisions for any individual bucket is L/B. There are k buckets of interest corresponding to the top-k most frequent n-grams. If $f(x; p, |A|)$ is the Zipfian probability distribution function with cumulative distribution $F(\cdot)$, the total number of observed n-grams that do not collide with the top-k is $L \cdot (1 - F(k; p, |A|))$. Then, the expected number of infrequent n-grams that collide with the top-k n-grams is then equal to this value times k/B:

$$k \cdot L \cdot \left(1 - \frac{H_k^{(p+1)}}{H_{|A|}^{(p+1)}}\right)/B \tag{4}$$

Equation 4 includes multiple occurrences of the same infrequent n-gram, and so is pessimistic. If one makes the Space-Saving algorithm large enough to handle all possible collisions, then the true top-k n-grams are obtained with high probability. This is because the Space-Saving algorithm degrades to a simple hash-table that counts everything exactly when the number of buckets in the Space-Saving data structure is greater than or equal to the number of unique items in the table.

From a theoretical perspective, using the Space-Saving algorithm instead of a hash-table gives us added flexibility to deal with the rare possibility of having more than the expected number of n-gram collisions. More practically, we use the Space-Saving algorithm because real data is not *truly* Zipfian distributed, and this gives us a method of gracefully handling deviations from theoretical expectations.

Considering the expected collisions in Equation 4, we can make some practical simplifications given hardware constraints and data assumptions. First, we pessimistically assume that $p = 1$, which is the worst case for "interesting" power law distributions observed in real data, which generally fall in the range of $[1, 4]$ [11].

Next, we pessimistically assume that the alphabet A is infinite in size. This technically degrades to the Zeta distribution, and is pessimistic because it maximizes the amount of probability mass that exists in the tail of the distribution (i.e., reduces the value of $F(k;, p, |A|)$). Doing so, we obtain

$$\lim_{|A|\to\infty} - k \cdot L \cdot \left(\frac{H_k^{(2)}}{H_{|A|}^{(2)}} - 1\right)/B = \frac{kL\left(\pi^2 - 6H_k^{(2)}\right)}{B\pi^2}$$

which further simplifies to

$$\frac{6 \cdot k \cdot L \cdot \psi^{(1)}(k+1)}{B\pi^2}$$

where $\psi^{(\alpha)}(\beta)$ is the PolyGamma function. This simplification is significant, because $\forall k \geq 1, k^{-1} > \psi^{(1)}(k+1)$, allowing us to replace the PolyGamma function with a pessimistic upper bound. Further, because $\lim_{k\to\infty}\left(\frac{1}{k}/\psi^{(1)}(k+1)\right) = 1$, this upper bound is tight. We may further simplify by replacing the PolyGamma evaluation with $1/k$, yielding Equation 3.

The bound in (3) states that the number of collisions is linear in the total number of n-grams processed. This is not a surprising result. More important is that we have a numerical upper bound that can be employed to reduce the number of collisions dramatically.

Under a more pessimistic assumption that $p < 1$, the (3) bound would not hold. However, a similar result can be obtained using the Spave-Saving structure. For that algorithm, Zipfian data with $p = 0$ (the worst possible case) requires $O(k^2 \log(|A|))$ buckets. Since we have already found the top-k colliding hashes based on a table of size B, plugging this into the proof from [23] leads to a requirement of $O(B^{-1}k^2 \log(|A|))$ buckets. Noting that for processing n-grams, $|A| = 256^n$, this can be simplified to $O(B^{-1}nk^2)$. We note that in all experiments in this paper, $B > n \cdot k$, which allows those terms to cancel. This leaves the Kilogram approach with a total of $O(k)$ buckets to process any Zipfian dataset for all $p \geq 0$, a considerable improvement compared to $O(nk^2)$ buckets for the Space-Saving algorithm alone.

4 CLASSIFICATION RESULTS

To test and evaluate the proposed KiloGram approach, we make use of four datasets that include Windows PE files and Adobe Portable Document Format (PDF) files. The datasets are summarized in Table 1, with more detail in the appendix.

Table 1: All datasets used in our experiments, including size of training and testing sets, and primary year the data is from.

Dataset	Year	Train	Test	Storage Size
Industry EXE	2014-2015	2,011,786	400,000	5 TB
EMBER	2017	600,000	200,000	936 GB
Public PDF	2018	75,1829	83,780	464 GB
VirusShare-20C	2013-2018	160,000	40,000	141 GB

The "Industry EXE" dataset was provided to us, under a non-disclosure agreement, by a third party AV company. The training set contains 2 million Windows PE executables, evenly split between benign and malicious [32], and a test-set of 400,000 binaries, also evenly split[34].

The files from which the EMBER dataset [4] were created can be obtained from VirusTotal [1]. EMBER has an even split between benign and malicious, and since it is 2-3 years newer than Industry EXE, we can use it as an extreme test of generalization over time. This is important since malware is known to exhibit concept drift [17].

Our "Public PDF" dataset was constructed from VirusShare for PDF malware [36] (19% of the data) and using Common Crawl[2] for benign PDF files.

Our "VirusShare-20C" (or "VS-20C") dataset was constructed from VirusShare [36], using AVclass to identify 20 PE malware families with exactly 10,000 total samples each [38, 39].

We use all four datasets in our evaluations throughout the paper, and observe consistent results across each in terms of the nature of larger n-gram sizes. For clarity, we consider each dataset in turn to highlight results and behavior across the four sets. Following [34], we use Elastic-Net regularized logistic regression [30, 45, 48] to train predictive models from the byte n-grams. Using the elastic net regularizer of $\|w\|_1 + 0.5\|w\|_2^2$ provides an important feature selection as part of the model training process, as the $\|w\|_1$ term will shrink insignificant features to 0, and provides empirical and theoretical robustness to high-dimensional problems with noisy and irrelevant features [27]. To parllelize the Kilo-Gram algorithm, we use the approach specified in [31] for lines 1-6, and naive parallelization of the Space-Saving algorithm using the approach of [8] to merge Space-Saving data structures for lines 8-13. QuickSelect is run as a single thread.

For all datasets we use *balanced accuracy* [6], which re-weights the test data as if there were an equal number of files in all test sets. This is done to make the accuracy number comparisons more meaningful across each dataset, where there may be slightly different ratios of benign-to-malicious files. For our binary classification problems, we will also use the Area Under the ROC Curve (AUC) [5]. This metric is of particular interest in malware detection, since one wishes to select a threshold that corresponds to low false positive rates, and AUC is the integral of true positive rate across all false positive rates, without requiring one to select a threshold *a priori* (in contrast to accuracy).

4.1 Hashing-Stride Improves Performance

First, we evaluate the inclusion of our hashing-stride approach, as discussed in § 3.1. The expectation is that, as n becomes larger, the performance of models built from the top-k most frequent features will drop due to an increasing redundancy in the top-k list. Our results back up this theoretical prediction, as shown in Figure 1.

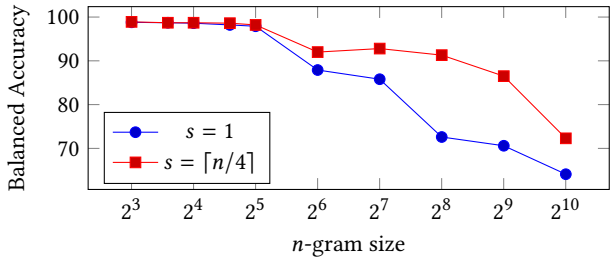

Figure 1: Balanced Accuracy results (y-axis) on the Public PDF dataset as we increase the n-gram size (x-axis, log-scale), and alter the hashing stride s. Using a hashing-stride retains more performance as n becomes larger.

[2]http://commoncrawl.org

Figure 1 shows that for small $n \leq 16$, the absolute difference in accuracy is less than 0.1 in all cases, and the hashing-strides are correspondingly small values $s \in [2, 4]$. At $n = 32$ the performance gap increase slightly, and by $n = 64$ the difference becomes significant. Across all $n \in [8, 1024]$, the use of a hashing-stride ($s = \lceil n/4 \rceil$) dominates a naive approach without a hash-stride ($s = 1$). This result appeared across all datasets, so for the remainder of the paper all results are shown with the hashing-stride of $s = \lceil n/4 \rceil$. In extended testing, we also investigated other ratios such as $s = n/2$ and $s = n$. While all $s = O(n)$ performed better than $s = 1$, the choice of $n/4$ seemed to consistently perform best among the options tested.

4.2 Computational Efficiency of KiloGrams

Computing the top-k most frequent n-grams has historically been computationally demanding, restraining most to consider only $n \leq 6$ unless working with small datasets. We have shown, from a theoretical view, that the KiloGram algorithm is $O(L)$ complexity and practically fixed memory cost at $O(B + k \cdot n)$. We now show that this result is matched empirically.

We measure runtime on a server with four Xeon E7-8870 CPUs for a total of 80 cores, 2 TB of RAM, and 40 TB of SSD storage. Because of the hashing-stride, we find that the runtime tends to decrease as n increases. For the VS-20C corpus, computing 8-grams took 27 minutes whereas 1024-grams took only 12 minutes.

While our primary results come from the use of a powerful server due to the need to train large logistic regression models, we note that such high-end equipment is not necessary to perform the n-gramming. The nature of the KiloGram algorithm means that any machine with ≈10 GB of RAM should have no difficulty in performing the computation. To emphasize this, we re-ran the same KiloGram code on a workstation with a 10 core Xeon E5-2650 at 2.30GHz, 128 GB of RAM, and a 4 TB SSD. It took only 41 minutes to compute the 1024-grams on this machine. Thee KiloGram algorithm can apparently run on modest hardware thanks to its computational and memory efficiency.

Even if one is interested in small values of n, the KiloGram approach exhibits superior run-time complexity and can provide dramatic speedups over naive approaches. On the largest corpus, Industry EXE dataset (2M files), KiloGram took $\leq$ 12 hours of computation for all values of $n \leq 1024$. Mature code with three-years of performance tuning required one month to compute 6-grams in the classical way: a 60x speedup for Kilogram over this baseline.

4.3 Investigating Large n

As we discussed in § 2, many have suggested the need for large n-gram sizes in building models for malware classification. However, after an extensive literature review, we found that no prior work empirically evaluated large n-grams on a large modern dataset. We present the first evaluation of large n-grams, and show in Table 2 the balanced accuracy and AUC across all four datasets. The last "Ind-2-EMBER" columns show results applying a model trained on Industry EXE to the EMBER test set, making a strong test for durability against concept drift over three years.

Across each dataset, we found that predictive accuracy does not increase beyond $n = 8$. Indeed, the maximal performance on all

Table 2: Results as n increases, using hashing-stride.

	Industry EXE		EMBER		Public PDF		VS-20C	Ind-2-EMBER	
n	Acc	AUC	Acc	AUC	Acc	AUC	Acc	Acc	AUC
8	98.2	99.8	99.2	99.9	98.9	99.7	95.2	97.6	99.7
12	97.5	99.7	98.9	99.9	98.7	99.7	93.8	97.4	99.4
16	96.7	99.5	98.6	99.8	98.7	99.6	92.3	95.9	98.9
24	96.4	99.4	97.9	99.7	98.6	99.6	88.1	95.5	98.4
32	96.0	99.3	97.1	99.4	98.2	99.6	85.2	93.9	97.9
64	94.9	99.1	96.3	99.2	92.0	99.3	87.4	92.9	96.8
128	94.0	98.7	93.6	97.8	92.8	99.0	79.4	88.9	94.9
256	92.6	98.0	90.3	95.6	91.3	98.5	76.5	86.6	91.9
512	92.2	96.8	78.7	84.8	86.5	96.8	71.7	71.9	69.9
1024	91.9	96.1	78.6	85.2	72.3	90.9	67.1	72.6	72.6

metrics occurs at $n = 8$. With some variation, we found that the performance in AUC degrades slowly for $n \leq 32$ across all datasets, but accuracy sometimes degrades faster. For example, the gap on the Public PDF dataset for $n = 8$ and $n = 32$ is 0.7 points, but is a more significant 10.0 points for the VS-20C corpus.

More surprising was that 1024-grams had *any* predictive utility at all, let alone reaching 90%+ accuracy or AUC across many of our datasets. Our intuition was that n-grams for large n would be extremely brittle, common accross only a few sample, and therefore ineffective for generalizing to new files. This was not the case, however, and suggests re-use (perhaps in the form of code, header information, resources, or compiler fingerprints) in EXE and PDF document formats that allow these n-grams to generalize. We also see from the "Ind-2-EMBER" experiment that these n-grams can generalize across *years* of concept drift. At $n = 8$, a small loss of 0.6 points occurs. As n gets larger, the performance after three years drops faster, indicating that after $n \geq 128$, they lose significant robustness to concept drift.

Given these results, we expect that as the size of n increases, the features may begin to correspond to ever more specific indicators of benign or malicious intent. For example, use of the Windows API function "GetProcAddress" is a common indicator of maliciousness across many Windows PE malware samples and can be detected with $n \leq 6$ [34], but this indicator alone is not enough to detect malicious files since there are many benign use cases for this function. As n becomes larger, we expect to see features that instead address sub-populations of malware, rather than the population at large.

In Figure 2, we plot the balanced accuracy as a function of the number of non-zero (NNZ) weights in the learned Elastic-Net regularized logistic regression model, where fewer non-zero features corresponds to a larger regularization penalty λ. Here we see that for small n, there is a smooth and continuous increase in accuracy as more features are selected along the regularization path. As n increases, the behavior transitions to an initial rise in accuracy, followed by a plateau once a minimum number of features are obtained. The start of this plateau occurs earlier and the initial slope larger as n becomes larger.

This behavior is intuitive, and corresponds with our expectation that the specificity of n-grams will increase with their size. For small values, a large number of n-grams are necessary to cover a wide range of smaller components that reflect the work and actions of larger features. At larger values of n, the model quickly selects

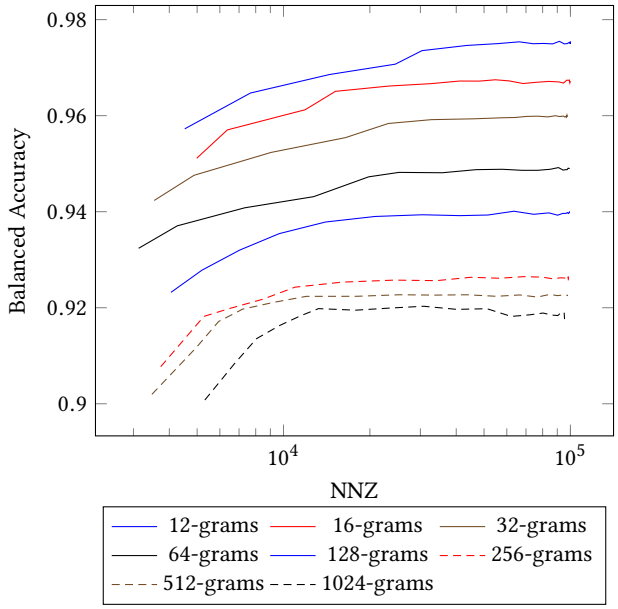

Figure 2: Balanced Accuracy results (y-axis) on the Industry EXE dataset as the number of non-zero weights (x-axis) learned by the logistic regression model increases.

all "useful" features. We believe this is because it becomes easier to delineate the predictive subset due to feature occurrences clustering around increasingly specific subsets of the population. Once a sub-population is well separated, additional features have little value unless they can "carve off" a different sub-population, and so performance plateaus. A unique benefit of larger n-grams is their increased interpretability, which allows us to provide additional evidence to this interpretation of our results in § 5.

5 FEATURE ANALYSIS

The previous sections described how KiloGrams are computed, and how they perform as features in a machine learning algorithm. For malware classification we find large KiloGrams have considerably more value in their application to analyst work-flow and integration into larger systems. In this section we will describe how malware analysts can use larger n-gram features in the course of their investigations, how they can be used in current signature based tools like Yara, and how they can be integrated with domain knowledge features to build a more powerful malware classification system. In going through a number of n-gram features, both experienced and junior analysts determined that it usually took a few minutes to understand what a single feature meant or represented, with some features taking longer.

5.1 Analyzing Individual Features

In a machine learning context, there are many features that could be pulled from binary files for use in classification, such as information from the PE header, printable strings, or (in our case) raw byte sequences. Malware analysts, whose job usually involves dissecting pieces of malware to write detection signatures or understand how

they operate, work with many of these feature types in the course of these investigations. From the point of view of these analysts, a machine learning system based on small n-grams is opaque; it takes many of these features to compile enough evidence to apply a benign or malicious label, and since these n-grams may be only a portion of a single x86 instruction as discussed earlier, they are not very helpful in shedding light on why the algorithm chose the label it did. When working with malware analysts in production, the inability to understand what a particular feature means when it is found in a binary has been a source of frustration, and impeded adoption.

The sheer size of KiloGrams changes this dynamic and presents a way to interpret features in what may otherwise be a black box. The presence of a KiloGram provides an immediate indication of where to start looking, and a large enough section of bytes to be meaningful to an analyst. Features that contribute the most to a malicious/benign decision can contain strings, embedded data like images, or code (that may or may not require disassembly or decompilation[3], depending on file format). This can help analysts reach a conclusion about a binary's nature, which is important since the average time to process a malicious file is 10 hours [25]. The ability to understand what the features mean is also important to build trust so that developed solutions will be adopted and used. Below we will provide some examples of the interpretability of features found by malware analysts at different n-gram sizes.

5.1.1 EMBER Examples with 64-grams. In Figure 3, we see the disassembly of a code snippet discovered by a 64-gram that occurred in 8% of the EMBER dataset. Upon inspection this code assembles the string "VirtualAlloc". This is then later used to obtain the "GetProcAddr" function in an obfuscated manner, so that the binary can then load other libraries at runtime. This is a technique to obfuscate the true intentions of the binary from malware analysts, and is considered a strong indicator of maliciousness.

```
push eax #50
call DWORD PTR [ebp-0xcc] #ff9534ffffff
mov DWORD PTR [ebp-0x20], eax #8945e0
mov DWORD PTR [ebp-0xa4], 0x74726956 #c7855cffffff56697274
mov DWORD PTR [ebp-0xa0], 0x416c617f #c78560ffffff75616c41
mov DWORD PTR [ebp-0x9c], 0x636f6c6c #c78564ffffff6c6c6f63
and DWORD PTR [ebp-0x98], 0x #83a568ffffff00
lea eax, [ebp-0xa4] #8d855cffffff
push eax #50
push DWORD PTR [ebp+0xe] #ff750e
xor bh,bh #30ff
xchg ebp,eax #95
cmp bh,bh #95
.byte 0xff #ff
```

Figure 3: Example of a disassembled 64-gram feature found in the EMBER dataset. The hex values of the raw bytes are shown in comments for each line of assembly.

A considerable number of 64-grams contained sub-strings of registry keys. In the EMBER dataset, 10% of malware was found to have `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\Explorer`, which may be used to hijack the explorer process. Another 12% contained `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run` for securing persistence for the malware.

One 64-gram triggered on the image icon shown in Figure 8 in § A.2 of the Appendix. Several malware samples contained precisely this icon in the resources section, causing it to become a predictive feature. This is another example of a feature with an easily understood nature, thanks to the large scale of 64-grams, which would have been uninformative if broken up into standard $n \leq 6$ grams.

5.1.2 PDF Example from 512-grams. For an example of a 512-gram, Figure 4 is a fragment from our Public PDF dataset discovered by simply looking through the extracted features. The scripting code was identified as JavaScript, which is often used in PDF files, but more frequently used (and obfuscated, as in this case) in malicious PDF files. The long string in the middle also stood out; 0x41 is the letter 'A' in hexadecimal format, and long strings of this character are often used by exploit writers to assist in crafting the correct string to take advantage of a vulnerability.

```
ey=aba();ek=24;ek++;ef=6192;ef--;if(ey>=wf){ep=4810;if(ep!=null){ev
    =3742;ev+=0.013}eq=0.03;eq--;ed=we;ea=0.017;if(ea==0.0161){er=
    'tae'}ex=sub(wd,wu);ez=5544;ez++;ej=true;es=wy;eu=8;eu--;eb=''
    ;en=9;en++;eh='';eo=0.034;eo++;ei=6343;ei++;
    if(ey&lt;f){el=8952;el--;eb=r;mt=6172;mt+=0.0101;mw=1834;mw-=7
    115;eh='4c20600f0517804a3c20600f0f63804aa3eb804a3020824a6e2f80
    4a4141414126000000000000000000000000000001239804a6420600f0004
    0000414141414141141';me=6;me++}else
    if(ey&lt;h){mm=18;mm++;eb=u;mg=false;mc=[7,35,21,4wp>null){wv=
    0.0082;wv++}wf+=2177;wq=0.0032;wq+=3755;wd=zoa('qXM7reN6',15);
    wa=0.006;if(wa!=21){wr=0.014;wr++}wx=zoa('tCTF3OdREync',13);wz
    =[40,24,32,8,48,0,16,56];wj=0.008;if(wj&lt;0){ws=null;ws+=12}w
    u=18690;wb=0.007;if(wb!=13){wn=[0,16,8,24];wh=17;wh+=7832}wu-=
    7706;wo=28;if(wo==8){wi=['fen','lag','het']}wl=22;if(wl&lt;437
    0)
```

Figure 4: A 512-gram found in PDF dataset. This example contains obfuscated JavaScript code to build an exploit string targeting particular versions of PDF reader software.

The next step in analysis is to find and extract the entire piece of embedded code. In this case a web search yielded the exploit code in a public repository. The code was then manually deobfuscated which revealed its functionality. Two different versions of an exploit string are built, depending on the software version (the target is likely to be Adobe Acrobat, the most popular PDF reader, but this was not confirmed). A successful exploit results in a visit to http://phjqxagpgdw.com/nte/goldmn.asp; and a quick search revealed a number of identified malicious domains with this naming scheme.

5.2 Features as Signatures

As the value of n increases, the resulting features gain increased specificity in the files they target. It also becomes less likely to observe an exact n-byte sequence by chance. This inspired us to explore how well some of these larger n-grams might serve as Yara signatures for malware detection. Yara [3] is an industry standard regular expression tool designed for malware analysis. Yara rules are usually designed to have low false-positive rates, in the sense that

[3]Disassembly is the process of converting raw bytes into the low-level assembly language they represent, while decompilation converts assembly instructions into a higher-level language like C.

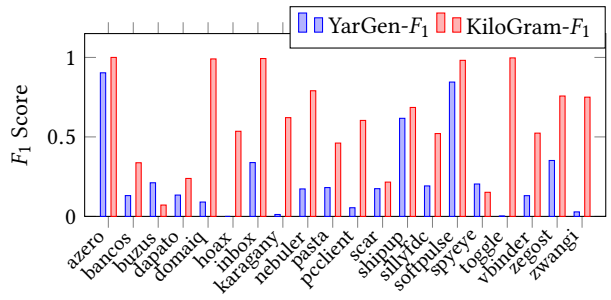

Figure 5: The F_1 score of Yara rules automatically generated from the VirusShare-20C training data, and evaluated on the test data. A different rule set is created for each family.

if a specimen causes a rule to match, the specimen is probably malicious. Using KiloGrams this way may seem counter-intuitive: Our n-grams make powerful features for machine learning algorithms, which many believe will ultimately replace signature-based malware detectors. However, signature-based systems are still widely used, and will likely have a large role to play in layered defensive systems for the foreseeable future.

YarGen[37] is the only currently-maintained tool we know of for automatically generating Yara signatures for Windows PE files. YarGen uses a number of domain knowledge processing steps to create a signature from several files. Where it is feasible, we use YarGen as a comparison to our method.

For our approach, we will use the coefficients learned by logistic regression to select the 4000 n-grams most indicative of the class we are interested in. We then look at the false positive rate of each individual n-gram (on the training data), and discard any with a false-positive rate above 5%. If any two or more n-grams *always* co-occur, we randomly select one of the n-grams and discard the others. We then combine the remaining set to form a simple Yara rule, which looks for the exact n-grams, and fires if any of the n-grams occur. Normally a combination of sub-rules is necessary; for example, YarGen usually only fires if 3 or more out of a list of patterns match. However, the statistical improbability of any individual KiloGram makes them independently robust detectors. After creating a rule for many values of n, we select the rule with the best F_1 score on the training set.

5.2.1 Results on VirusShare-20C Dataset. First we used the malware family dataset described in § 4 to automatically create Yara rules to identify specific families. This is a common and arduous task normally done manually by a malware analyst. We trained one family-vs-the rest for $n \in [8, 1024]$, and found that no single value of n was best for all families. For 9 out of 20 families, $n = 1024$ did perform best, which is a trend counter to the use of KiloGrams as purely predictive features in § 4. We compared the results of our new approach to the existing YarGen in Figure 5, where we look at the F_1 score for each family. A Wilcoxon signed rank test [13] shows that our KiloGram based approach is better, with p-value of 3.1×10^{-5}.

These results should not be taken to mean that YarGen is inferior to KiloGrams; YarGen is a tool that iterates over various features (strings, byte opcodes, etc) to find the best predictive rules. In a sense, KiloGrams represent a new class of features that tools such as YarGen could incorporate to improve their detection rates. We believe these results show that KiloGrams can be a valuable tool in the creation of signatures for malware detection, and the combination of these large features with machine learning tools can help automate the process of signature creation.

For a final test the Yara rules used for each family were run over the EMBER benign test set, as having low false positives on benign files is a critical feature. Note that no benign datasets were used in the creation of these KiloGrams. The KiloGram based rules had a median false-positive rate of 0.0065%. This is 24× better than YarGen, which had a median false-positive rate of 0.1595%.

5.2.2 Results on Industry EXE Dataset. We repeated the same experiments on our Industry EXE dataset, to create a *generic* "malware" signature. This is unprecedented in the standard use of Yara, which is meant for identifying files of a *specific* nature. YarGen failed to run on 2 million files in a timely fashion, so we are unable to compare with any prior works in the goal of creating a generic malware signature. The results when attempting to use different values of n are shown in Table 3.

Table 3: Yara generation results on Industry EXE

n	# of rules	True Neg	False Pos	False Neg	True Pos	Precision	Recall
32	25	36.643%	63.357%	15.742%	84.259%	57.08%	84.26%
64	23	81.554%	18.446%	56.791%	43.209%	70.08%	43.21%
128	22	98.599%	1.401%	78.44%	21.56%	93.90%	21.56%
256	4	100%	0%	95.034%	4.966%	100.00%	4.97%
512	31	99.987%	0.013%	78.777%	21.223%	99.94%	21.22%
1024	52	99.947%	0.054%	76.68%	23.32%	99.77%	23.32%
2048	35	99.989%	0.012%	78.392%	21.609%	99.95%	21.61%
4096	84	99.992%	0.009%	89.79%	10.21%	99.92%	10.21%
8192	145	99.684%	0.316%	89.61%	10.39%	97.05%	10.39%
[256-4096]	206	99.938%	0.063%	74.958%	25.042%	99.75%	25.04%

We can see that any $n \in [256, 4096]$ produces signatures with low false positive rates, and surprisingly can catch up to 23% of the malware in the test set. Naively combining all KiloGrams in this range into one larger signature of 256 through 4096-grams boosts the recall up to 1/4 of the test set malware. This produced 125 false positives, which we investigated with VirusTotal[1]. Of these, 10 are now reported as `OutBrowse` malware; 45 behave very similarly to each other and are almost certainly adware/spyware; 10 are unsigned (a huge red flag) versions of mmc.exe (Microsoft Management Console) in various languages; and another 11 have other malicious indicators (such as 1 or more malicious AV reports or relationships with other malicious files). Only 49 of the 125 reported false positives display no evidence of being malware.

Further, we tested the Industry EXE generated signatures on the EMBER test set, which is 2-3 years newer. This signature was still able to catch 8.7% of the EMBER malware, with a false positive rate of 0.0093% on the EMBER benign set. The ability of these signatures to catch mislabeled data in our test set, and still generalize to data three years later (despite the concept drift common in this domain) increase our confidence in the usefulness of KiloGrams as signatures.

5.3 KiloGrams & Domain Knowledge

Based on the analysis in § 5.1 we find large n-grams represent interesting and relevant features present in large sub-populations of the malicious or benign binaries. This leads us to ask, can combing large n-gram features with human-engineered features produce a stronger model?

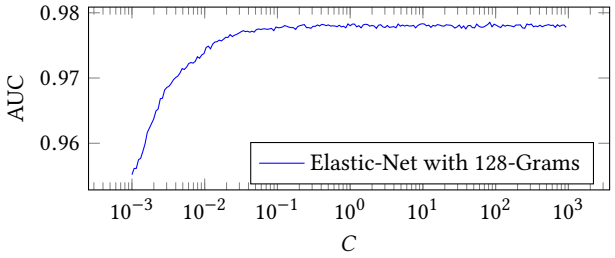

Figure 6: AUC as a function of regularization parameter C using Elastic-Net to force most coefficients to zero. Selecting $C = 10^{-1}$ gave 17,294 nonzero 128-gram features.

To assess this, the top 100,000 128-grams were extracted from the EMBER training files. Using the Elastic-Net regularization path on the training data (Figure 6), we selected the regularization parameter, C, corresponding to the sparse subset that maximizes AUC. This resulted in $C = 10^{-1}$ and 17,294 nonzero 128-gram features. We measured the lift provided by these 128-gram features by prepending the counts to two different domain knowledge feature sets: the 2,351 EMBER features crafted via domain knowledge, as well as a production set of feature extractors designed by malware analysis experts. These were compared to 128-grams alone, EMBER features alone, and to proprietary features alone. We trained a gradient-boosted decision tree model using `xgboost` on each of these feature sets, with 200 boosting rounds, tree depths up to 9 levels, 50% column subsampling per tree, and a $\eta = 0.29$ learning rate [10]. ROC curves on the validation features are shown in Figure 7.

Adding 128-grams improved AUC in all cases. EMBER features alone achieved an AUC of 0.999597, and improved to 0.999718 when augmented with 128-grams. The proprietary features result in a slightly higher AUC at 0.999822, and further improved to 0.99985 when augmented with 128-grams.

For a production malware detector deployed as an anti-virus, we care about the true positive (TPR) rate at n very low false-positive rate (FPR). The zoomed inset in Figure 7 shows the TPR at at FPR of 5:10000, which is reasonable for a production system. At that rate, the TPR of EMBER with 128-grams is comparable to the proprietary features alone, and then further outperformed by proprietary features with prepended 128-gram counts. Also of interest is the ROC curve for 128-grams alone, which exhibits a peculiar jump in TPR near 2×10^{-3} FPR. This fits intuition that KiloGrams are essentially "getting the easy ones" via the top k n-grams spanning large subsets of malicious or benign PE files. Feature combinations involving domain knowledge features cover the remaining samples. Indeed, it required 20 boosting rounds for a model trained on EMBER features to exceed 0.999 AUC on the evaluation set, but only 15 boosting rounds when augmented with KiloGrams.

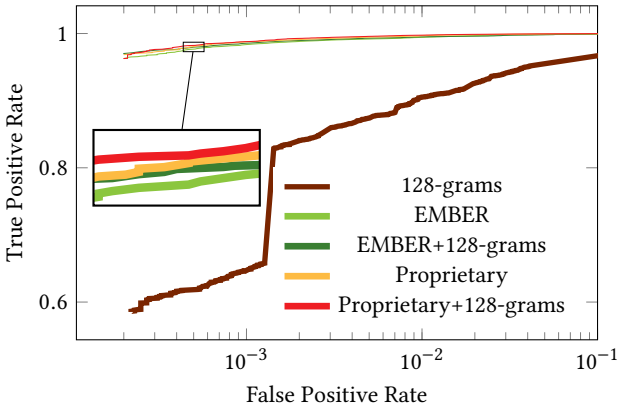

Figure 7: KiloGrams augment the EMBER features to create a model that rivals one built using proprietary features.

While the EMBER dataset has been noted as a relatively "easy" dataset [4], the results are a positive indicator of the utility of large n-grams in conjunction with domain knowledge features. Based on these promising results, more extensive work is being prepared to test KiloGram augmented production features on industry-representative datasets.

6 CONCLUSION

We have introduced the KiloGram algorithm for computing the top-k most frequent n-grams. It is over 60x faster than prior approaches for small n, and allows the new capability to select $n \geq 1024$ with no increase in runtime. This allows us to explore previously unanswerable questions about large n-grams for malware classification. More careful consideration about the nature of such large n-grams allows us to address several issues in real-life malware analysis work. We can create new features that are interpretable and increase analyst trust, automate the creation of signatures with greater precision and recall than was previously possible, and enhance the detection rate of production anti-virus malware detectors.

Our belief is that the introduced ability to use $n \geq 16$ grams will lead to interesting new ways of using n-grams, and may be applicable in natural language processing, network traffic analysis, bioinformatics, and other fields. For malware detection, malware authors trying to evade detection can no longer just obfuscate strings or header fields. They must now consider any potential binary pattern, such as compiler tags or code reuse, increasing the effort for them to operate undetected.

REFERENCES

[1] 2018. VirusTotal-Free online virus, malware and URL scanner. https://www.virustotal.com
[2] Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan. 2004. Detection of New Malicious Code Using N-grams Signatures. *Second annual conference on privacy, security and trust* (2004), 193–196. https://doi.org/10.1109/CMPSAC.2004.1342667
[3] Victor M. Alvarez. 2013. Yara: The pattern matching swiss knife for malware researchers (and everyone else). https://doi.org/yara/

[4] Hyrum S. Anderson and Phil Roth. 2018. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints* (2018). http://arxiv.org/abs/1804.04637

[5] Andrew P. Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition* 30, 7 (1997), 1145–1159. https://doi.org/10.1016/S0031-3203(96)00142-2

[6] Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M Buhmann. 2010. The Balanced Accuracy and Its Posterior Distribution. In *Proceedings of the 2010 20th International Conference on Pattern Recognition (ICPR '10)*. IEEE Computer Society, Washington, DC, USA, 3121–3124. https://doi.org/10.1109/ICPR.2010.764

[7] Christian Buck, Kenneth Heafield, and Bas van Ooyen. 2014. N-Gram Counts and Language Models from the Common Crawl. In *Proceedings of 9th International Conference on Language Resources and Evaluation (LREC 2014)*. 26–31.

[8] Massimo Cafaro, Marco Pulimeno, Italo Epicoco, and Giovanni Aloisio. 2018. Parallel Space Saving on Multi and Many-Core Processors. *Concurrency and Computation: Practice and Experience* 30, 7 (4 2018). https://doi.org/10.1002/cpe.4160

[9] William B. Cavnar and John M. Trenkle. 1994. N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*. 161–175.

[10] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: Reliable Large-scale Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[11] Aaron Clauset, Cosma Rohilla Shalizi, and M E J Newman. 2009. Power-Law Distributions in Empirical Data. *SIAM Rev.* 51, 4 (2009), 661–703. https://doi.org/10.1137/070710111

[12] Jonathan D Cohen. 1997. Recursive Hashing Functions for N-grams. *ACM Trans. Inf. Syst.* 15, 3 (7 1997), 291–320. https://doi.org/10.1145/256163.256168

[13] Janez Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research* 7 (12 2006), 1–30. http://dl.acm.org/citation.cfm?id=1248547.1248548

[14] Leslie Ann Goldberg, Paul W Goldberg, Cynthia A Phillips, and Gregory B Sorkin. 1998. Constructing Computer Virus Phylogenies. *Journal of Algorithms* 26, 1 (1 1998), 188–208. https://doi.org/10.1006/JAGM.1997.0897

[15] A H Ibrahim, M B Abdelhalim, H Hussein, and A Fahmy. 2010. Analysis of x86 instruction set usage for Windows 7 applications. In *Computer Technology and Development (ICCTD), 2010 2nd International Conference on*. 511–516. https://doi.org/10.1109/ICCTD.2010.5645851

[16] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the 18th ACM conference on Computer and communications security - CCS*. ACM Press, New York, New York, USA, 309–320. https://doi.org/10.1145/2046707.2046742

[17] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting Concept Drift in Malware Classification Models. In *26th USENIX Security Symposium (USENIX Security 17)*. {USENIX} Association, Vancouver, BC, 625–642. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/jordaney

[18] Md. Enamul. Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. 2005. Malware phylogeny generation using permutations of code. *Journal in Computer Virology* 1, 1 (2005), 13–23. https://doi.org/10.1007/s11416-005-0002-9

[19] Jeffrey O Kephart, Gregory B Sorkin, William C Arnold, David M Chess, Gerald J Tesauro, and Steve R White. 1995. Biologically Inspired Defenses Against Computer Viruses. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1 (IJCAI'95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 985–996. http://dl.acm.org/citation.cfm?id=1625855.1625983

[20] J Zico Kolter and Marcus A Maloof. 2006. Learning to Detect and Classify Malicious Executables in the Wild. *Journal of Machine Learning Research* 7 (12 2006), 2721–2744. http://dl.acm.org/citation.cfm?id=1248547.1248646

[21] Bo Li, Kevin Roundy, Chris Gates, and Yevgeniy Vorobeychik. 2017. Large-Scale Identification of Malicious Singleton Files. In *7TH ACM Conference on Data and Application Security and Privacy*.

[22] Mohammad M. Masud, Latifur Khan, and Bhavani Thuraisingham. 2008. A scalable multi-level feature extraction technique to detect malicious executables. *Information Systems Frontiers* 10, 1 (3 2008), 33–45. https://doi.org/10.1007/s10796-007-9054-3

[23] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory (ICDT'05)*. Springer-Verlag, Berlin, Heidelberg, 398–412. https://doi.org/10.1007/978-3-540-30570-5{_}27

[24] Ethan Millar, Dan Shen, Junli Liu, and Charles Nicholas. 2006. Performance and Scalability of a Large-Scale N-gram Based Information Retrieval System. *Journal of Digital Information* 1, 5 (2006). https://journals.tdl.org/jodi/index.php/jodi/article/view/22

[25] Abedelaziz Mohaisen and Omar Alrawi. 2013. Unveiling Zeus: Automated Classification of Malware Samples. In *Proceedings of the 22Nd International Conference on World Wide Web (WWW '13 Companion)*. ACM, New York, NY, USA, 829–832. https://doi.org/10.1145/2487788.2488056

[26] Makoto Nagao and Shinsuke Mori. 1994. A new method of N-gram statistics for large number of n and automatic extraction of words and phrases from large text data of Japanese. *Proceedings of the 15th conference on Computational linguistics* 1 (1994), 611–615. https://doi.org/10.3115/991886.991994

[27] Andrew Y. Ng. 2004. Feature selection, L1 vs. L2 regularization, and rotational invariance. *Twenty-first international conference on Machine learning - ICML '04* (2004), 78. https://doi.org/10.1145/1015330.1015435

[28] Adam Pauls and Dan Klein. 2011. Faster and Smaller N-gram Language Models. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1 (HLT '11)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 258–267. http://dl.acm.org/citation.cfm?id=2002472.2002506

[29] Steven T Piantadosi. 2014. Zipf's word frequency law in natural language: a critical review and future directions. *Psychonomic bulletin & review* 21, 5 (10 2014), 1112–30. https://doi.org/10.3758/s13423-014-0585-6

[30] Edward Raff. 2017. JSAT: Java Statistical Analysis Tool, a Library for Machine Learning. *Journal of Machine Learning Research* 18, 23 (2017), 1–5. http://jmlr.org/papers/v18/16-131.html

[31] Edward Raff and Mark McLean. 2018. Hash-Grams On Many-Cores and Skewed Distributions. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 158–165. https://doi.org/10.1109/BigData.2018.8622043

[32] Edward Raff and Charles Nicholas. 2017. Malware Classification and Class Imbalance via Stochastic Hashed LZJD. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security (AISec '17)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/3128572.3140446

[33] Edward Raff and Charles Nicholas. 2018. Hash-Grams: Faster N-Gram Features for Classification and Malware Detection. In *Proceedings of the ACM Symposium on Document Engineering 2018*. ACM, Halifax, NS, Canada. https://doi.org/10.1145/3209280.3229085

[34] Edward Raff, Richard Zak, Russell Cox, Jared Sylvester, Paul Yacci, Rebecca Ward, Anna Tracy, Mark McLean, and Charles Nicholas. 2016. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques* (9 2016). https://doi.org/10.1007/s11416-016-0283-1

[35] D Krishna Sandeep Reddy and Arun K Pujari. 2006. N-gram analysis for computer virus detection. *Journal in Computer Virology* 2, 3 (11 2006), 231–239. https://doi.org/10.1007/s11416-006-0027-8

[36] J-Michael Roberts. 2011. Virus Share. https://virusshare.com/

[37] Florian Roth. 2013. yarGen. https://github.com/Neo23x0/yarGen

[38] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. AVclass: A Tool for Massive Malware Labeling. In *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016*, Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro (Eds.). Springer International Publishing, Paris, France, 230–253. https://doi.org/10.1007/978-3-319-45719-2{_}11

[39] John Seymour and Charles Nicholas. 2016. Labeling the VirusShare Corpus: Lessons Learned. In *BSidesLV*. Las Vegas, NV.

[40] Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev, and Yuval Elovici. 2012. Detecting unknown malicious code by applying classification techniques on OpCode patterns. *Security Informatics* 1, 1 (2012), 1–22. https://doi.org/10.1186/2190-8532-1-1

[41] Eugene C. Spafford. 2014. Is Anti-virus Really Dead? *Computers & Security* 44 (2014), iv. https://doi.org/10.1016/S0167-4048(14)00082-0

[42] Gil Tahan, Lior Rokach, and Yuval Shahar. 2012. Mal-ID: Automatic Malware Detection Using Common Segment Analysis and Meta-features. *Journal of Machine Learning Research* 13 (4 2012), 949–979. http://dl.acm.org/citation.cfm?id=2188385.2343677

[43] Andrew Walenstein, Michael Venable, Matthew Hayes, Christopher Thompson, and Arun Lakhotia. 2007. Exploiting similarity between variants to defeat malware. In *Proc. BlackHat DC Conf.*

[44] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*. ACM Press, New York, New York, USA, 1113–1120. https://doi.org/10.1145/1553374.1553516

[45] Guo-xun Yuan, Chia-Hua Ho, and Chih-jen Lin. 2012. An improved GLMNET for L1-regularized logistic regression. *Journal of Machine Learning Research* 13 (2012), 1999–2030. https://doi.org/10.1145/2020408.2020421

[46] Richard Zak, Edward Raff, and Charles Nicholas. 2017. What can N-grams learn for malware detection?. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 109–118. https://doi.org/10.1109/MALWARE.2017.8323963

[47] George Kingsley Zipf. 1949. *Human behavior and the principle of least effort*. Addison-Wesley Press, Oxford, England. xi, 573–xi, 573 pages.

[48] Hui Zou and Trevor Hastie. 2005. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society, Series B* 67, 2 (4 2005), 301–320. https://doi.org/10.1111/j.1467-9868.2005.00503.x

APPENDIX A SUPPLEMENTARY MATERIAL

A.1 Dataset Source Details

For completeness and reproducibility, we will review the nature of our datasets in greater detail.

The "Industry EXE" dataset was provided by a third party AV company. The training set contains approximately 2 million Windows PE executables, 1,000,020 benign and 1,011,766 malicious, and was used as the training data in [32]. A smaller corpus of 400,000 files from the same AV company as a large test set, which is also evenly split and was used as the training data in[34]. All the executable binaries in this corpus were first seen in the 2014-2015 time frame. This company has asked to remain anonymous, but allowed the use of their older data.

The EMBER corpus [4] normally provides the SHA256 hashes, VirusTotal outputs, and domain knowledge features pre-extracted for the public. The raw files of EMBER were obtained from VirusTotal[1]. This corpus has 600k training and 200k testing files which are evenly split between benign and malicious. The training corpus files contain a small subset from 2016, but the majority (and all test files) are from 2017. The entirety of the EMBER test set has a first-observed date newer than anything in the training data to make a better test of generalization. As mentioned previously, the whole corpus is on average 2-3 years newer than the Industry EXE dataset. It was also collected and organized by a different company, with no collaboration. Using it as a cross-dataset generalization test is then particularly powerful and informative to the longevity of extracted features and models, as we minimize common source bias and generalization occurs after at least 2 years of separation. This is the longest scale test of generalization across time we are aware of in this domain.

Our PDF dataset was constructed from publicly available sources. As such, this corpus may not represent benign and malicious PDF populations in the same way as data collected by AV companies. This is because the AV companies can observe a subset of real-life benign and malicious traffic as they occur on real networks, where our collection from public resources may reflect different sub-populations.

For the malicious PDFs, we downloaded the VirusShare corpus of malware [36], and selected all files in the corpus that were PDFs. The VirusShare dataset is mostly Windows PE data, and so we are only able to collect a total of 157,780 malicious files. For our benign files, we used Common Crawl[4], a non-profit effort to produce a publicly available "crawl" of the internet similar to that used by search engines. We randomly downloaded PDF files indexed in the common crawl to create our benign files. The entire corpus is about 19% malware. We are aware that assuming all files indexed by common crawl are benign may not be absolutely true, but several spot checks did not turn up any obviously malicious files. The inspection of KiloGrams from § 5 also gives us confidence that this is not a systematic problem, as we would otherwise be unlikely to learn such useful and interpretable features from the PDF corpus if contamination did occur.

Our last dataset, VirusShare-20C, is a malware family classification dataset so that we could study a multi-class problem instead of a binary one. This dataset was constructed by again using the public VirusShare corpus [36]. To determine the malware families, we use the VirusTotal [1] Anti-Virus labels provided by [39]. The VirusTotal results include a label from several different Anti-Virus products. Each AV product may use different naming schemes, have conflicts, and sometimes different sets of AVs run against each file. We use AVclass[38] to take the results from VirusTotal and produce a single canonical family name and label for each file. This produced 184 families which each had at least 10,000 samples. 20 of these were selected at random to create our dataset. For each family, 8,000 files were used for the training set and 2,000 for the testing set.

A.2 Additional Figures

Figure 8: Frequently-used malware icon, found by inspecting 64-gram features of Lasso.

[4] http://commoncrawl.org