BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis

Jiyong Jang
Carnegie Mellon University
Pittsburgh, PA, USA
jiyongj@cmu.edu

David Brumley
Carnegie Mellon University
Pittsburgh, PA, USA
dbrumley@cmu.edu

Shobha Venkataraman
AT&T Labs – Research
Florham Park, NJ, USA
shvenk@research.att.com

Abstract

The sheer volume of new malware found each day is growing at an exponential pace. This growth has created a need for automatic malware triage techniques that determine what malware is similar, what malware is unique, and why. In this paper, we present BitShred, a system for large-scale malware similarity analysis and clustering, and for automatically uncovering semantic inter- and intra-family relationships within clusters. The key idea behind BitShred is using feature hashing to dramatically reduce the high-dimensional feature spaces that are common in malware analysis. Feature hashing also allows us to mine correlated features between malware families and samples using co-clustering techniques. Our evaluation shows that BitShred speeds up typical malware triage tasks by up to 2,365x and uses up to 82x less memory on a single CPU, all with comparable accuracy to previous approaches. We also develop a parallelized version of BitShred, and demonstrate scalability within the Hadoop framework.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Design, Performance, Security

Keywords

Malware Triage, Feature Hashing, Co-clustering, Hadoop

1. INTRODUCTION

The volume of new malware, fueled by easy-to-use malware morphing engines, is growing at an exponential pace [8]. In 2009 security vendors received upwards of 8,000 unique by hash malware samples per day [8], with the projected total to reach over 1,000,000 per day within the next 7 years. The sheer volume of malware means we need automatic methods for large-scale malware triage techniques and systems.

At a high level, triage has two steps. First, per-sample malware analysis is run on each sample to extract a set of features. Second, malware are compared in a pairwise fashion to determine similarity, e.g., in our work, like others, using the Jaccard distance. Once we determine what malware are similar, and what are the important semantic similarities and differences to known malware cases, triage can make informed decisions. For example, triage may perform further in-depth analysis on one representative malware sample per family that would be cost-prohibitive to do on the entire data set.

In this paper, we present BitShred, a system for large-scale malware similarity analysis and clustering, and for automatically uncovering semantic inter- and intra-family relationships within clusters. The main feature of BitShred is it is agnostic to the particular per-malware analysis routine, even when the extracted feature set has a very large feature space. Malware authors and defenders are caught in a cyclic battle where defenders invent ever-more advanced and accurate per-malware analyses for feature extraction, which are then defeated by new malware obfuscation algorithms. The cyclic battle brings the need for malware triage techniques that allow us to plug-in the latest or most appropriate analysis for feature extraction. We empirically show BitShred meets the desired requirement by demonstrating BitShred on two previously proposed per-sample analysis: dynamic behavior analysis from Bayer *et al.* [13] where the feature space is 2^{17}, and static code reuse detection as proposed in [9, 25, 38] where the feature space is 2^{128}.

The main issues for handling large volumes of malware are (a) efficiently representing malware features (so we can fit more in main memory without paging), and (b) comparing feature sets between malware, and (c) determining which features are correlated for malware groups. To give a sense of scale, currently over 8000 new malware per day are observed, requiring about 31 million comparisons to find families using hierarchical clustering. If we perform n-item analysis when $n = 16$ bytes, an exact representation of the features would require 2^{128} (2^{95} *gigabytes*) per sample. We could not perform all 31 million comparisons on previous data structures in 24 hours on a single CPU.

The central idea in BitShred is to use feature hashing [36, 37, 39]. Feature hashing allows for dramatic dimensionality reduction, so the hashed representation takes less room in memory, and is also L1/L2 cache efficient. The catch is that feature hashing introduces collisions in the reduced feature space. For example, we use a hash function that compresses the 2^{128} feature space down to 2^{18}, there will be an enormous number of collisions in the feature space. The surprising thing is that with feature hashing, we need just a single hash function for the dimensionality reduction of the feature space. Requiring only single hash function as well as dimensionality reduction have immediate performance implications. This result is backed by theory and experimentation that shows pairwise comparison, thus algorithms built on top like hierarchical clustering, will be close to exact.

Feature hashing allows us to also simultaneously mine correlated features between malware families and samples using co-clustering techniques. Clustering alone acts like a blackbox, telling us only that malware are grouped because they are similar. Co-clustering goes one step further and tells us why malware are similar by simultaneously clustering features. For example, co-clustering allows us to group two malware and to say the features that explain why they are similar (e.g., a significant amount of shared code) and why they are different (e.g., contacting different command and control hosts).

Contributions. Our main contribution is a system for performing the triage tasks described above that scales to data sets orders of magnitude larger than existing approaches. We present a theoretical analysis showing that feature hashing with the Jaccard offers near optimal results, and build a real system called BitShred that is independent of the particular per-malware analysis engine and works even for high-dimensional feature sets. We extensively evaluate BitShred's scalability, speed and accuracy using two different per-sample analysis: code similarity and dynamic behaviors. Our performance evaluation shows that BitShred can cluster over 116,000 malware per day on a single node, and over 1.9 million per day on a Hadoop cluster where we develop an optimal schedule that minimizes communication overhead and provides uniform node work. We also propose novel techniques based upon co-clustering adapted to BitShred for identifying similar or distinguishing semantic features in malware families.

2. THE CORE IDEA, FEATURE HASHING, AND THE RELATION TO PREVIOUS APPROACHES

We focus on any analysis that outputs a set of features that are Boolean, or can be encoded as Boolean variables. For example, in code reuse detection the features are whether a code fragment is present or not. Real-value features can be encoded by bucketizing them, where a Boolean feature is true if the feature falls within a particular bucket. This allows us to plug in many types of analysis.

2.1 Feature Hashing & Malware Similarity

We compute malware similarity using the Jaccard similarity metric. The Jaccard calculates the percentage of common features, with the idea that the larger the sharing is, the more alike the malware are, and is used extensively in previous work [13, 32]. More formally, given two feature sets g_a and g_b for malware s_a and s_b respectively, the Jaccard similarity (i.e., index) is:

$$J(g_a, g_b) = \frac{|g_a \cap g_b|}{|g_a \cup g_b|} \tag{1}$$

In order to motivate feature hashing, consider first using a standard implementation of Jaccard, e.g., as found in SimMetrics [5]. The advantage of this approach is the size of the feature data structure is linear in the number of features a malware sample actually presents, e.g., if our feature space is of size 2^{128} but a particular malware only has 2^{30} features, the data structure is still only 2^{30} in size. Unfortunately, set operations are not amenable to feature extraction using co-clustering, and the set union and intersection operations themselves are a bottleneck. In our experiments, we could only cluster about 2,388 malware/day using this approach (§ 5, labeled as exact Jaccard).

We take an approach of encoding features as a bitvector. The Jaccard becomes fast CPU-friendly logic operations:

$$J_{bv}(f_a, f_b) = \frac{S(f_a \wedge f_b)}{S(f_a \vee f_b)} \tag{2}$$

where f_i is the bitvector representation of the feature set for malware s_i and $S(\cdot)$ counts the number of set bits.

Feature hashing [36, 37, 39] is a specific way of encoding features as a bitvector. Most existing implementations of bitvector Jaccard, e.g., the one found in python, assume the feature space is completely encoded using index variables where feature 1 corresponds to bit 1, feature 2 to bit 2, feature 3 to bit 3, and so on. This scheme is impractical when the feature space is large, e.g., as in our case where such an encoding would result in a per-malware data structure that is gigabytes in size.

A more efficient encoding is to use Bloom filters, which we initially tried. A Bloom filter is a probabilistic data structure used

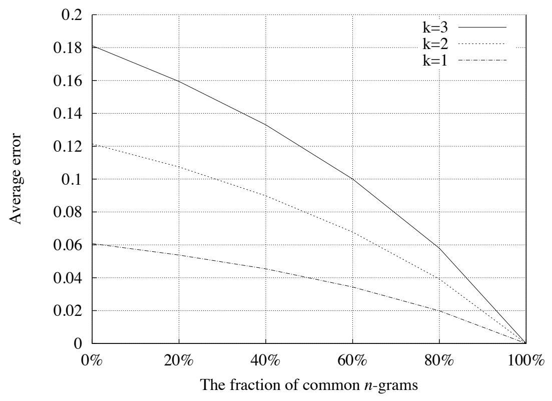


Figure 1: Error with various k where m=8,192

to efficiently encode sets and perform set membership tests. Let $h_1, h_2, h_3, ..., h_k$ be a set of hash functions of type $D \rightarrow R$ and $|D| \gg |R|$, i.e., each hash is a compression function. A Bloom filter calculates $h_i(x) = d$ and sets the d'th bit in the m-length bitvector for all hash functions h_i and each feature value x. To test if an element x' is in the feature set, you check that the $h_i(x')$ bit is set for all i. If any are not set, then x' is not in the set. Bloom filters have false positives due to hash collisions, but never have a false negative. The false positive rate is reduced, all things being equal, by adding more hash functions.

Bloom filters did not work well. The catch in our problem setting is we want to approximate the Jaccard, not perform set membership tests. We had naively estimated the error rate to be the expected Bloom filter collisions in the numerator divided by the expected number of Bloom filter collisions in the denominator, which is mathematically unsound: dividing two expected values will not provide the proper expectation of the Jaccard expression.

Feature hashing is similar to Bloom filters where we compute $h(x) = d$ and set the d'th bit, except that only a single hash function is used. We call the hashed version of features the *malware fingerprint*. Theorem 1 (§ 3.1) shows that the malware fingerprints provide a near-optimal approximation of the true Jaccard index. To the best of our knowledge, no previous work (e.g., [15]) has performed similar analysis. Indeed, the proof shows that increasing the number of hash functions increases error, which is why Bloom filters don't work well. This corresponds well to feature hashing, where only one hash function is used. Further, requiring only one hash has obvious performance improvement implications.

We also showed through simulations on random sets of n-grams, the use of single hash function minimized the difference between the Jaccard in Equation 1 and the bitvector Jaccard in Equation 2. In particular, we created two sets that contain 1000 n-grams each, with varying number of overlapping n-grams, and measured how much the bitvector Jaccard differs from the Jaccard. Figure 1 shows the average error as the fraction of common n-grams and the number of hash functions k vary (the standard deviation is very small and therefore, not shown). We note that the error increases as k increases, with minimum error achieved at $k = 1$, which is different from the usual Bloom filter set membership tests.

The theoretical and empirical analysis motivates feature hashing in BitShred. The main quality metric for our approach is how well our approach approximates a full set representation vs performance improvements. We show through numerous experiments that our approach has extremely high accuracy with up to 2,365 times speedup on a single CPU.

2.2 Co-clustering in BitShred

A BitShred fingerprint is a m-length bitvector where the intuition is a bit i is 1 if the particular malware sample has a feature g_i, and 0

$$\mathbf{M} = \begin{vmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{vmatrix} \Rightarrow \left|\begin{array}{ccc|cc} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{array}\right| = \mathbf{M}'$$

Figure 2: $\mathbf{M}$ is co-clustered to identify the checkerboard sub-matrix $\mathbf{M}'$ of highly correlated malware/feature pairs.

otherwise. Given n malware samples, the m-sized list of BitShred fingerprints can be viewed as a matrix $\mathbf{M}$ of size $n \times m$ where each row is a malware fingerprint, and each column is a particular feature. This intuition leads us to the idea of using co-clustering (aka bi-clustering) to auto-correlate both features and malware simultaneously. Within the matrix, co-clustering does this by creating sub-matrices among columns (features) and rows (malware) where each sub-matrix is a highly correlated malware/feature pair.

Co-clustering allows us to discover substantial, non-trivial structural relationships between malware samples, many of which will not be discovered with simpler approaches. For example, consider how the following simple approaches for mining features between two malware families would be limited:

- Identify all common features between families. In BitShred, this is accomplished by taking the bitwise-and ($\wedge$) of the malware fingerprints. However, we would miss identifying code that is present in 99% of family 1 and 99% of family 2.
- Identify all distinctive features in a list of malware. In our setting, this is accomplished with bitwise xor ($\oplus$) of the fingerprints. This would have limited value for the same reasons as above.
- A third approach might be to cluster features either *before* or *after* the malware fingerprints have been clustered. Note, however, this approach would also result in misleading information, e.g., clustering features *after* the clustering malware fingerprints would not reveal structural similarity across fingerprints in different families, and clustering features *before* the malware fingerprints may result in poor malware clusters if there are many feature clusters that are common to multiple groups of malware fingerprint clusters.

We introduce some terminology to make co-clustering precise. A matrix is *homogeneous* if the entries of the matrix are similar, e.g., they are mostly 0 or mostly 1, and define the *homogeneity* of a matrix to be the (larger) fraction of entries that have the same value. Define a *row-cluster* to be a subset of the rows M (i.e., malware samples) that are grouped together, and a *column-cluster* to be a subset of the columns (i.e., the features) that are grouped together. The goal of co-clustering is to create a pair of row and column labeling vectors:

$$\mathbf{r} \in \{1, 2, ..., k\}^n \quad \text{and} \quad \mathbf{c} \in \{1, 2, ..., \ell\}^m$$

The sub-matrices created are homogeneous, rectangular regions. The number of rectangular regions is either given as input to the algorithm, or determined by the algorithm with a penalty function that trades off between the number of rectangles and the homogeneity achieved by these rectangles [1].

For example, Figure 2 shows a list of 5 malware BitShred fingerprints where there are 5 possible features. The result is the 5×5 matrix M. Co-clustering automatically identifies the clustering to produce sub-matrices, as shown by the checkerboard M'. The sub-matrices are homogeneous, indicating highly-correlated feature/-malware pairs. In this case the labeling vectors are $\mathbf{r} = (12122)^T$ and $\mathbf{c} = (21121)^T$. These vectors say that row 1 in $\mathbf{M}$ mapped to row cluster 1 (above the horizontal bar) in **M'**, row 2 mapped to row cluster 2 (below the horizontal bar), etc., and similar for the column vectors for features. We can reach two clustering conclusions. First, the row clusters indicate malware s_1 and s_3 are in one family, and s_2, s_4, and s_5 are in another family. The column clusters say the distinguishing features between the two families are features 2, 3, and 5.

[1]The goal is to make the minimum number of rectangles which achieve the maximum homogeneity. For this reason, co-clustering algorithms ensure that homogeneity of the rectangles is penalized by the number of rectangles if they need to automatically determine k and ℓ.

2.3 Related Approaches

Feature Hashing and Locality Sensitive Hashing. Our main goal is to increase scalability in malware comparison, as well as enable data mining on co-occurring features. We are not alone. Bayer *et al.* [13] has made significant strides in this area by using locality sensitive hashing (LSH) [10]. The main idea in LSH is to define a hash function h such that $h(s_1) = h(s_2)$ if the two malware s_1 and s_2 are similar. The hash is run over all malware samples, and only those with unique hash values are compared. LSH is complementary to feature hashing, because it reduces the number of items while feature hashing reduces the number of features. While complementary, our evaluation shows that using feature hashing alone outperforms LSH alone by a factor of 2-to-1. Previous work has shown this bears on theoretical analysis as well [36]. While in our evaluation we focus on the effects of each algorithm independently, both feature hashing and locality-sensitive hashing could be combined in a real system.

Feature Hashing and graph analysis comparison. Others have proposed malware similarity methods that do not use boolean features. For example, the Zynamics BinDiff tool [7] and Hu *et al.* [21] use a similarity metric based upon isomorphisms between control flow and function call graphs. While we can compute call graph similarity based upon features, e.g., how many basic blocks are in common, our approach cannot readily be adapted to actually computing the isomorphism. Hu *et al.* argue that although graph-based isomorphism is expensive, it is less susceptible to being fooled by polymorphism. In Hu *et al.* 's implementation they return the 5 nearest neighbors, and achieve an 80% success rate in having 1 of the 5 within the same family on a data set of 102,391 samples. The query time was between 0.015s to 872s, with an average of 21s using 100MB of memory. We did not have access to their data set; results for the same size of our data set for finding nearest neighbor using the Jaccard are reported in § 5.1.

Classification vs. Clustering. Classification uses *labeled* samples to learn a rule for assigning labels to new samples. Feature hashing was previously used to build an efficient spam classifier [11], which is trained with labeled (i.e., spam/not-spam) emails and then determines whether an incoming email is spam or not. On the contrary, clustering groups *unlabeled* samples based on given similarity metrics. In our setting, (unlabeled) malware are grouped based upon similar features – static code or dynamic behaviors. To the best of our knowledge, ours is the first study to introduce clustering techniques combining feature hashing with the Jaccard and to present a theoretical proof of correctness.

2.4 Security and Rest of Paper

Security. Feature hashing, like LSH, uses a hash function which must be kept secret. If the hash function is known, then an attacker may be able to "fool" the algorithm into an atypical number of collisions, thus potentially reducing the overall accuracy. This problem is mitigated by using a keyed hash function as is usual, e.g., picking a secret key k and computing $h(k||s_a||k)$ for item s_a. For simplicity, in the rest of this paper we refer to the hash as simply h

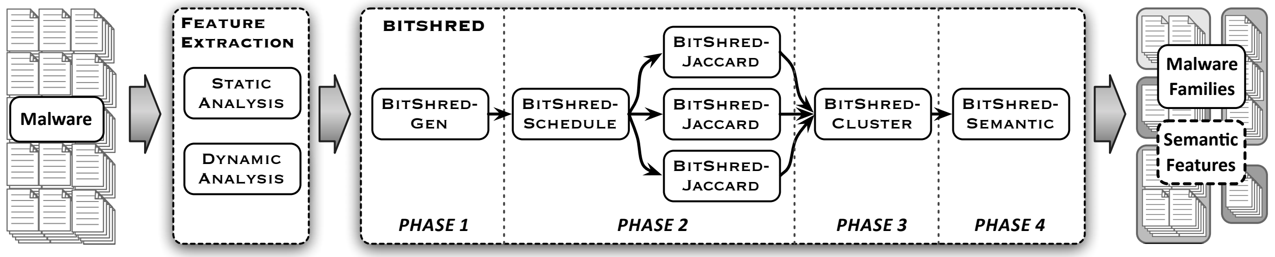


Figure 3: BitShred Overview

instead of a keyed variant, with the expectation it is used as a keyed function for security.

In the rest of this paper we describe BitShred, our system for malware triage and semantic feature identification on very large malware data sets based on the above ideas. We focus on the performance of our hash feature approach vs. previously proposed methods such as straight set-based analysis, winnowing, and locality sensitive hashing. The main conclusion is that BitShred provides the same accuracy but better performance. Better performance means we scale to much larger malware volumes, as well as deal with current volumes much more quickly. We also show that BitShred can be parallelized, which allows us to take advantage of infrastructures like supercomputers and Hadoop as performance requirements exceed that which can be provided by a single CPU.

Finally, we show BitShred in the context of an end-to-end system for malware triage. In our data set BitShred has above 90% accuracy at automatically identifying malware families and semantic features. While malware authors can always add more obfuscation and make analysis harder, thus decreasing accuracy of any system, the core concepts in BitShred can "plug-in" any malware analysis that outputs Boolean or (binary-encoded) integer-valued values, and speed it up while retaining similar accuracy to the exact Jaccard.

3. BITSHRED OVERVIEW

At a high level, BitShred takes in a set of malware, runs per-malware analysis, and then performs inter-malware comparison, correlation, and feature analysis, as shown in Figure 3. BitShred's job is to speed up subsequent correlation after using existing techniques to perform per-sample feature extraction. In our implementation, we experiment with using n-grams as proposed in [9, 25, 38] because the feature space is extremely large, and dynamic behavior analysis from Bayer *et al.* [13] because it has been shown effective.

Throughout the rest of this paper we use s_i to denote malware sample i, G to denote the set of all features, and g_i to denote the subset of all features G present in s_i.

We use full hierarchical clustering as a representative computationally expensive triage task. Hierarchical clustering has a lower bound of $s(s-1)/2$ comparisons for s malware to cluster [19]. Other problems, such as incremental clustering and finding nearest neighbor, are algorithmically less expensive. For example, incremental clustering, comparing incoming newly reported s' malware against s malware in a database, requires $s' \times s$ comparisons where $s' \ll s$. The nearest neighbor to a malware s_i can be performed by comparing it to all other samples, which is linear in s.

3.1 Single Node BitShred

In this section we describe the core components of BitShred: BITSHRED-GEN, BITSHRED-JACCARD, BITSHRED-CLUSTER and BITSHRED-SEMANTIC. In § 3.2, we show how the algorithm can be parallelized, e.g., to run on top of Hadoop or multi-core systems.

• **BITSHRED-GEN:** $G \to F$.

BITSHRED-GEN is an algorithm from the extracted feature set $g_i \in G$ to fingerprints $f_i \in F$ for each malware sample s_i. A BitShred fingerprint f_i is a bit-vector of length m, initially set to 0. BITSHRED-GEN performs feature hashing to represent feature sets g_i in fingerprints f_i. More formally, for a particular feature set we define a hash function $h : \chi \to \{0,1\}^m$ where the domain χ is the domain of possible features and m is the length of the bitvector. We use djb2 [14] and reduce the result modulo m. (Data reduction techniques such as locality-sensitive hashing [13] and Winnowing [34] can be used to pare down the data set for which we call BITSHRED-GEN and perform subsequent steps.)

• **BITSHRED-JACCARD:** $F \times F \to \mathbb{R}$.

BITSHRED-JACCARD computes the similarity $d \in [0,1]$ between fingerprints f_a and f_b using the bitvector Jaccard from Equation 2. A similarity value of 1 means the two samples are identical, while a similarity of 0 means the two samples share nothing in common (in our setting, this means they share no features in G).

Formally, Theorem 1 states that BITSHRED-JACCARD well-approximates the Jaccard index.

Theorem 1. Let g_a, g_b denote two sets of size N with c common elements, and f_a, f_b denote their respective fingerprints with bit-vectors of length m and k hash functions. Let Y denote $\frac{S(f_a \wedge f_b)}{S(f_a \vee f_b)}$. Then, for $m \gg N$, $\epsilon, \epsilon_2 \in (0,1)$,

$$Pr[Y \leq \frac{c(1+\epsilon_2)}{2N - c - m\epsilon}] \geq 1 - e^{-mq\epsilon_2^2/3} - 2e^{-2\epsilon^2 m^2/Nk}$$

and

$$Pr[Y \geq \frac{c(1-\epsilon_2)}{(2N - c) + m\epsilon}] \geq 1 - e^{-mq\epsilon_2^2/2} - 2e^{-2\epsilon^2 m^2/Nk}$$

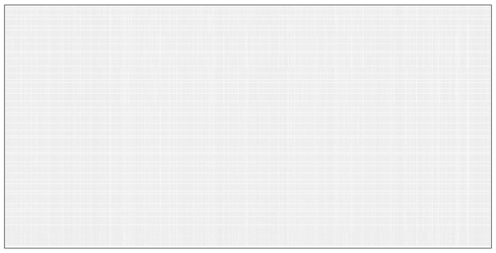
for $q = 1 - 2\left(1 - \frac{1}{m}\right)^{kN} + \left(1 - \frac{1}{m}\right)^{k(2N-c)}$.

We defer a full proof to the Appendix. Note that because the goal of feature hashing is different from Bloom filters, our guarantees are not in terms of the false positive rate for standard Bloom filters, but instead are of how well our feature hashing data structure lets us approximate the Jaccard index.

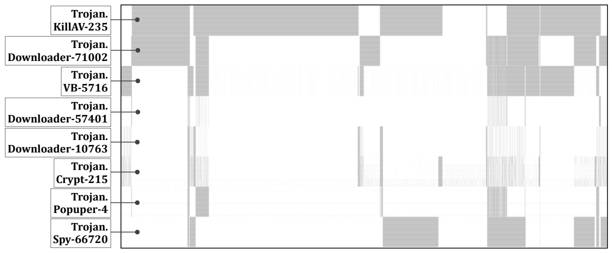
• **BITSHRED-CLUSTER:** $(F \times F \times \mathbb{R}$ **list**$) \times \mathbb{R} \to C$.

BITSHRED-CLUSTER takes the list containing the similarity between each pair of malware samples, a threshold t, and outputs a clustering C for the malware. BITSHRED-CLUSTER groups two malware if their similarity d is greater than or equal to t: $d \geq t$. The threshold t is set by the desired precision tradeoff based upon past experience. While a smaller t divides malware into a few general families, a larger t discovers specific variants of a family. See § 5 for our experiments for different values of t.

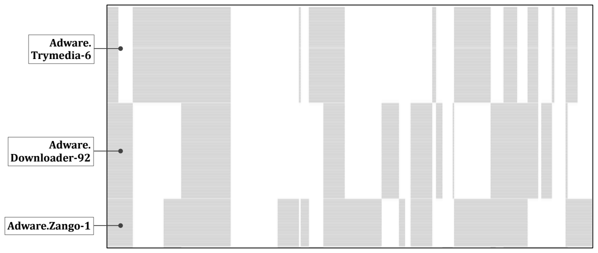
BitShred currently uses an agglomerative hierarchical clustering algorithm to produce clusters in that the number of clusters is typically not known in advance. Initially each malware sample s_i is assigned to its own cluster c_i. The closest pair is selected and merged into a cluster. We iterate the merging process until there is no pair whose similarity exceeds the input threshold t. When there are multiple samples in a cluster, we define the similarity between cluster c_A and cluster c_B as the maximum similarity between all possible pairs (i.e., single-linkage), i.e., BITSHRED-JACCARD(c_A, c_B) =



(a) A typical matrix before co-clustering



(b) 8 different kinds of Trojan



(c) 3 different kinds of Adware

Figure 4: Semantic feature information. Grey dots represent 1 in the binary matrix, i.e., the presence of a feature.

$\max\{\text{BITSHRED-JACCARD}(f_i, f_j) | f_i \in c_A, f_j \in c_B\}$. We chose a single-linkage approach as it is efficient and accurate in practice.

• **BITSHRED-SEMANTIC:** $C \times F \rightarrow G'$.

Based on the BITSHRED-CLUSTER results, BITSHRED-SEMANTIC performs co-clustering on subset of fingerprints to cluster features as well as malware samples. Co-clustering yields correlated features-malware subgroups G' which shows the common or distinct features among malware samples, as discussed in § 2.2.

We have adapted the cross-associations algorithm [16], redesigned for the Map-Reduce framework [30], to BitShred fingerprints. The basic steps are row iterations and column iterations. A row iteration fixes a current column group and iterates over each row, updating r to find the "best" grouping. In our algorithm, we seek to swap each row to a row group that would maximize homogeneity of the resulting rectangles. The column iteration is similar, where rows are fixed. The algorithm performs a local optimal search (finding a globally optimal co-clustering is NP-hard [30]).

Unlike typical co-clustering problems, co-clustering in BitShred needs to operate on hashed features, i.e., recall that our fingerprints are not the features themselves, but hashes of these features. However, because our feature hashing is designed to approximately preserve structural similarities and differences between malware samples, we can apply co-clustering on our hashed features (just as if they were regular features) and still extract the structural relationships between the malware samples, and with the increased computational efficiency that comes from feature hashing.

Figure 4a shows a matrix before co-clustering where each row is a malware fingerprint and each column is a particular feature. Figure 4b and Figure 4c graphically depict the results of co-clustering on 8 different kinds of Trojan and 3 different kinds of Adware, respectively. Co-clustering reveals the semantic feature information, i.e., the checkerboard patterns which describe distinguishing or common features across the malware families. We discuss both inter- and intra-family feature extraction in detail in § 5.3.

3.2 Distributed BitShred

3.2.1 BITSHRED-SCHEDULE

There are two things we parallelize in BitShred: fingerprint generation in Phase 1, and the $s(s-1)/2$ fingerprint comparisons in Phase 2 during clustering. Parallelizing fingerprint generation is straight-forward: given s malware samples and r resources, we assign s/r malware to each node and run BITSHRED-GEN on each assigned sample.

Parallelizing BITSHRED-JACCARD in a resource and communication-efficient manner requires more thought. There are $s(s-1)/2$ comparisons, and every comparison takes the same fixed time, so if every node does $s(s-1)/2r$ comparisons all nodes do equal work. Unlike BITSHRED-JACCARD, comparisons between variable length of two sets take time (computation) depending on the length, thus distributing uniform node work is not simple.

To accomplish this we first observe that while the first malware needs to be compared against all other malware (i.e., $s-1$ fingerprint comparisons), each of the remaining malware require fewer than $s-1$ comparisons each. In particular, malware i requires only $s-i$ comparisons, and malware $s-i$ requires $s-(s-i)$ comparisons. The main insight is to pair the comparisons for malware i with $s-i$, so that the total comparisons for each pair is $s-i+s-(s-i) = s$. If we pair together the comparisons for malware i with $s-i$, the total comparisons for each pair is $s-i+s-(s-i) = s$. Thus, for each node to do uniform work, BITSHRED-SCHEDULE ensures that the $s-i$ comparisons for malware i are scheduled on the same node as the $s-(s-i)$ comparisons for malware $s-i$. BITSHRED-SCHEDULE then simply divides up the pairs among the r nodes.

Although there may be other protocols for distributing computations, we note that this approach is simple, and optimal in the sense that there all nodes do equal work and there is no inter-node communication during the s^2 Jaccard calculations.

3.2.2 BitShred on Hadoop

Our distributed implementation uses the Hadoop implementation of MapReduce [1, 17]. MapReduce is distributed computing technique for taking advantage of a large computer nodes to carry out large data analysis tasks. In MapReduce, functions are defined with respect to $\langle$key,value$\rangle$ pairs. MapReduce takes a list of $\langle$key,value$\rangle$ pairs, and returns a list of values. MapReduce is implemented by defining two functions:

1. MAP: $\langle K_i, V_i \rangle \rightarrow \langle K_o, V_o \rangle$ list. In the MAP step the master Hadoop node takes the input pair of type $\langle K_i, V_i \rangle$ and partitions into a list of independent chunks of work. Each chunk of work is then distributed to a node, which may in turn apply MAP to further delegate or partition the set of work to complete. The process of mapping forms a multi-level tree structure where leaf nodes are individual units of work, each of which can be completed in parallel. When a unit of work is completed by a node, the output $\langle K_o, V_o \rangle$ is passed to REDUCE.
2. REDUCE: $\langle K_o, V_o \rangle$ list $\rightarrow V_f$ list. In the REDUCE step the list of answers from the partitioned work units are combined and assembled to a list of answers of type V_f.

We also take advantage of the Hadoop distributed file system (HDFS) to share common data among nodes.

In phase 1, distributed BitShred produces fingerprints using the Hadoop by defining the following MapReduce functions:

1. MAP: $\langle K_i, s_i\rangle$ list $\rightarrow$ $\langle K_i, f_i\rangle$ list. Each MAP task is assigned the subset of malware samples s_i and creates fingerprints f_i to be stored on HDFS. Fingerprint files are named as K_i representing the index to the corresponding malware samples.
2. REDUCE. In this step, no REDUCE step is needed.

In phase 2, distributed BitShred runs BITSHRED-JACCARD across all Hadoop nodes by defining the following functions:

1. MAP: $\langle K_i, f_i\rangle$ list $\rightarrow \langle\mathbb{R}, (s_a, s_b)\rangle$ list MAP tasks read fingerprint data files created during phase 1 and runs BITSHRED-JACCARD on each fingerprint pair, outputting the similarity $d \in \mathbb{R}$.
2. REDUCE: $\langle\mathbb{R}, (s_a, s_b)\rangle$ list $\rightarrow$ sorted $\langle\mathbb{R}, (s_a, s_b)\rangle$ list REDUCE gathers the list of the similarity values for each pair and returns a sorted list of pairs based upon similarity.

This phase returns a sorted list of malware pairs by similarity using standard Hadoop sorting. The sorted list is essential for the agglomerative single linkage clustering. In particular, malware s_i's family is defined as the set of malware whose distance is less than θ, thus all malware in the sorted list with similarity $> \theta$ are in the cluster.

4. IMPLEMENTATION

We have implemented single-node BitShred in 2000 lines of C code. Since BitShred is agnostic to the particular per-malware analysis methods, we only need individualized routines for extracting raw input features, before converting into fingerprints. In case of static code analysis, BitShred divides executable code section identified by `GNU BFD` library into n-grams and hashes each n-gram to create fingerprints. For dynamic behavior analysis, BitShred simply parses input behavior profile logs and hashes every behavior profile to generate fingerprints. We use `berkeley DB` to store and manage fingerprints database. After building the database, BitShred retrieves fingerprints from the database to calculate the Jaccard similarity between fingerprints. After applying an agglomerative hierarchical clustering algorithm, malware families are formed. We use graphviz and Cluto [24] for visualizing the clustering and family trees generated as shown in Figure 10, 11.

Distributed BitShred is implemented in 500 lines of Java code. We implemented a parser for extracting section information from Portable Executable header information because there is no `BFD` library for Java. In our implementation, we perform a further optimization that groups several fingerprints into a single HDFS disk block in order to optimize I/O. In the Hadoop infrastructure we use, the HDFS block size is 64MB. We optimize for this block size by dividing the input malware set so each node works on 2,048 malware samples at a time because 64MB = 32KB $\times$ 2048. That is, each MAP task is given 2,048 samples $(s_i, s_{i+1}, \cdots s_{i+2047})$ and generates a single file containing all fingerprints. We can similarly optimize for other block sizes and different bit-vector lengths, e.g, 64KB bit vectors result in batching 1,024 malware samples per node.

Distributed co-clustering is implemented in 1200 lines of Java code. We implemented a python-wrapper to iterate row and column operations to find an optimal co-clustering.

5. EVALUATION

We have evaluated BitShred for speed and accuracy using two types of per-sample analysis for features. First, we use a static code reuse detection approach where features are code fragments, and two malware are considered similar if they share common code fragments. Second, we use a dynamic analysis feature set where features are displayed behaviors, and two malware are considered similar if they have exhibit similar behaviors. Note that similarity is a set comparison, so order does not matter (e.g., re-ordering basic blocks is unlikely to affect the results). We stress that we are not advocating a particular approach such as static or dynamic analysis, but instead demonstrating how BitShred could be used once an analysis was selected.

Equipment. All single-node experiments were performed on a Linux 2.6.32-23 machine (Intel Core2 6600 / 4GB memory) using only a single core. The distributed experiments were performed on a Hadoop using 64 worker nodes, each with 8 cores, 16 GB DRAM, 4 1TB disks and 10GbE connectivity between nodes [2]. 53 nodes had a 2.83GhZ E5440 processor, and 11 had a 3GhZ E5450 processor. Each node is configured to allow up to 6 map tasks and up to 4 reduce tasks at one time.

Malware Dataset. We performed our experiments on a malware data set collected from a variety of open repositories such as Malware Analysis System (aka CWSandbox) [3], Offsensive Computing [4], and from our Universities infrastructure-wide security infrastructure inbetween 2009-2010. Our total data set consists of 655,360 unique samples by MD5 hash.

5.1 BitShred with Code Reuse as Features

Setup. Our static experiments are based upon reports that malware authors reuse code as they invent new malware samples [9, 25, 38]. Since malware is traditionally a binary-level analysis, not a source analysis, our implementation uses n-grams to represent binary code fragments. Malware similarity is determined by the percentage of n-grams shared.

We chose n-grams based analysis because it is one previously proposed approach that demonstrates a high dimensionality feature space. We set n = 16, so there are 2^{128} possible n-gram features. We chose 16 based upon experiments that show it would cover at least a few instructions (not shown for space reasons). We can use other features such as basic blocks, etc. as well by first building the appropriate feature and then defining a hash function on it; all possible extensions of the per-sample analysis is out of scope for this work. Surprisingly, even this simple analysis had over 90% accuracy when the malware is unpacked using off-the-shelf unpackers. Pragmatically, n-gram analysis also has the advantage of not requiring disassembling, building a control flow graph, etc., all of which are known hard problems on malware.

Single Node Performance. Table 1 shows BitShred's performance using a single node in terms of speed, memory consumed, and the resulting error rate. We limited our experiment to clustering 1,000 malware samples (which requires 499,500 pairwise comparisons) in order to keep the exact Jaccard time reasonable. The "exact Jaccard" row shows the overall performance when computing the set operations as shown in Equation 1 using the SimMetrics library [5]. Clustering using exact Jaccard took more than 4 hours, and required 644.13MB of memory. This works out to about 33 malware comparisons/sec and 2,388 malware clustered per day.

We performed two performance measurements with BitShred: one with 32KB fingerprints and one with 64KB fingerprints. With 64KB fingerprints, BitShred ran about 317 times faster than exact Jaccard. With 32KB fingerprints, BitShred runs about 2 times faster compared to 64KB fingerprints, and about 631 times faster than exact Jaccard.

Since BitShred uses feature hashing, hash collisions may impact the accuracy of the Jaccard distance computations. The overall error rate in the distance computations is a function of the fingerprint length, the size of the feature space, and the percentage of code that is similar. The statement in Theorem 1 formally expresses this tradeoff. We also made two empirical measurements. First, we computed the average error on all pairs, which worked out to be about 2% with 64KB fingerprints and 4% with 32KB fingerprints.

	Size of fingerprints	Time to compare every pair	Average error on all pairs	Average error on similar (>0.5) pairs	Malware comparisons per second	Malware clustered per day
EXACT JACCARD	644.13MB	4h 12m 16s	-	-	33	2,388
BS64K	62.50MB	48s	0.0199	0.0017	10,472	42,538
BS32K	31.25MB	24s	0.0403	0.0050	20,812	59,970
WINNOW (W4)	66.97MB	41m 5s	0.0019	0.0109	203	5,918
WINNOW (W12)	30.16MB	20m 35s	0.0081	0.0128	404	8,360
BS32K (W4)	31.25MB	24s	0.0159	0.0009	20,812	59,970
BS32K (W12)	31.25MB	24s	0.0062	0.0039	20,812	59,970
BS8K (W4)	7.81MB	6s	0.0649	0.0086	78,047	116,131
BS8K (W12)	7.81MB	6s	0.0247	0.0016	78,047	116,131

Table 1: BitShred (BS) vs. Jaccard vs. Winnowing. We show BitShred with several different fingerprint sizes.

The error goes up as the fingerprint size shrinks because there is a higher chance of collisions. We also computed the average error on pairs with a similarity of at least 50%, and found the error to be less than 1% of the true Jaccard. Note that the second metric (i.e., average error on pairs with higher similarity), is the more important metric – these are the numbers with the most impact on the accuracy, as these are the numbers that will primarily decide which family a malware sample belongs to. Thus, BitShred is a very close approximation indeed.

BitShred vs. Winnowing. In this paper so far we have considered techniques that provide an exact ranking between all pairs of malware. Nonetheless, malware practitioners are constantly facing hard choices on how much time to spend given finite computing resources, thus may want faster but approximate over theoretically correct but slower clustering. LSH is one type of data reduction technique that improves performance. Here we discuss another called *Winnowing*.

Winnowing, the algorithm used by the MOSS plagiarism detection tool, is a fuzzing hashing technique that selects a subset of features from a sample for analysis [34]. Let w be a window measured in some way, e.g., w statements, w consecutive n-grams, w behaviors, etc. Winnowing guarantees at least one shared unit in any window of length at least $w + n - 1$ will be included in the feature set [34]. In our evaluation we measure Winnowing because a) MOSS is well-known, and b) it corresponds to similarity detection based upon code as proposed in previous work [9, 25, 38], thus is directly related to our approach, and c) it is guaranteed to be within 33% of an upper bound on performance algorithms for similarity detection [34]. We compared in two settings: BitShred vs. Winnowing as in previous work, and BitShred extended to include Winnowing. Table 1 also shows these results for window sizes 4 (denoted as W4) and 12 (denoted as W12).

BitShred beats straight Winnowing. We reimplemented Winnowing as detailed in [34] using a 32-bit hash function as the original implementation is not public. For the purpose of performance comparison, we computed the similarity using SimMetrics library. BitShred is anywhere from 26-102 times faster, while requiring less memory. Winnowing does have a slightly better error rate, though none of the error rates is very high. A more interesting case is to consider pre-processing the feature set with Winnowing and then applying BitShred. With Winnowing applied, we can reduce the BitShred fingerprint size down to 8KB, allowing all 1,000 samples to be clustered in 6 seconds.

Figure 5 relates all experiments with respect to the total number of malware clustered per day. Recall there are about 8,000 new malware found per day. BitShred deals easily with current volumes, and has room to spare for future growth. Figure 5 also shows on the right-hand y-axis one reason BitShred is faster. Recall we mentioned exact Jaccard computations are slow in part because they

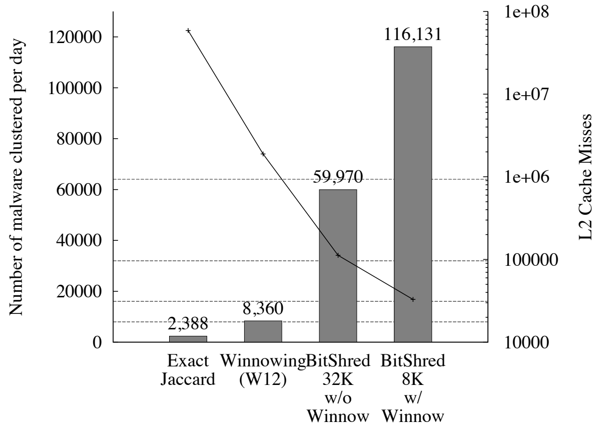


Figure 5: Overall malware clustered-per-day capabilities. We also report relative L1/L2 cache misses.

use set operations. These, in turn, are not efficient on real architectures. BitShred's bitvector fingerprints, on the other hand, are L1/L2 cache friendly.

Distributed BitShred. We have implemented the Hadoop version of BitShred, and performed several experiments to measure overall scalability and throughput. We use up to 655,360 samples in this experiment. Note all samples were unpacked as the goal of this experiment is to measure overall performance and not accuracy.

Figure 6 shows the BITSHRED-GEN fingerprint generation time. In this experiment, we utilized 80 map tasks for small datasets (20,480 $\sim$ 81,920) and 320 map tasks for large datasets (163,840 $\sim$ 655,360). The total time to create fingerprints for all samples was 5m 45s with BS8K (W12) and 4m 40s with BS32K (W1). The graph also shows a linear trend in the fingerprint generation time, e.g., halving the total number of samples to 327,680 samples approximately halves the generation time to about 2m 54s and 2m 25s, respectively. BITSHRED-GEN performance slightly dropped at 163,840 samples because the startup and shutdown overhead of each map dominates the benefit of utilizing more maps.

Figure 7 shows the amount of time for computing the pairwise distance for the same sample set. We utilized 200 map tasks for small datasets and 320 map tasks for large datasets. Given the values in the graph, we can work out the number of comparisons per second. For example, 163,840 samples requires approximately 1.3×10^{10} comparisons, and takes 10m 15s with BS8K (W12), which works out to 21,823,888 comparisons/sec. 327,680 samples requires about 5.4×10^{10} comparisons, and takes 40m 55s with BS8K (W12), which works out to a similar 21,868,402 comparisons/sec.

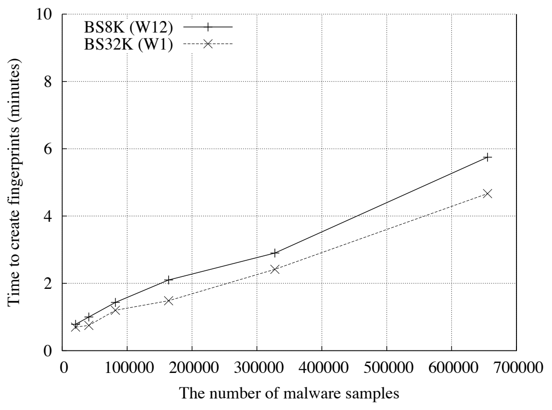


Figure 6: Performance of Distributed BITSHRED-GEN

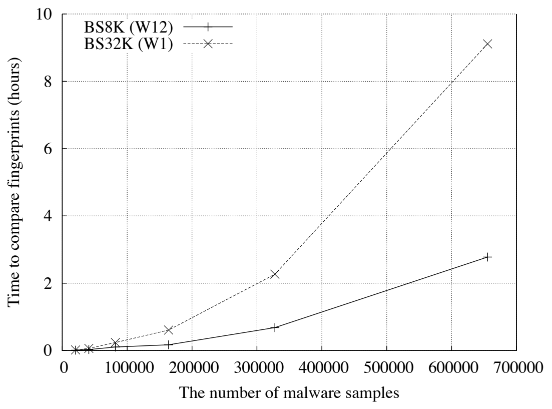


Figure 7: Performance of Distributed BITSHRED-JACCARD

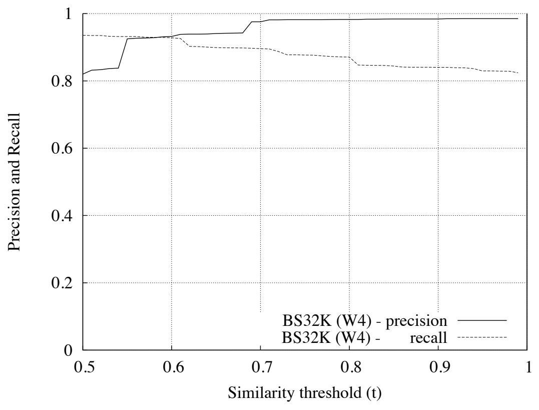


Figure 8: Precision and Recall (3,935 samples)

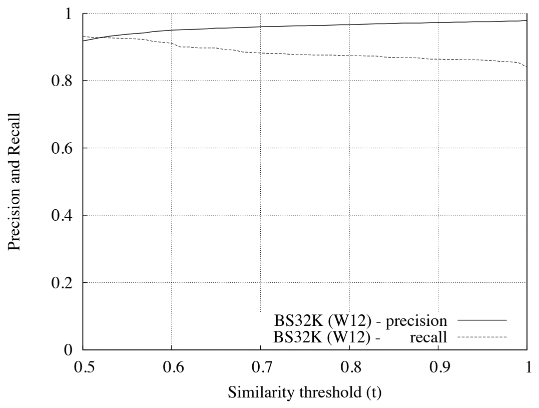


Figure 9: Precision and Recall (131,072 samples)

Overall, the distributed version achieved a pairwise comparison throughput of about 1.9×10^{12} per day. This works out to *full* hierarchical clustering over 1.9 million malware per day. In the case of *incremental* clustering, this works out to comparing over 190,000 malware per day against 10 million known malware that are already in a database.

Triage Tasks. Three common triage tasks are to automatically identify malware families via clustering [13], to identify the nearest neighbors to a particular malware sample [21], and to visualize malware by creating phylogenetic trees [23]. In this experiment we explore using BitShred with n-grams as the extracted features. While we stress that we are not advocating n-gram analysis, we also note it is interesting to see what the actual quality would be in such a system. We repeat these analysis in § 5.2 using dynamic behavior features.

• *Clustering.* We refer to how close a particular clustering is to the "correct" clustering with respect to labeled data set as the *quality* of a clustering. Overall quality will heavily depend upon the feature extraction tool (e.g., static or dynamic), the particular data set (e.g., because malware analysis often relies upon undecidable questions), and the quality of the reference data set.

To create a reference clustering data set, we used 30∼40 different anti-virus labels provided by VirusTotal [6]. First, we chose samples that were detected as malware by at least 20 anti-virus programs to get more reliable labels. We normalized and tokenized all the labels; then, we assigned the family name based upon only the tokens occurring at the majority of the detecting anti-virus programs. As a result, we had 3,935 samples.

The overall clustering quality is measured with respect to two metrics: precision and recall. Precision measures how well malware in separate families are put in different clusters, and recall measures how well malware within the same family are put into the same cluster. Formally, precision and recall are defined as:

$$\text{Precision} = \frac{1}{s}\sum_{i=1}^{c} \max(|C_i \cap R_1|, ..., |C_i \cap R_r|)$$
$$\text{Recall} = \frac{1}{s}\sum_{i=1}^{r} \max(|C_1 \cap R_i|, ..., |C_n \cap R_i|)$$

We clustered the reference data set 3,935 samples based upon n-grams. Figure 8 shows the overall quality of BitShred with Winnowing (BS32K (W4)). Surprisingly, simple n-gram analysis did quite well. When $t = 0.6$, BS32K (W4) clustering produced 200 clusters with a precision of 0.932 and a recall of 0.928 in 8 minutes.

For a larger-scale experiment, we unpacked 131,072 malware samples using off-the-shelf unpackers. We then clustered the malware and compared the identified families to a reference clustering using ClamAV labels [2]. Figure 9 shows the overall results with BitShred with Winnowing (BS32K (W12)). When $t = 0.57$, BS32K (W12) clustering produced 7,073 clusters with a precision of 0.942 and a recall of 0.922. It took about 27m with 256 map tasks.

• *Nearest Neighbor.* Hu *et al.* describe finding the nearest k-neighbors to a given sample as a common triage task [21] (§2.3). We have implemented similar functionality in BitShred by comparing the given malware to all other malware. We performed experiments finding the 5 nearest neighbors to randomly chosen malware samples on the 102,391 malware data set. We achieved the same 94.2% precision and 92.2% recall as above. The average time to find the neighbors was 6.8s (w/ BS8K) and 27s (w/ BS32K), using 25MB memory, with variance always under 1s.

• *Visualization.* We also have implemented several ways to visualize clustering within BitShred. First, we can create boxed malware

[2]Considerable amount of manual work is required to prepare a reference data set. For this reason, we simply used ClamAV as a reference for 131,072 samples.

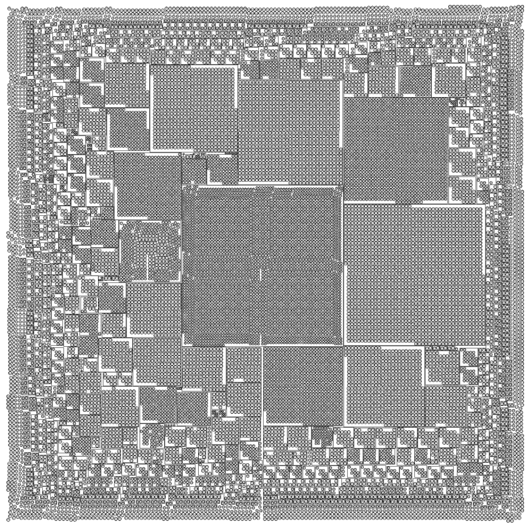


Figure 10: Clustering graph when $t = 0.57$

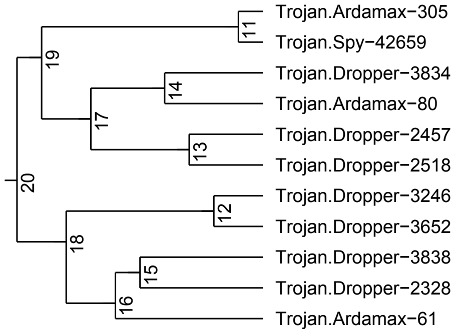


Figure 11: Lineage tree for a single malware family

graphs where each square represents a malware family, with circles representing individual samples. Figure 10 shows a clustering of 20,000 malware samples when $t = 0.57$ [3]. In the figure we can see larger families with many malware in the center, with the size of the family decreasing as we move to the edges. At the very edge are malware samples that cluster with no family.

Another way to visualize the results using BitShred is to create phylogenetic family trees based upon similarity [23]. The more alike two malware samples are, the closer they are on the tree. Figure 11 depicts an example tree created from our data set, labeled with ClamAV nodes. It is interesting to note ClamAV labels the malware as coming from three families: Spy, Dropper, and Ardamax. We manually confirmed that indeed all three were extremely similar and should be considered of the same family, e.g., Trojan.Ardamax-305 and Trojan.Spy-42659 are in different ClamAV families, but only differ in 1 byte.

5.2 BitShred with Dynamic Behaviors as Features

Static analysis may be fooled by advanced obfuscation techniques, which has led researchers to propose a variety of dynamic behavior-based malware analysis approaches, e.g., [12, 13, 28, 29, 33, 35]. One popular variant of this approach is to load the malware into a clean virtual machine. The VM is started, and observed behaviors such as system calls, conditional checks, etc. are recorded as features.

Bayer *et al.* provided us with their implementation of clustering, and the 2658 behavior profiles they used to measure accuracy from their paper [13]. In this data set, each behavior profile is a list of feature index numbers. The total number of features was 172260. In our experiments, we used only a 1KB fingerprint size since the number of features was relatively small.

[3]We pick 20,000 samples because larger numbers created graphs that hung our, and potentially the reviewers', PDF reader.

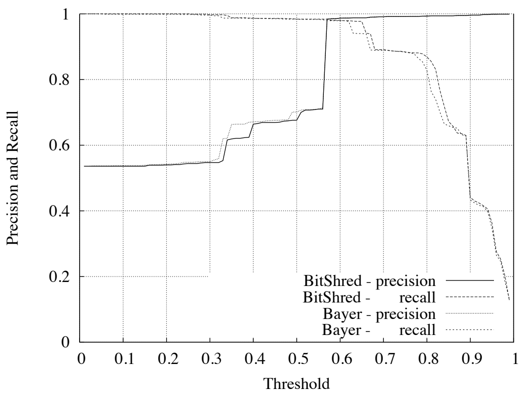


Figure 12: Clustering quality based upon behavior profiles

# of profiles	Clustering	Elapsed Time	Required Memory (max)
2,658	BAYER-EXACT	16s	86MB
	BITSHRED-EXACT	4s	12MB
75,692	BAYER-LSH	2h 25m 44s	4.3GB
	BITSHRED-SETBITS	24m 35s	89MB
	BITSHRED-EXACT	1h 2m 51s	89MB

Table 2: Scalability of Systems

As shown in Table 2, an exact clustering took 16s and 86MB of memory using the code from Bayer *et al.* BitShred took 4s (4x as fast) and used 12MB of memory (7x less memory). The average error was 2% using the 1KB fingerprint. Figure 12 depicts the exact clustering vs BitShred as a function of precision and recall. Both had the same precision of .99 and recall of .98 when $t = .61$. Overall, BitShred is faster and uses less memory, while not sacrificing accuracy for dynamic analysis feature sets.

Although Bayer *et al.* made the 2658 profiles they used for accuracy, they did not provide all 75,692 profiles they used when measuring performance. In order to measure performance on this size of data, we synthetically generated 73,034 variants using the 2658 as a basis. We then ran the code from Bayer *et al.* on 75,692 profiles using the same parameters as described in [13]: $k = 10$, $l = 90$, and $t = 0.7$. BAYER-LSH took 2h 25m 44s using 4.3GB of memory and performed 236,132,556 distance computations. BITSHRED-SETBITS [4] took 24m 35s (5.9x as fast) using 89MB of memory (49x less memory) and computed similarity of 1,021,322,219 pairs when $t = 0.7$. (Note that BITSHRED-SETBITS performed 4.3x more distance computations.) Even BITSHRED-EXACT at 1h 2m 51s outperformed (2.3x as fast) BAYER-LSH.

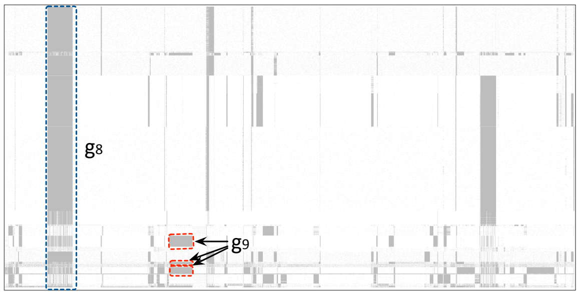
5.3 Semantic Feature Information

Finally, we used the co-clustering phase in BitShred to identify semantic distinguishing features among malware families. We performed a variety of experiments and found that, overall, co-clustering automatically identified both inter-family and intra-family semantic features. Typical features identified included distinguishing register keys set and internet hosts contacted.

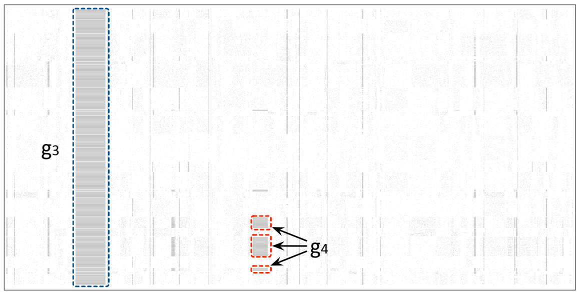
Co-clustering of Behavior-Based Profiles. We performed a full co-clustering on the entire dynamic analysis data set from § 5.2.

[4]We sorted the samples based upon the number of set bits in fingerprints. Then each sample only needed to be compared to the samples whose number of set bits are within the input threshold.

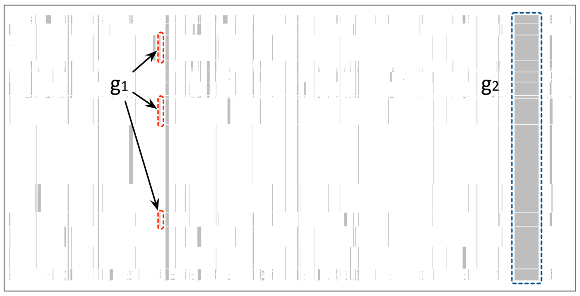
(a) A typical matrix before co-clustering.



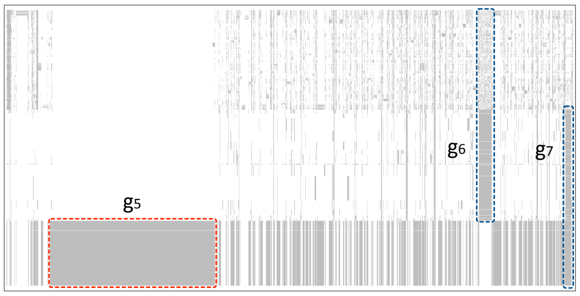
(b) Inter-family analysis based on dynamic behavior profile



(c) Intra-family analysis based on dynamic behavior profile



(d) Intra-family analysis based on static code analysis



(e) Inter-family analysis based on static code analysis

Figure 13: Feature extraction by co-clustering. Grey dots represent 1 in the binary matrix, i.e., the presence of a feature.

Figure 13a depicts the malware/feature matrix before co-clustering. We then co-clustered, which took 15 minutes.

Figure 13b shows the complete results. The checkerboard pattern corresponds to the sub-matrices identified as being homogeneous, i.e., corresponding malware/feature pairs that are highly correlated. For example, the large dark sub-matrix labeled g_8 corresponds to the fact that most malware had the same memory-mapped files including WS2HELP.dll, icmp.dll, and ws2_32.dll. The sub-matrix g_9 shows a commonality between two families, but no others. The commonality corresponds to opening the file \Device\KsecDD.

Figure 13c focuses on only the 717 samples in the Allaple malware family. One semantic feature, labeled g_3, is that almost all samples use the same memory-mapped files such as winrnr.dll, WS2HELP.dll, icmp.dll, and ws2_32.dll. More importantly, we also found that many family members were distinguished by the register entry they create (e.g., HKLM\SOFTWARE\CLASSES\-CLSID\{7BDAB28A-B77E-2A87-868A-C8DD2D3C52D3} in one sample) and the IP address they connect to, e.g., one sample connected to 24.249.139.x while another connected to 24.249.150.y (shown as g_4).

Co-clustering of n-gram features. We also experimented with co-clustering using the n-gram features. Figure 13d shows intra-family co-clustering for the Trojan.OnlineGames malware family. The features labeled g_2 correspond to all code from the code entry to a particular point that overlap. The feature set g_1 corresponds to new functionality in a few variants that makes tcp connections to a new host not found in previous variants.

We also performed inter-family analysis. In this set of experiments we envision that an analyst uses co-clustering to mine differences and similarities between malware family members or between malware families. We picked the Trojan.Dropper, Trojan.Spy, Trojan.OnlineGames, and Adware.Downloader families, which have 1280 total members. The total time taken by co-clustering was 10 minutes (about 2s per sample), with about 1 minute for each column and row iteration. We used 10 maps for each row iteration and 64 maps for column iteration.

Figure 13e shows the resulting co-clustering. Trojan.Dropper and Trojan.Spy were grouped together by co-clustering. This is accurate: we manually confirmed that the samples we have from those families are not well-distinguished. The submatrix labeled g_5 is one distinguishing feature corresponding to Adware.Downloader connecting to a particular host on the internet. The submatrix labeled g_6 corresponds to data section fragments shared between the Trojan family, but not present in Adware. The submatrix labeled g_7 corresponds to shared code for comparing memory locations. This code is shared between Adware.Downloader and Trojan.OnlineGames, but not Trojan.Spy/Trojan.Downloader.

6. RELATED WORK

We are not the first to propose the need for large-scale malware analysis and triage, e.g., [13, 21, 31] have similar goals. As discussed in § 2.3, the main difference in our work is scalability and co-clustering to automatically identify semantic features. Our work builds on per-sample analysis and feature extraction, which is an extremely active area of research including unpacking research [18, 20, 27, 33, 35], and even entire conferences like DIMVA.

While we focus on analyzing malware binaries, Perdisci *et al.* [32] have explored clustering based upon network behavior features. Their similarity metric is based upon the weighted combination of Euclidean distance, Levenshtein distance, and Jaccard distance. While BitShred could compute Hamming distance and Jaccard distance, we leave adapting the analysis, implementation, and experimentation for other distance metrics as future work.

Li *et al.* [26] argue it is difficult to get ground truth for clustering accuracy. In our setting the main metric is how close the feature-hashed version of clustering comes to the quality of an exact Jaccard clustering, which is different than looking for ground truth. Nonetheless, we do report overall clustering accuracy, where all the caveats from [26] apply.

7. CONCLUSION

In this paper we have presented *BitShred*, a system for large-scale malware triage and similarity detection. The key idea behind BitShred is using feature hashing to reduce the high-dimensional feature space in malware analysis, which is supported by theoretical and empirical analysis. Our approach makes inter-malware comparisons in typical large-scale triage tasks such as clustering and finding nearest neighbors up to 2,365x faster than existing methods while using less memory. As a result, BitShred scales to current and future malware volumes where previous approaches do not. We have also developed a distributed version of BitShred where $2x$ the hardware gives $2x$ the performance. In our tests, we show we can scale to up to clustering over 1.9 million malware per day. Finally, we have developed novel techniques based upon co-clustering to extract semantic features between malware samples and families. The extracted features provide insight into the fundamental differences and similarities between and within malware data sets.

Acknowledgment

This research was supported in part by sub-award PO4100074797 from Lockheed Martin Corporation originating from DARPA Contract FA9750-10-C-0170 for BAA 10-36. We would like to thank Malware Analysis System (aka CWSandbox) and Offensive Computing for providing malware data sets. We also would like to thank VirusTotal for providing the private API to access their resources.

References

[1] Apache hadoop. http://hadoop.apache.org/.

[2] CMU Cloud Computer Cluster. http://www2.pdl.cmu.edu/~twiki/cgi-bin/view/OpenCloud/ClusterOverview.

[3] Malware Analysis System. http://mwanalysis.org/.

[4] Offensive Computing. http://www.offensivecomputing.net/.

[5] SimMetrics. http://sourceforge.net/projects/simmetrics/.

[6] VirusTotal. http://www.virustotal.com/.

[7] zynamics bindiff. http://www.zynamics.com/bindiff.html.

[8] Symantec internet security threat report. http://www.symantec.com/business/theme.jsp?themeid=threatreport, April 2010.

[9] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts*, 2004.

[10] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):177–122, 2008.

[11] J. Attenberg, K. Weinberger, A. Dasgupta, A. Smola, and M. Zinkevich. Collaborative email-spam filtering with the hashing-trick. In *Proceedings of the Sixth Conference on Email and Anti-Spam*, 2009.

[12] M. Bailey, J. Oberheide, J. Andersen, F. J. Z. Morley Mao, and J. Nazario. Automated classification and analysis of internet malware. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection*, September 2007.

[13] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the Network and Distributed System Security Symposium*, 2009.

[14] D. Bernstein. http://www.cse.yorku.ca/~oz/hash.html.

[15] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[16] D. Chakrabarti, S. Papadimitriou, D. Modha, and C. Faloutsous. Fully automatic cross associations. In *Proceedings of ACM SIGKDD*, August 2004.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2004.

[18] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM CCS*, 2008.

[19] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. In *Proceedings of ACM Symposium on Discrete Algorithms (SODA)*, 1998.

[20] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 98–115, 2008.

[21] X. Hu, T. cker Chiueh, and K. G. Shin. Large-scale malware indexing using function call graphs. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2009.

[22] N. Jain, M. Dahlin, and R. Tewari. Using bloom filters to refine web search results. In *Proceedings of Eighth International Workshop on the Web and Databases (WebDB 2005)*, June 2005.

[23] M. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1):13–23, November 2005.

[24] G. Karypis. CLUTO: a clustering toolkit, release 2.1.1. Technical report, University of Minnesota, 2003.

[25] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2721–2744, Dec. 2006.

[26] P. Li, L. Lu, D. Gao, and M. Reiter. On challenges in evaluating malware clustering. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, 2010.

[27] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, generic, and safe unpacking of malware. In *In Proceedings of the Annual Computer Security Applications Conference*, 2007.

[28] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[29] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of the Annual Computer Security Applications Conference*, 2007.

[30] S. Papadimitrou and J. Sun. Disco: Distributed co-clustering with map-reduce. In *Proceedings of ICDM*, 2008.

[31] R. Perdisci, A. Lanzi, and W. Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recogn. Lett.*, 29(14):1941–1946, 2008.

[32] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of HTTP-based malware and signature generation using malicious network traces. In *Proceedings of NSDI*, 2010.

[33] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of Computer Security Applications Conference*, December 2006.

[34] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the ACM SIGMOD/PODS Conference*, 2003.

[35] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[36] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 2009.

[37] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smole, A. Strehl, and V. Vishwanathan. Hash kernels. In *Proceedings of the* 12^{th} *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2009.

[38] A. Walenstein and A. Lakhotia. The software similarity problem in malware analysis. In *Duplication, Redundancy, and Similarity in Software*, 2007.

[39] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large-scale multitask learning. In *Proceedings of ICML*, 2009.

APPENDIX

A. PROOF OF THEOREM 1

Our analysis shows that with high probability, the Jaccard index $\frac{|g_i \cap g_j|}{|g_i \cup g_j|}$ is well approximated by the $\frac{S(f_i \wedge f_j)}{S(f_i \vee f_j)}$, where f_i and f_j are the fingerprints of g_i and g_j. Throughout this analysis, we let c denote the number of shared elements between sets g_i and g_j; note that the Jaccard index $\frac{|g_i \cap g_j|}{|g_i \cup g_j|}$ is then $\frac{c}{2N-c}$. The focus of our analysis is to show that the ratio $\frac{S(f_i \wedge f_j)}{S(f_i \vee f_j)}$ is close to $\frac{c}{2N-c}$ with high probability (unlike other analyses [22] that restrict their focus to computing the expected value of $S(f_i \wedge f_j)$). We make the usual assumption that the hash functions used are k-wise independent.

We first consider the union $g_i \cup g_j$. We note that the bitvector obtained by computing the bitwise-or of the two fingerprints f_i and f_j is equivalent to the bitvector that would be obtained by directly inserting all the elements in $g_i \cup g_j$, if the same k hash functions are used on a bitvector of the same size.

Let the random variable U denote the number of bits set to 1 in $f_i \vee f_j$. Note that the set $g_i \cup g_j$ contains $2N - c$ elements. If these elements are inserted into a bitvector of size m with k hash functions, the probability q_u that a bit is set to 1 is: $1 - \left(1 - \frac{1}{m}\right)^{k(2N-c)}$. We can use this to compute the expected value of U:

$$E[U] = mq_u = m\left(1 - \left(1 - \frac{1}{m}\right)^{k(2N-c)}\right) \qquad (3)$$

As U is tightly concentrated around its expectation [15], we get:

$$Pr[|U - E[U]| \geq \epsilon m] \leq 2e^{-2\epsilon^2 m^2/(2N-c)k} \leq 2e^{-2\epsilon^2 m^2/Nk}.$$

Next, we consider the intersection $g_i \cap g_j$. Let the random variable I denote the number of bits set to 1 in $f_i \wedge f_j$. A bit z is set in $f_i \wedge f_j$ in one of two ways: (1) it may be set by some element in $g_i \cap g_j$, or (2) it may be set by some element in $g_i - (g_i \cap g_j)$ and by some element $g_j - (g_i \cap g_j)$. Let I_z denote the indicator variable for bit z in $f_i \wedge f_j$. Then,

$$\begin{aligned} Pr[I_z = 1] \;=\; & \left(1 - \left(1 - \frac{1}{m}\right)^{kc}\right) + \\ & \left(1 - \frac{1}{m}\right)^{kc}\left(1 - \left(1 - \frac{1}{m}\right)^{k(|g_i|-c)}\right) \\ & \cdot \left(1 - \left(1 - \frac{1}{m}\right)^{k(|g_j|-c)}\right) \end{aligned}$$

which may be simplified as:

$$1 - \left(1 - \frac{1}{m}\right)^{kN} - \left(1 - \frac{1}{m}\right)^{kN} + \left(1 - \frac{1}{m}\right)^{k(2N-c)}.$$

With linearity of expectation, we can compute $E[I]$ as $\sum_z Pr[I_z = 1]$, which reduces to:

$$E[I] = m\left(1 - 2\left(1 - \frac{1}{m}\right)^{kN} + \left(1 - \frac{1}{m}\right)^{k(2N-c)}\right). \quad (4)$$

Note that the random variables $I_1, I_2 \ldots I_m$ are negatively dependent, and so we can apply Chernoff-Hoeffding bounds to compute the probability that I deviates significantly from $E[I]$: e.g., $Pr[I \geq E[I](1+\epsilon_2) \leq e^{-mq\epsilon_2^2/3}$, where $q = 1 - \left(1 - \frac{1}{m}\right)^{kN} - \left(1 - \frac{1}{m}\right)^{kN} + \left(1 - \frac{1}{m}\right)^{k(2N-c)}$.

We now turn to the ratio $\frac{S(f_i \wedge f_j)}{S(f_i \vee f_j)}$; let the random variable Y denote this ratio. We have just shown that U and I are both likely to remain close to their expected values, and we can use this to compute upper and lower bounds on Y – since U and I lie within an additive or multiplicative factor of their expectations with probability at least $1 - 2e^{-mq\epsilon_2^2/3}$ and $1 - 2e^{-2\epsilon^2 m^2/Nk}$ respectively, we can derive upper and lower bounds on Y that hold with probability at least $1 - 2e^{-mq\epsilon_2^2/3} - 2e^{-2\epsilon^2 m^2/Nk}$.

To do this, we first simplify the quantities $E[U]$ and $E[I]$. Assuming that $m \gg 2kN$, we can approximate $E[U]$ and $E[I]$ by discarding the higher-order terms in each of binomials in 3 and 4:

$$\begin{aligned} E[U] &\geq m\left(1 - \left(1 - \frac{k(2N-c)}{m}\right)\right) \\ &= mk\left(\frac{2N-c}{m}\right) = k(2N-c). \end{aligned}$$

Likewise, we can approximate $E[I]$ as:

$$\begin{aligned} E[I] &\leq m\left(1 - 2\left(1 - \frac{kN}{m}\right) + \left(1 - \frac{k(2N-c)}{m}\right)\right) \\ &= mk\left(\frac{c}{m}\right) = ck. \end{aligned}$$

Using these approximations for $E[I]$ & $E[U]$, we see that $Y \leq \frac{c(1+\epsilon_2)}{2N-c-m\epsilon}$, with probability at least $1 - e^{-mq\epsilon_2^2/3} - 2e^{-2\epsilon^2 m^2/Nk}$. We can compute a similar lower bound for Y, i.e., $Y \geq \frac{c(1-\epsilon_2)}{(2N-c)+m\epsilon}$, with probability at least $1 - e^{-mq\epsilon_2^2/2} - 2e^{-2\epsilon^2 m^2/Nk}$. Thus, this shows that with high probability, the ratio $\frac{S(f_i \wedge f_j)}{S(f_i \vee f_j)}$ is close to the Jaccard index $\frac{c}{2N-c}$, for appropriately chosen values of m and k. We have thus proven our Theorem 1.

Lastly, we give an example to illustrate our bounds in our application scenario. Suppose we set $\epsilon m \geq 5$, $m \approx 1000N$, $k = 6$. Then, our analysis shows us that with probability at least 95%, $Y \in \left(\frac{c(1-\frac{1}{\sqrt{2c}})}{2N-c+5}, \frac{c(1+\frac{4}{\sqrt{c}})}{2N-c-5}\right)$, i.e., that ratio of the bits set to the union is very close to the Jaccard index.

B. EXTENSION

Containment. BITSHRED-JACCARD measures the proportional similarity between features. However, we may want to also measure when one feature set is contained within another, e.g., whether one malware is completely contained in another code. For example, suppose malware A is the composition of two malware samples B and C, and suppose $|B| \gg |C|$. Then the similarity between A and C will be proportionally very low. An alternative similarity metric for this case can be given as:

$$\text{BITSHRED-JACCARD}_c(f_a, f_b) = \frac{S(f_a \wedge f_b)}{S(f_b)},$$

when f_i is the fingerprint for malware s_i and $|s_a| \gg |s_b|$.